

# Applied Static Analysis

---

## Why Static Analysis?

---

Software Technology Group  
Department of Computer Science  
Technische Universität Darmstadt  
[Dr. Michael Eichberg](#)

Some of the following material is taken from the script "Static Program Analysis" from Anders Møller and Michael I. Schwartzbach and from the Slides "Mechanics of Static Analysis" from David Schmitt.

---

# Questions that we may ask about a program:

---

- Will the variable `x` always contain the same value?
  - Which objects can variable `x` points to?
  - What is a lower/upper bound on the value of the integer variable `x` ?
  - Which methods are called by a method invocation?
  - How much memory is required to execute the program?
  - Will it throw a `NullPointerException` ?
  - Will it leak sensitive data? Will data from component `a` flow to component `b` ?
- 

Buggy C-Program:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main(int argc, const char * argv[]) {
    char *p, *q;
    p = NULL;
    printf("%s",p);
    q = (char *)malloc(100);
    p = q;
    free(q);
    *p = 'x';
    free(p);
    p = (char *)malloc(100);
    q = (char *)malloc(100);
    q = p;
    strcat(p,q);
}
```

Line 7: The behavior of `printf` is undefined in the string referred to by `%s` is `NULL`. (Actually, the behavior of the program may even change depending on the optimizations done by the compiler!)

Line 11: We have a use after free (detected by some modern compilers).

Line 14 in combination with line 15: Resource leakage; the memory area is not freed.

Line 16: `q` and `p` overlap - this will lead to a segmentation fault.

---

# What is Static Analysis?

---

A static analysis of a program is a **sound**, *finite*, and **approximate** calculation of the program's execution semantics which helps us to solve practical problems.

**Sound** means that the analysis is consistent with the actual concrete execution semantics and that the result models all possible executions of the analyzed program.

An analysis is **conservative** if all errors lean to the same side. For example, if we want to remove useless synchronization primitives, then an analysis should always only identify those primitives as useless that are definitively not required. In case of doubt, it should return that the synchronization primitive is required.

Sound static analyses are generally *overapproximations* which describe a superset of the program's possible/observable executions. However, the analyses should still be precise enough to be practically useable. For example, an analysis that would list all methods as being possible targets for a method invocation would be impractical.

Sometimes, analyses are deliberately not sound and compute only *underapproximations*; i.e., the analysis does not describe all observable executions. This is, e.g., done to improve an analysis' scalability and/or to improve an analyses precision. For practical analyses it may be more important to produce fewer so called *false positives* then to report more *potential issues*.

A static analysis is finite in the sense that it will terminate for any given program. The analysis will not execute the program.

---

# Purposes of Code Analyses

---

- Finding code smells.
- Quality assessments.
- Improving the quality of the code.
- Support debugging of code.
- Optimizing the code.

Not every analysis which does not execute the code is a *static analysis*!

Code smells range from simple issues such as useless computations, which have no lasting effect on the program state, over bugs which will potentially lead to dead locks and race conditions up to security vulnerabilities which enable attackers to steal sensitive data. *Here, we use the term code smells to refer to those issues in the code that a developer (typically) did not add deliberately.*

Analyses which, e.g., identify code duplicates or which compute metrics are frequently used to perform quality assessments; however, these analyses are not static analyses.

Refactorings (e.g., extract method) often required (sound) static analyses to ensure that the refactoring does not break the code.

Debugging is supported, e.g., by computing slices which encompass all statements in a program that affect a given statement.

Classical code optimizations are, e.g., removing dead code, hoisting loop invariants, removing useless synchronization, avoiding runtime checks to detect `null-pointer` accesses, allocating objects on the stack, or avoiding `array index out of bounds` checks.

---

# Finding Programming Bugs

---

[JavaDoc of java.lang.BigDecimal.setScale](#):

[...] Note that since BigDecimal objects are immutable, calls of this method do not result in the original object being modified, contrary to the usual convention of having methods named setX mutate field X. Instead, setScale returns an object with the proper scale; the returned object may or may not be newly allocated.

