HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Static Analysis for C++ with Phasar

Block 2 & 3

Philipp Schubert
philipp.schubert@upb.de

Ben Hermann
ben.hermann@upb.de

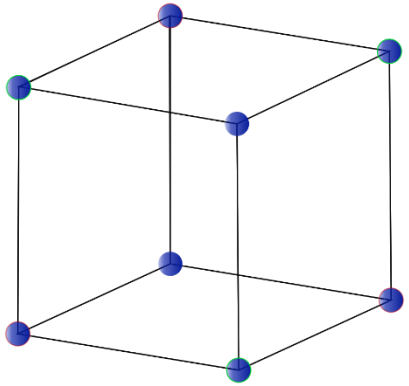
Eric Bodden
eric.bodden@upb.de

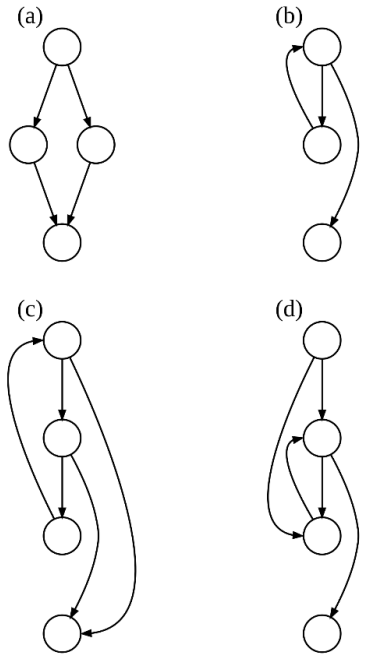
SOFTWARE
TECHNIK

In this Block

1. **Analyses as Plug-ins**
2. **Hello, World!**
3. **Taint Analysis and IFDS**
4. **Writing an Analysis**
5. **Run/ Test/ Debug an Analysis**
6. **Advanced Techniques**

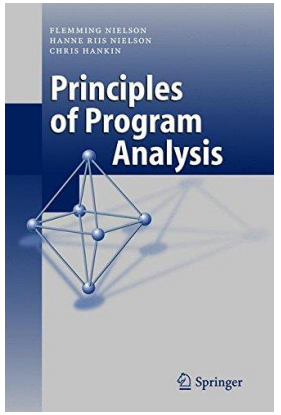
Analyses as plug-ins

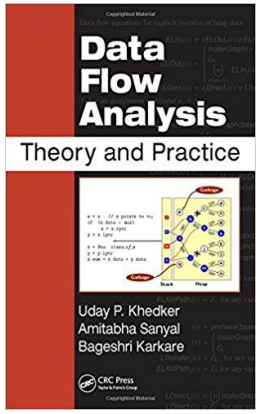




- What can you plug-in?
  - Data-flow analyses
    - IFDS/ IDE
    - Monotone framework (intra/ inter)
  - Control-flow / call graph analyses
    - Control flow graph
    - Inter-procedural control flow graph

- How to implement analysis plug-ins in C/C++?
  - Shared object libraries
    - `dlopen()`, `dlsym()`, `dlclose()`





Analyses as plug-ins

- Simplified plugin interface

```
class Problem {
public:
    Problem() = default;
    virtual ~Problem = default();
    virtual void print() = 0;
};
extern "C" unique_ptr<Problem>
createProblem();
```