```
class X {  
    private long scale;  
    X(long scale) { this.scale = scale; }  
    void adapt(BigDecimal d){  
        d.setScale(this.scale);  
    }  
}
```

In this case the bug is that the `setScale` method is side-effect free (though the name `setScale` suggests something different) and therefore the call `d.setScale(...)` is useless. Hence, this code smells.

There is typically more than one way to find certain bugs and not all require (sophisticated) static analyses!

---

# Finding Bugs Using Bug Patterns

Code smells such as useless calls to `setScale` can easily be described and found using *bug patterns*. A bug pattern is an abstract description of a problem that occurs over and over again and which can be used to identify concrete instances of the problem.

A complete implementation of an analysis in the OPAL static analysis framework which analyzes a Java project's bytecode is shown next:

```
import org.opalj.br._
import org.opalj.br.instructions.{INVOKEVIRTUAL, POP}
val p = analyses.Project(org.opalj.bytecode.JRELibraryFolder) // <= analyze the JRE
p.allMethodsWithBody.foreach{m =>
  m.body.get.collectPair{
    case (
      i @ INVOKEVIRTUAL(ObjectType("java/math/BigDecimal"), "setScale", _),
      POP
    ) => i
  }
  .foreach(i => println(m.toJava(i.toString)))
}
```

The above static analysis uses Scala's pattern matching (line 8 and 9) to find the sequence of instructions that calls the `setScale` method ( `INVOKEVIRTUAL` ) and then immediately throws the returned result away ( `POP` ).

Real instance in Java 8u191: [com.sun.rowset.CachedRowSetImpl.updateObject\(int,Object,int\)](#)

```
public void updateObject(int columnIndex, Object x, int scale) throws SQLException {
  // sanity check.
  checkIndex(columnIndex);
  // make sure the cursor is on a valid row
  checkCursor();
  ^
  int type = RowSetMD.getColumnType(columnIndex);
  if (type == Types.DECIMAL || type == Types.NUMERIC) {
    ((java.math.BigDecimal)x).setScale(scale);
  }
  getCurrentRow().setColumnObject(columnIndex, x);
}
```

# Finding Bugs Using Bug Patterns (Assessment)

---

- Advantages:
  - very fast and scale well to very large programs
  - (usually) simple to implement
- Disadvantages:
  - typically highly specialized to specific language constructs and APIs
  - requires some understanding how the issue typically manifests itself in the (binary) code
  - small variations in the code may escape the analysis
  - to cover a broader range of similar issues significant effort is necessary

Finding bugs using bug patterns is supported by, e.g., [FindBugs/SpotBugs](#).

---

# Finding Bugs Using Machine Learning

---

The core idea is to use code that is known to be buggy and code that is expected to be correct to train a machine learning model. The resulting classifier is then used to find buggy code in new code. Buggy code can, e.g., be extracted from public repositories by analyzing the commit history and looking for issues. An alternative is to inject code that will most likely result in buggy code.



An interesting approach which is able to find bugs which could not be found with others approaches is described by Michael Pradel et al.<sup>1</sup>. The idea is to find contradicting code snippets.

---



# Finding Bugs Using Machine Learning

---

Guess the problem of the following JavaScript code snippet:

```
function setPoint(x, y) { ... } // <= given  
  
var x_dim = 23;  
var y_dim = 5;  
setPoint(y_dim, x_dim);
```

In this example, the order of the parameters was confused. The idea to find such issues is based on checking the names of the variables passed to the method in relation to the names of the parameters. This is generally non-trivial, because the names of the parameters and the passed values generally don't match!

---

# Finding Bugs Using Machine Learning (Assessment)

---

- Finds bugs that are practically impossible to find using other approaches; hence, often complementary to classic static analyses and also bug pattern based analyses.
- Requires the analysis of a huge code base; it may be hard to find enough code examples for less frequently used APIs.

Today, a lot of code can be found on open source code repositories such as BitBucket, GitHub and SourceForge; even for niche languages. This generally facilitates all kinds of *big code* based analyses.

---

# Finding Bugs by Mining Usage Patterns (Idea)

---

Is the following code buggy?