- Virtual defaulted destructors?

```
Animal *A = new Lion;
delete A;
```

- A plugin implementation

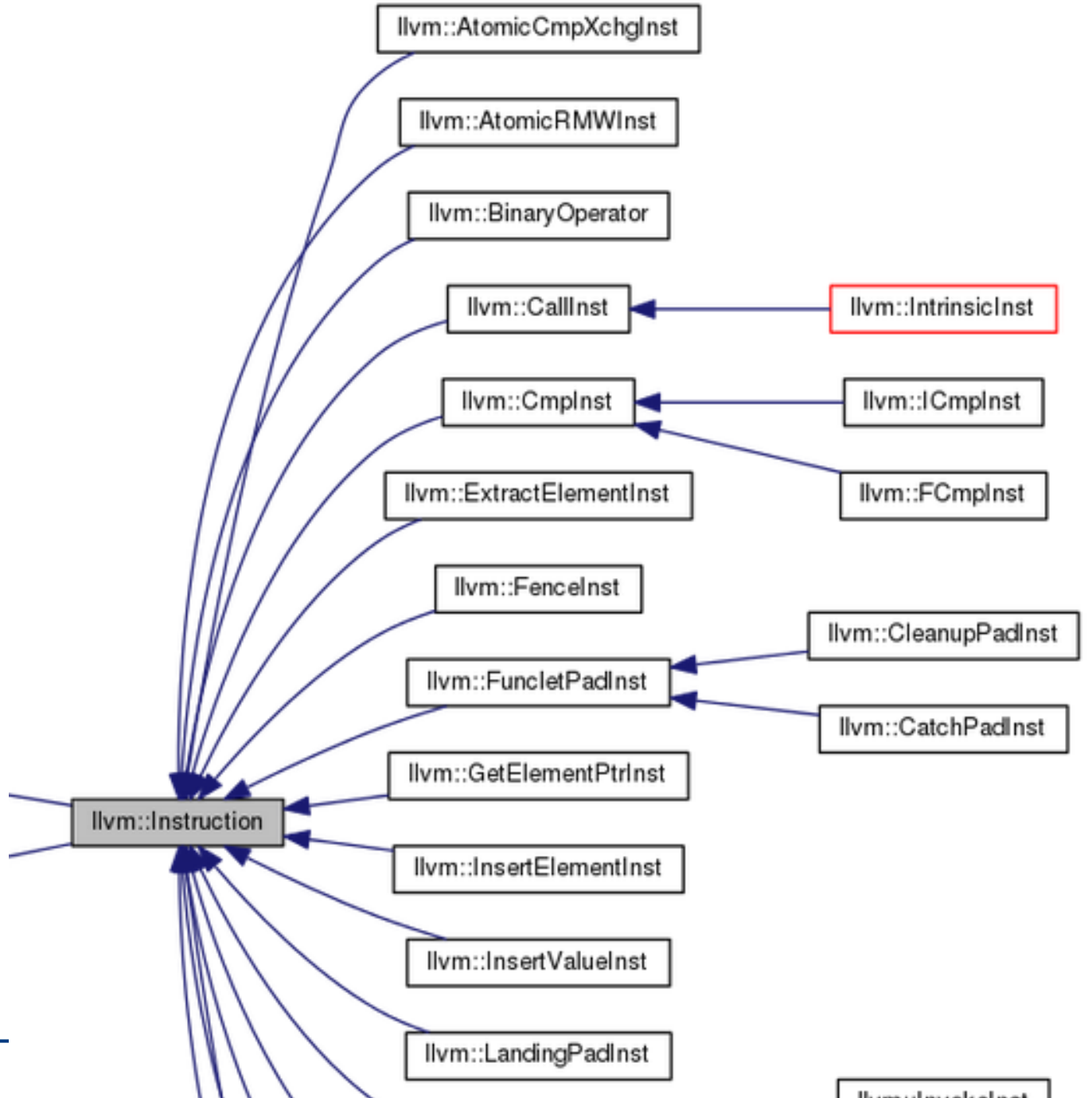
```
class MyProblem : public Problem {
public:
    virtual ~MyProblem() = default();
    virtual void print() override;
};
MyProblem::print() {
    cout << "plug-in\n";
}
unique_ptr<Problem> createProblem() {
    return make_unique<MyProblem>();
}
```