```
Iterator<?> it = ...
it.next();
while (it.hasNext()) {
    it.next();
    ...
}
```

The core idea is to first extract usage patterns from existing code. E.g., that calling `close` on a stream should be the last operation or that an Iterator's `next` method should always be called after a `hasNext` method. After extracting the usage patterns from a given code base, a corresponding algorithm is applied to new code to extract its usage patterns and to check for deviations. However, as shown recently, all existing approaches are far away from being practically useable <sup>2</sup>.

An alternative (manual) approach would be to let developers explicitly specify the valid usage patterns (e.g., using state machines) and then to check for deviations between the implementations and the explicit specification. However, so far this approach is too expensive.

---

# Finding Bugs Using Generic Static Code Analysis

---

```
class com.sun.imageio.plugins.png.PNGMetadata{
    void mergeStandardTree(org.w3c.dom.Node) {
        [...]
        if (maxBits > 4 || maxBits < 8) {
            maxBits = 8;
        }
        if (maxBits > 8) {
            maxBits = 16;
        }
        [...]
    }
}
```

The condition of the first `if` (line 4) will always be true. Every possible int value will either be larger than four or smaller than eight. Hence, line 5 will **always** be executed and therefore the condition of the second `if` will never be true; `maxBits` will never be 16.

This bug can be found using a very generic analysis that just tries to be very precise (e.g., by computing the interval of the values stored in an int variable) and which then looks for dead code. <sup>3</sup>

---

# Finding Bugs Using Generic Static Code Analysis

---

```
class sun.font.StandardGlyphVector {
    private int[] glyphs; // always
    public int getGlyphCharIndex(int ix) {
        if (ix < 0 && ix >= glyphs.length) {
            throw new IndexOutOfBoundsException("'" + ix);
        }
    }
}
```

The condition (line 4) will never be `true` ; `glyphs.length` will either be a value equal or larger than `0` or will throw a `NullPointerException` . Hence, there is no effective validation of the parameter `ix` .

This bug can be found by the same (type of) analysis as the previous issue.

---

# Finding Bugs Using Generic Static Code Analysis

---

```
class sun.tracing.MultiplexProviderFactory {  
    public void uncheckedTrigger(Object[] args) {  
        [...]  
        Method m = Probe.class.getMethod(  
            "trigger",  
            Class.forName("[java.lang.Object]")  
        );  
        m.invoke(p, args);  
    }  
}
```

The `m.invoke` statement will never be executed, because the string passed to `Class.forName` is invalid.

This bug can be found by the same (type of) analysis as the previous issue (but requires – at least rudimentarily – handling of reflection.)

---

# Finding Bugs Using Generic Static Code Analysis

---

```
package com.sun.corba.se.impl.naming.pcosnaming;
class NamingContextImpl {
    public static String nameToString(NameComponent[] name)
    [...]
        if (name != null || name.length > 0) {
            [...]
        }
    [...]
}
}
```

If `name` is `null` the test `name.length > 0` (line 5) will be executed and will result in a `NullPointerException`. In this case, we have confused logical operators. The developer most likely wanted to use `&&`.

This bug can be found by the same (type of) analysis as the previous issues (but requires – at least rudimentarily – handling of null values.)

---

# Finding Bugs Using Static Code Analysis ?

---

Found in `javax.crypto.CryptoPolicyParser` (Java 8u25):

```
private boolean ...isConsistent(  
    String alg,  
    String exemptionMechanism,  
    Hashtable<String, Vector<String>> processedPermissions) {  
    String thisExemptionMechanism =  
        exemptionMechanism == null ? "none" : exemptionMechanism;  
    if (processedPermissions == null) {  
        processedPermissions = new Hashtable<String, Vector<String>>();  
        Vector<String> exemptionMechanisms = new Vector<>(1);  
        exemptionMechanisms.addElement(thisExemptionMechanism);  
        processedPermissions.put(alg, exemptionMechanisms);  
        return true;  
    }  
    [...]  
}
```

The code shown in line 7 to 11 has no sideeffect; however, identifying this situation in a generic manner requires sophisticated analyses.

---



# Finding Bugs Using (Highly) Specialized Static Analyses

---

Do you see the security issue?