Self-registering plug-ins

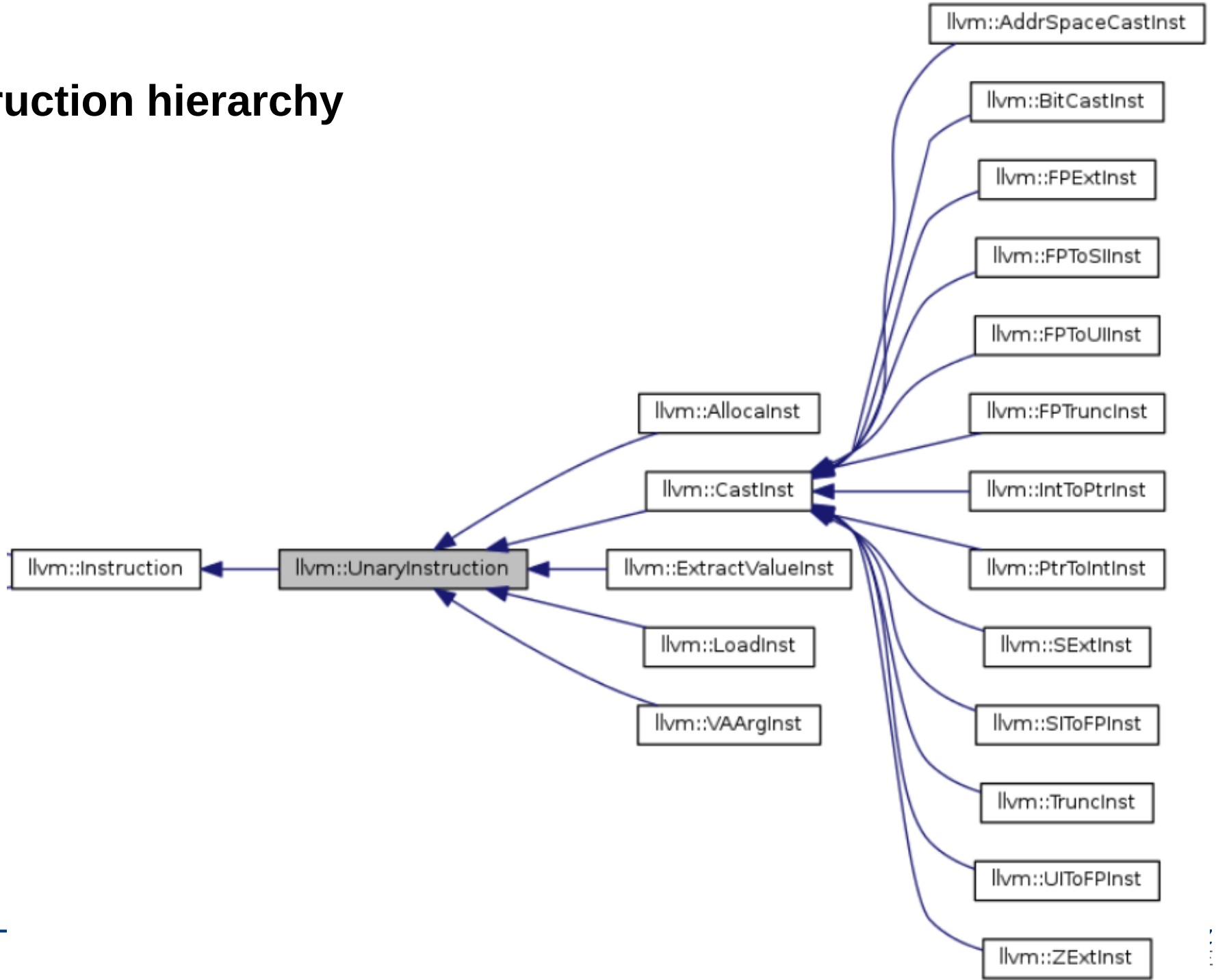
- Provide two special functions annotated with
  - `__attribute__((constructor))`
    - Is called on load
  - `__attribute__((destructor))`
    - Is called on unload
- Use to register to a factory map in constructor annotated function
  - `std::map<std::string, std::unique_ptr<Problem> (*)()> RegisteredPlugins;`
- Each analysis corresponds to a shared object library → plug-in
- Phasar can detect what is registered
- May be executed on demand

Writing a static analysis

LLVM instruction hierarchy



LLVM instruction hierarchy



Important LLVM instructions

- LLVM IR instructions
  - **`AllocaInst`**: allocate stack memory
  - **`LoadInst`**: load from memory location
  - **`StoreInst`**: store to memory location
  - **`CallInst`**: call function
  - **`InvokeInst`**: call function that might throw
  - **`ReturnInst`**: return (value) from function
- Instructions form a hierarchy
  - Use LLVM´s custom RTTI
    - **`llvm::isa<instType>(inst *)`**
    - **`llvm::dyn_cast<instType>(inst *)`**
    - This is cheap! ➔ One lookup

```
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i8*
  %4 = alloca i32
  store i32 0, i32* %1, align 4
  store i32 42, i32* %2, align 4
  call void @_Z3foov()
  invoke void @_Z3barv()
          to label %5 unwind label %6

; <label>:5:
; preds = %0
  br label %13

; <label>:6:
; preds = %0
  %7 = landingpad { i8*, i32 }
          catch i8* null
  ...
```

Some handy functionality

- `User` is a common base class for nodes refereeing to values → provides use-def chain
  - Instructions provide
    - `User::operands()`
    - `User::getNumOperands()`
    - `User::getOperand()`
- `Value::users()`
  - Iterate def-use information
  - Who uses this value?
- `Value::uses()`
  - Iterate use-def information
  - What is used by this value?

```
Instruction *Inst = /* get instruction */

for (auto &Operand : Inst->operands()) {
   Operand->print(outs());
}

Function *F = /* get function */
for (auto &user : F->users()) {
   user->print(outs());
}
```

Accessing memory: LLVM IR is register based

- Values are preferable stored in registers
  - First class types → just use them
  - Structure types use
    - **`extractvalue`**
    - **`insertvalue`**
- To access values in memory use load and store
  - **`load`**
  - **`store`**
- Vector types (platform independent SIMD)
  - **`extractelement`**
  - **`insertelement`**

Hello, World! in LLVM

- Code can be found in `llvm-hello_world/`
- LLVM documentation `llvm_doxygen-5.0.0.tar.xz`
  - `$ tar xvf llvm_doxygen-5.0.0.tar.xz`

```
int main(int argc, char **argv) {
  if (argc != 2) {
    cout << "usage: <prog> <IR file>\n";
    return 1;
  }
  // parse a llvm module from an IR file
  llvm::SMDiagnostic Diag;
  unique_ptr<llvm::LLVMContext>
          C(new llvm::LLVMContext);
  unique_ptr<llvm::Module>
          M = llvm::parseIRFile(argv[1],
                                Diag,
                                *C);
  // check if the module is alright
  bool broken_debug_info = false;
  if (llvm::verifyModule(*M, &llvm::errs(),
                                &broken_debug_info)) {
    llvm::errs() << "error\n";
    return 1;
  }
  if (broken_debug_info) {
    llvm::errs() << "caution\n";
  }
  auto F = M->getFunction("main");
  if (!F) {
    llvm::errs() << "error\n";
    return 1;
  }
  llvm::outs() << F->getName() << '\n';
  for (auto &BB : *F) {
    for (auto &I : BB) {
      I.print(llvm::outs());
      llvm::outs() << '\n';
    }
  }
  llvm::llvm_shutdown();
  return 0;
}
```

Target code

- Target code can be found in `target/assignment_2.c(.ll)`
- First example
  - C

```
int taint() { return -13; }

void print(int i) { /* sink */ }

int main() {
    int i = 0;
    int j = taint();
    print(i);
    print(j);
    return 0;
}
```

  - LLVM IR

```
define i32 @taint() {
  ret i32 -13
}
define void @print(i32) {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ret void
}
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %4 = call i32 @taint()
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  call void @print(i32 %5)
  %6 = load i32, i32* %3, align 4
  call void @print(i32 %6)
  ret i32 0
}
```

`llvm::outs() << "Hello, World!\n";`

- Use the Hello, World! template and run it on some IR files
- Observe the code comments
- Inspect the instructions
  - Use some **`llvm::isa<>()`** to check for specialized instructions
  - Use **`llvm::dyn_cast<>()`** to perform some dynamic casts
  - Inspect their operands
  - Create an **`llvm::ImmutableCallSite`** variable wrapping a **`llvm::CallInst`**

Taint Analysis

- Tracking data-flows through a program
- Used for numerous applications
  - Detect SQL injection
  - Detect XSS
  - Find data leaks
  - …

Taint Analysis

- Source(s)
  - Function(s) or operation(s) of interest
  - Generate tainted values
- Tainted values are tracked through the target program
- Sink(s)
  - Function(s) or operation(s) of interest
  - Receive data
  - Report warning if tainted value is received
    - Leak!