```
Cipher c = Cipher.getInstance("DES/CBC/PKCS5PADDING")
```

In this case an insecure algorithm (DES) is used.

Security bugs related to the Java Cryptographic API can be found using [4](#). CogniCrypt's underlying analysis performs an in-depth data-flow analysis to ensure that the number of false positives is very low. (In this case, false negatives are not acceptable and therefore the analyses issues a warning if the analysis is not conclusive.)

---

# (True|False) (Positives|Negatives)

---

- a **true positive** is the correct finding (of something relevant)
- a **false positive** is a finding that is incorrect (i.e., which can't be observed at runtime)
- a **true negative** is the correct finding of no issue.
- a **false negative** refers to those issues that are not reported.

Typically the goal of every static analysis is to only report true positives and to avoid as many false positives as possible. Often an analysis is considered useable if 80% of all findings are true positives. False positives are typically caused by imprecisions of the analysis. False negatives are usually not a (major) concern of static analyses; however, they are a primary concern of formal approaches.

---

# Unguarded Access - True Positive ?

---

```
void printIt(String args[]) {  
    if (args != null) {  
        System.out.println("number of elements: " + args.length);  
    }  
    for (String arg : args) {  
        System.out.println(arg);  
    }  
}
```

In this case, the first access of `args` ( `args.length` in line 3) is guarded by an explicit null check. However, the second access (line 5) isn't! Hence, we could detect such code smells (here, lack of inner consistency) using a basic analysis that detects object accesses (e.g., `args.length` ) that appear in a guarded context ( `if (o != null)` ) and also outside of an explicitly guarded context.

---

# Implicitly Guarded Access

---

```
void printReverse(String args[]) {  
    int argscount = 0;  
    if (args != null) {  
        argscount = args.length;  
    }  
    for (int i = argscount - 1; i >= 0; i--) {  
        System.out.println(args[i]);  
    }  
}
```

This code is correct, but the analysis sketched previously would return a false positive in this case; the guard is implicitly established: `argscount` is only larger than 0 if `args` is definitively not null.

---

# Irrelevant True Positives

---

Better static analyses can sometimes prove assertions to always hold; which are then useless.

Let's assume that the following function is only called with `non-null` parameters.

In that case the `assert` statement is useless, but adding an assert statement (even in such cases) is a common best practice!

```
private boolean isSmallEnough(Object i) {  
    assert(i != null);  
    Object o = ""+i;  
    return o.length < 10;  
}
```

Don't use assert for validating parameters of non-private methods, i.e., if you don't have full control of the calling context.

---

# Complex True Positives

---

The cast in line 5 will fail:

```
GeneralPath result = new GeneralPath(GeneralPath.WIND_NON_ZERO);  
[...]  
if (dx != 0 || dy != 0) {  
    AffineTransform tx = AffineTransform.getTranslateInstance(dx, dy);  
    result = (GeneralPath)tx.createTransformedShape(result);  
}
```

Given the available code, it is non-obvious that the cast fails: an affine transformation of a `GeneralPath` is performed. An investigation of `createTransformedShape` would reveal that the actual type that is returned is a `Path2D.Double`. This still suggests that a cast to `GeneralPath` is reasonable! However, when you then study the class hierarchy, you'll realize that `Path2D.Double` is not a subtype of `GeneralPath`.

---

# Complex True Positives - Assessment

---

The sad reality:

[...] the general trend holds; a not-understood bug report is commonly labeled a false positive, rather than spurring the programmer to delve deeper. The result? We have completely abandoned some analyses that might generate difficult to understand reports. <sup>5</sup>

---

# Soundness

---

[...] in practice, soundness is commonly eschewed: we [the authors] are not aware of a single realistic whole-program analysis tool [...] that does not purposely make unsound choices.

[...]

Soundness is not even necessary for most modern analysis applications, however, as many clients can tolerate unsoundness. <sup>6</sup>

Reasons for soundness: engineering effort or (likely) costs w.r.t. the precision and scalability of the resulting analysis.

Most practical analyses have a sound core w.r.t. some language features and APIs. In particular the most relevant language features and APIs are typically soundly supported. Such analyses are called **soundy**; i.e., the analyses are concerned with soundness, but do not support *a well identified set of features*. Unfortunately, often the set of features that is unsoundly handled is not at all well identified as often suggested.

---



# Soundness - Java

---

Common features that are often not soundly handled in Java:

1. Intents (in Android Programs)
2. Reflection (*often mentioned in research papers*)
3. Native methods (*often mentioned in research papers*)
4. Dynamic Class Loading / Class Loaders (*sometimes mentioned in research papers*)
5. (De)Serialization (*often not considered at all*)
6. Special System Hooks (e.g., `shutdownHooks` ) (*often not considered at all*)
7. "Newer" language features

Examples in C/C++ are (1) `setjmp` / `longjmp` or (2) pointer arithmetic.

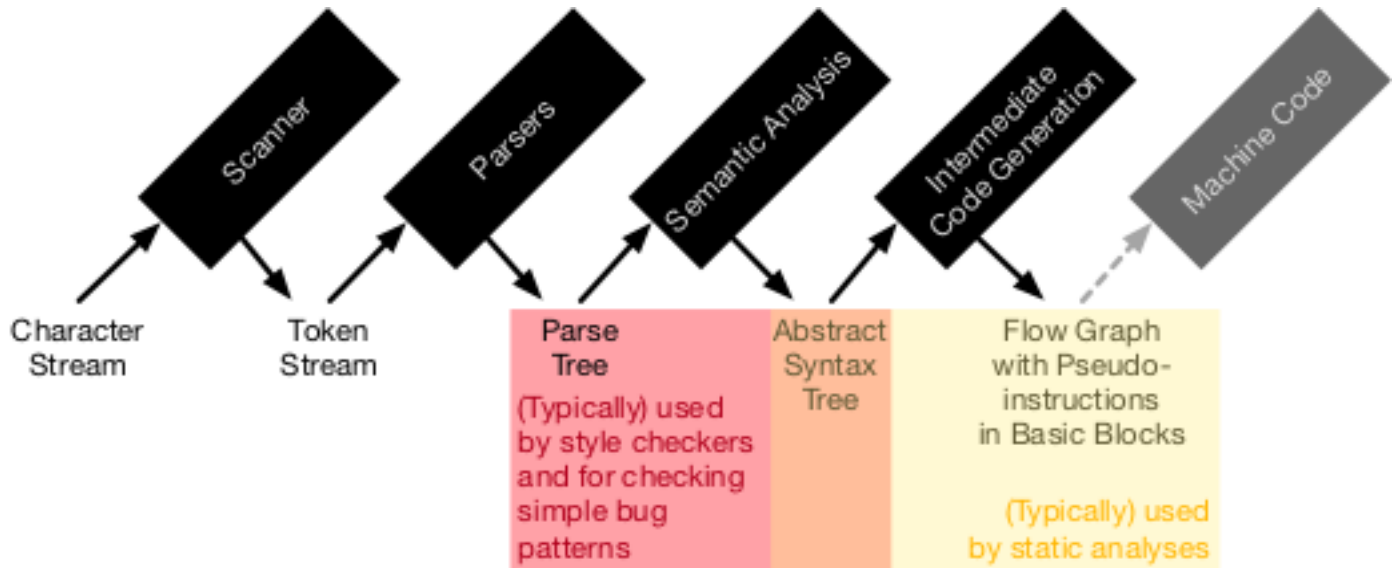
Examples in Javascript are (1) `eval` if the code is not *trivially* known or (2) data-flow through the DOM.

---

# The Relation between Compilers and Static Analyses

---

Usually (static) code analyses are built on top of the results of the first phases of compilers.



# The Relation between Compilers and Static Analyses

---

Source Code:

```
i = j + 1;
```

Tokens:

```
Ident(i) WS Assign WS Ident(j) WS Operator(+) WS Const(1) Semicolon
```

AST with (type) annotations:

```
AssignmentStatement(  
  target    = Var(name=i, type=Int),  
  expression = AddExpression(  
    type = Int,  
    left  = Var(name=j, type=Int),  
    right = Const(1))
```

In the (later) intermediate representations, nested control-flow and complex expressions are unraveled. Intermediate values are given explicit names. The instruction set is usually limited. All of that facilitates static analysis (but renders syntax oriented code analyses impossible).

---

# (Classical) Compiler Optimizations

---

- common subexpression elimination
- constant folding and propagation
- loop fusion
- loop invariant code motion
- code-block reordering
- dead-code elimination
- ...

often rely on the identification of the so-called *control dependencies* which are computed using control-flow graphs (CFGs). Though the optimizations as such are already relevant (e.g., to help deobfuscate code), the underlying techniques, such as, loop identification or live-variable analyses are also relevant for static analyses on their own.

Traditional analyses<sup>7</sup> often rely on a program's control-flow graphs to compute the control dependencies, but make unrealistic assumptions about the structure of the code. Exception handling and infinite loops which lead to methods with multiple exit points or no exit points are often neglected<sup>8</sup>. Standard techniques to handle such methods (like **picking** an arbitrary instruction and making it the exit instruction in case of an infinite loop) potentially lead to incorrect control dependencies.

---

# (Control-)Flow Graph (CFG)

---

Next, we repeat the basics of control-flow graphs and how to compute control dependencies. Both are needed by static analysis.

Given a labeled, directed control-flow graph  $G = (N, E, n_o)$  consisting of the finite set  $N$  of the statements of the program and the set of edges  $E$  which represent the control flow between statements.

- $N$  is partitioned into two subsets:
    - $N^S$  – *statement nodes* with at most one successor and
    - $N^P$  – *branch nodes* with at least two *distinct* successors
  - $N^E \subseteq N^S$  denotes the nodes in  $N^S$  that have no successors (the exit nodes)
  - the start node  $n_0$  has no incoming edges and all nodes in  $N$  are reachable.
  - if  $N^E$  contains only one element and this element is reachable from all other nodes of  $G$  then  $G$  satisfies the *unique end node property*
-

# CFG - Nodes

---

```
public static long sumOfInts(int upTo) {  
    long s = 1l;           // stmt node (assign)  
    int i = 2;             // stmt node (assign)  
    while (i < upTo) {     // branch node (if)  
        s += i;            // stmt node (add followed by assign)  
        i++;              // stmt node (inc)  
    } // goto loop start - stmt node; control-transfer instruction  
    return s;              // stmt node  
}
```

---

# CFGs - Paths

---

A CFG path  $\pi$  from  $n_i$  to  $n_k$  is a sequence of nodes  $n_i, n_{i+1}, \dots, n_k$  such that for each consecutive pair  $(n_i, n_j)$  in the path an edge from  $n_i$  to  $n_j$  exists.

A path is *nontrivial* if it contains at least two nodes.

A path is *maximal* if it is infinite or terminates in an end node.

---

# Dominance

---

A Node  $n$  dominates node  $m$  in  $G$  ( $\text{dom}(n,m)$ ) if every path from the start node  $n_0$  to  $m$  passes through  $n$  (hence,  $\text{dom}$  is reflexive).

Please recall that the classical definitions of post-dominance can often not reliably used to Java bytecode.

---



# Dominance - Example

---