Taint Analysis

- Malicious input: `'OR TRUE; SHOW TABLES LIKE '%`

```
void executeSQL(string q) {
  Driver *d = get_driver_instance();
  unique_ptr<Connection> conn(
      d->connect(/* credentials */));
  conn->setSchema(/* db to be used */);
  unique_ptr<Statement> stmt(conn->createStatement());
  unique_ptr<ResultSet> res = stmt->execute(q);        // Statement::execute is sink
  res->beforeFirst();
  while (res->next()) {
    cout << "Student: " << res.getString(1) << '\n';
  }
}
int main(int argc, char **argv) {
  string input = argv[1];                              // argv is source
  string query =
    "SELECT Students.Name FROM Students
     WHERE Students.id='" + input + "'";
  executeSQL(query);
}
```

IFDS or inter-procedural finite distributive subset problems

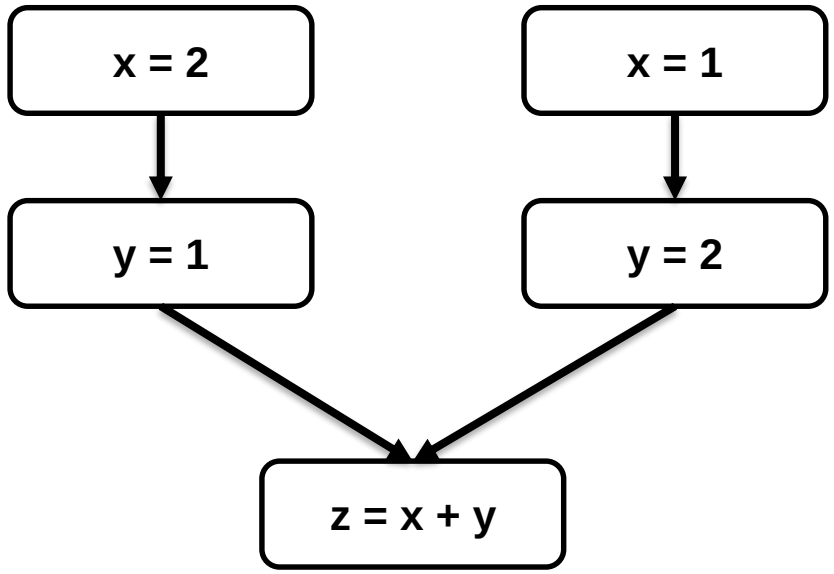
- Map data-flow problem to graph reachability problem
- Reachability is tested on “exploded super-graph” (ESG)
  - super-graph ⇔ inter-procedural control-flow graph (ICFG)
  - Annotated ICFG
  - Data-flow fact ⇔ node within ESG
- If a node (s_i, d_j) is reachable in ESG
  - ➔ data-flow fact d_j holds at statement s
  - Starting point is Λ
- How to build the ESG?
  - Flow functions
- Complexity $O(|E| \cdot |D|^3)$

Pros & cons of IFDS

- Pros
  - Efficient
    - Reusable summaries
  - Fully context-sensitive
    - (unlike call-string approach with k-sensitivity)

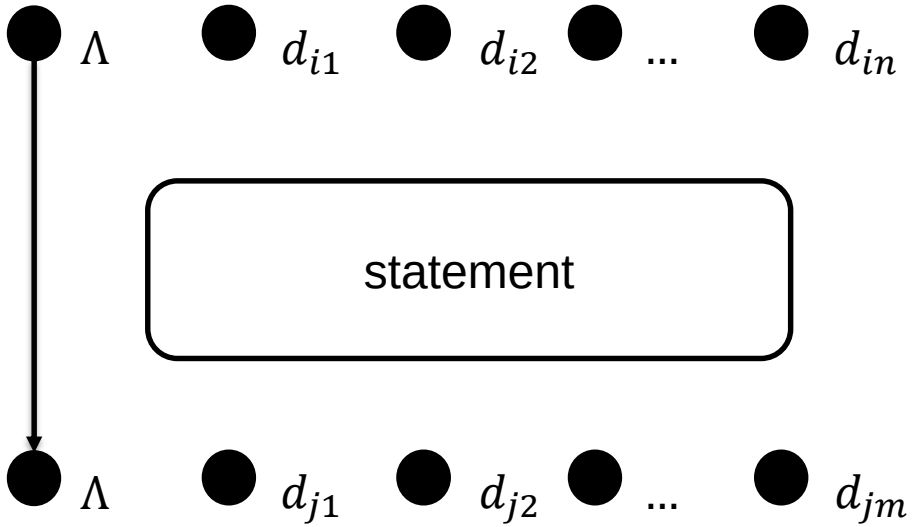
- Cons
  - Set union as merge operator
  - Distributive flow functions
  - No ad-hoc solution
    - A framework is needed

- If f is monotonic, we have
  - $f(x) \sqcap f(y) \sqsubseteq f(x \sqcap y)$
- If f is distributive, we have
  - $f(x) \sqcap f(y) = f(x \sqcap y)$



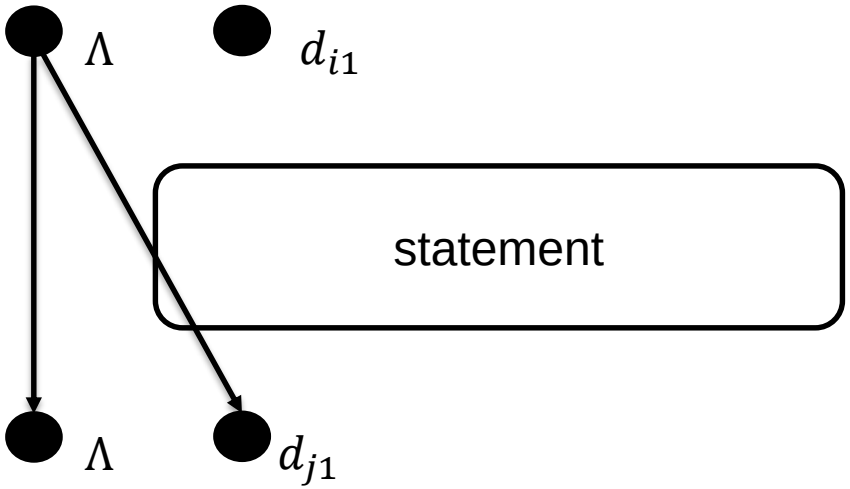
IFDS or inter-procedural finite distributive subset problems

- Separate data-flow facts



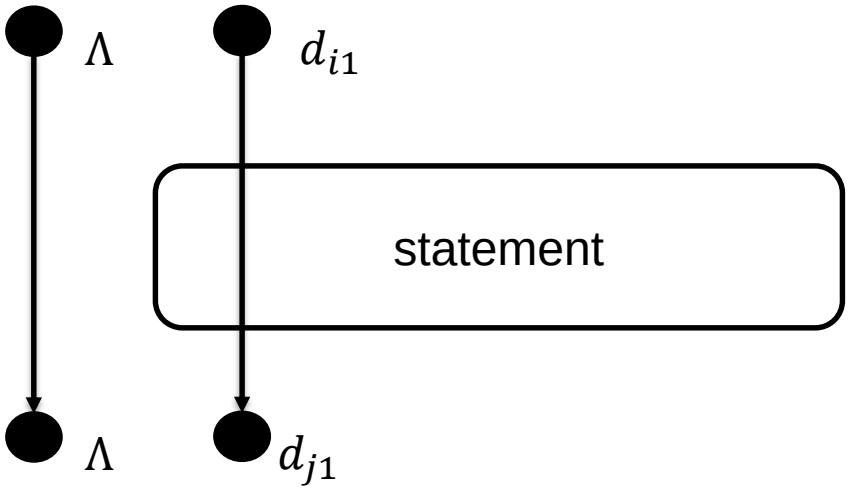
IFDS or inter-procedural finite distributive subset problems