```
public static int abs(int i) {  
    if(i == Integer.MIN_VALUE)  
        throw new ArithmeticException();  
    if(i < 0)  
        i = -i;  
    return i;  
}
```

Here, e.g., the `if` statements dominate the return instruction, but `i = -i` or `throw new ...` doesn't. Additionally, the `throw new ...` statement is also dominated by the first `if` (line 2).

---

# (Ir)Reducible CFGs

---

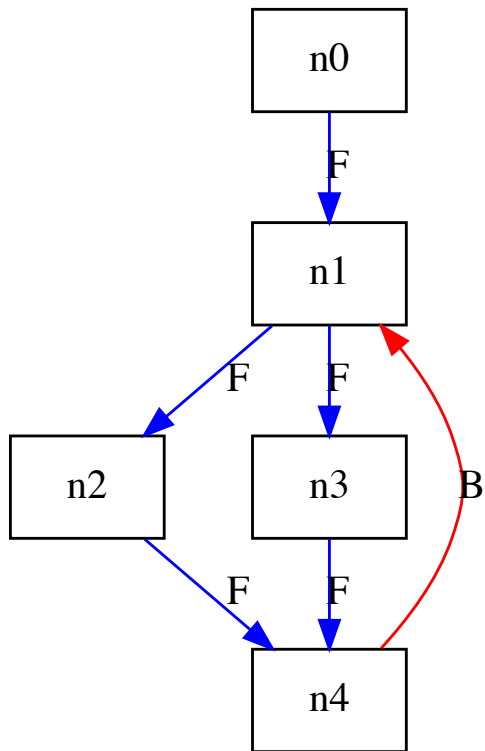
A CFG  $G = (N, E, n_0)$  is **reducible** if ...

- $E$  can be partitioned into disjoint sets  $E_f$  (forward edges) and  $E_b$  (backward edges)
  - such that  $(N, E_f)$  forms a directed-acyclic graph (DAG) in which **each** node can be reached from the start node  $n_0$
  - and for all edges  $e \in E_b$ , the target of  $e$  dominates the source of  $e$ .
-

# Reducible CFGs - Example

---

B = Backward Edge; F = Forward Edge



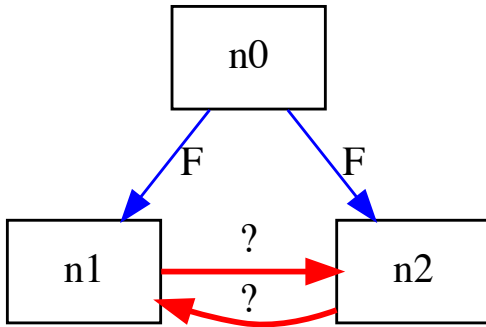
It is (trivially) possible to partition the edges in forward and backward edges.

---

# Irreducible CFGs - Example

---

B = Backward Edge; F = Forward Edge



The edges (n0, n1) and (n0, n2) are forward edges. Either (n1, n2) or (n2, n1) can be considered a forward edge, but not both at the same time. However, neither of these edges is a backward edge. Hence, it is not possible to partition the edges such that  $E = E_f \cup E_b$

---

# CFGs based on *Basic Blocks*

---

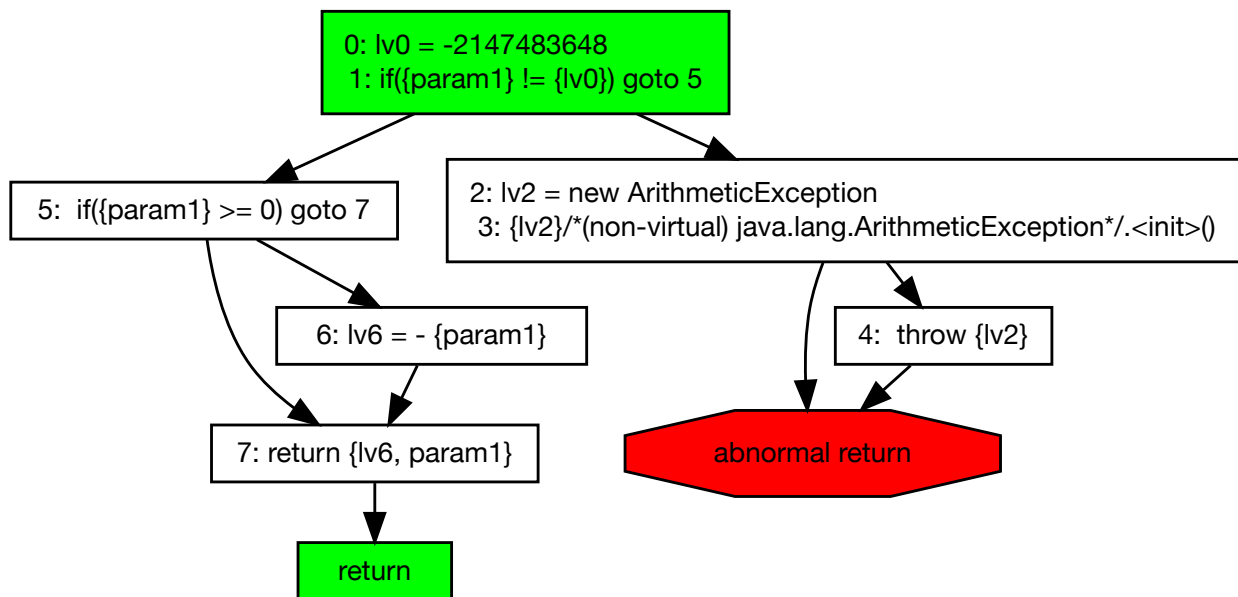
In real-world CFGs the nodes are typically based on using basic blocks.

- A basic block is a maximal-length sequence of statements without jumps in and out (and no exceptions are thrown by intermediate instructions).
  - A basic blocks based CFG (still) represents the control flow of a single method.
-

# CFGs based on *Basic Blocks* - Example

In the following, we'll see the control-flow graph for the given method based on the three-address code representation of the method.

```
public static int abs(int i) {  
    if(i == Integer.MIN_VALUE)  
        throw new ArithmeticException();  
    if(i < 0)  
        i = -i;  
    return i;  
}
```



Please note, that the basic blocks are rather small. This is a typical property of real world code. Furthermore, the call of the constructor may throw an arbitrary exception and therefore also lead to an abnormal return.

The following code snippet – which can be executed using the sbt console – demonstrates how to get the CFG for a specific method/class using the OPAL framework:

```
val p = org.opalj.br.analyses.Project(new java.io.File("APSA/Lecture/2019/2-Java-Bytecode/ControlFlow.class"))  
val tacKey = p.get(org.opalj.tac.ComputeTACAIKey)  
val taCode = tacKey(p.classFile(org.opalj.br.ObjectType("ControlFlow")).get.findMethod("abs").head)  
println(org.opalj.tac.tacToDot(taCode.stmts, taCode.cfg))
```

# References

---

1. Pradel, M., & Sen, K.; DeepBugs: a learning approach to name-based bug detection. Proceedings of the ACM - OOPSLA, 2018, [doi↗](#)
2. Sven Amann, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezin; A Systematic Evaluation of Static API-Misuse Detectors; IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 2018 [↗](#)
3. Eichberg, M., Hermann, B., Mezini, M., & Glanz, L.; Hidden truths in dead software paths; ACM, 2015, [doi↗](#)
4. Krüger, S., Nadi, S., Reif, M., Ali, K., Mezini, M., Bodden, E., et al.; CogniCrypt: supporting developers in using cryptography; IEEE Press., 2017 [↗](#)
5. Bessey, A., Block, K., Chelf, B.; A few billion lines of code later: using static analysis to find bugs in the real world; Communications of the ACM, 2010, vol. 53, No. 2 [↗](#)
6. Livshits, B., Sridharan, M., Smaragdakis, Y., Amaral, J. N., Møller, A., Lhoták, O., et al.; In Defense of Soundness: A Manifesto; Communications of the ACM, 2015 , 58(2). [↗](#)
7. A. Aho, R. Sethi and J. D. Ullman; Compilers - Principles, Techniques and Tools; Addison Wesley 1988 [↗](#)
8. Ranganath, V. P., Amtoft, T., Banerjee, A., Hatcliff, J., & Dwyer, M. B.; A new foundation for control dependence and slicing for modern program structures. ACM Trans. Program. Lang. Syst., 29(5), 2007, [doi↗](#)