- Map functions to graphs
  - Gen



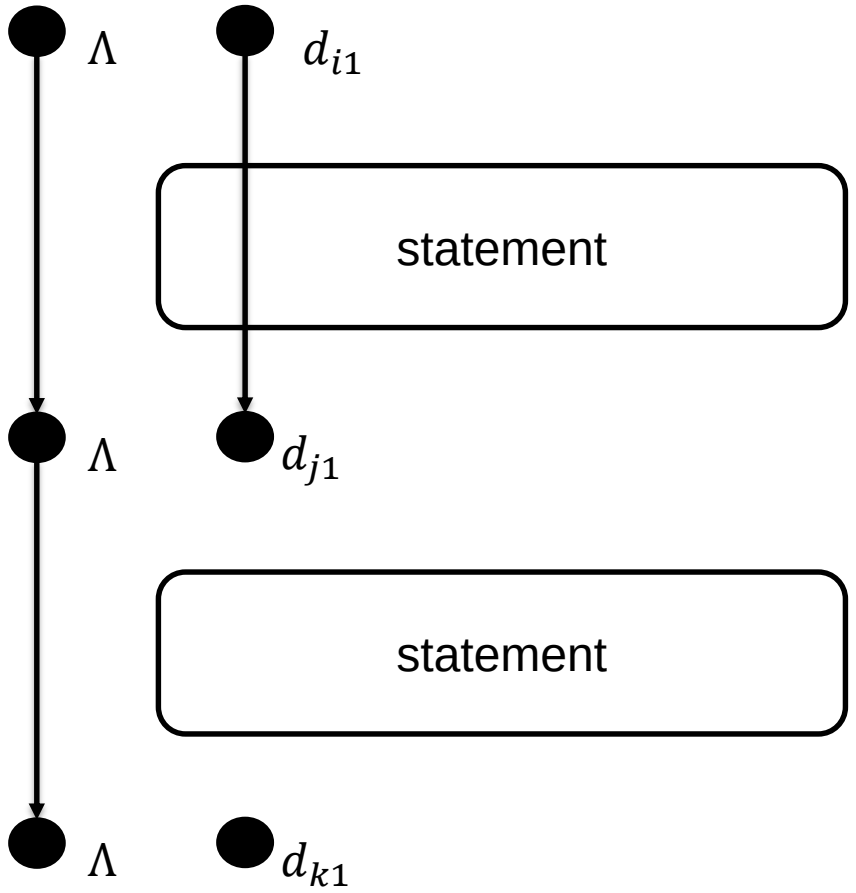
IFDS or inter-procedural finite distributive subset problems

- Map functions to graphs
  - ID



IFDS or inter-procedural finite distributive subset problems

- Map functions to graphs
  - Kill

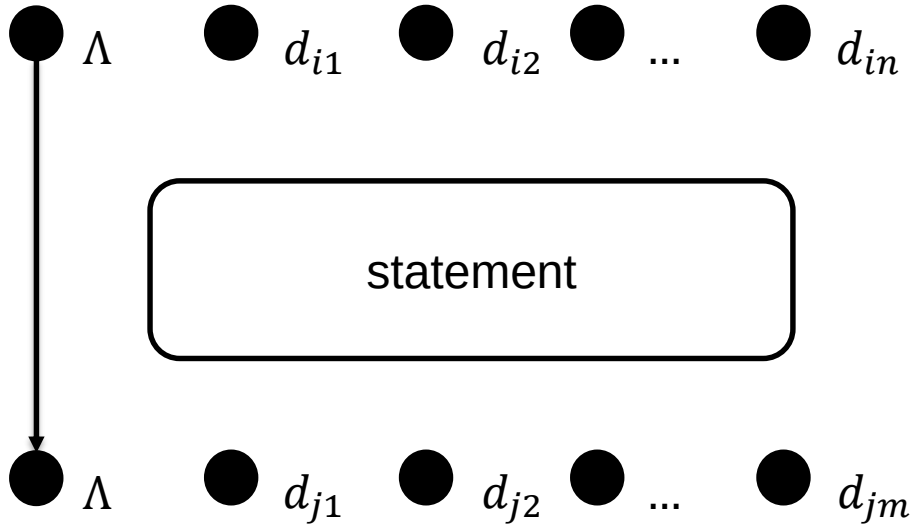


IFDS or inter-procedural finite distributive subset problems

- ESG is build according to flow functions
- Flow functions describe the effect of a statement
- Inter-procedural analysis
  - `getNormalFlowFunction()`
  - `getCallFlowFunction()`
  - `getCallToRetFlowFunction()`
  - `getReturnFlowFunction()`
  - `( getSummaryFlowFunction() )`

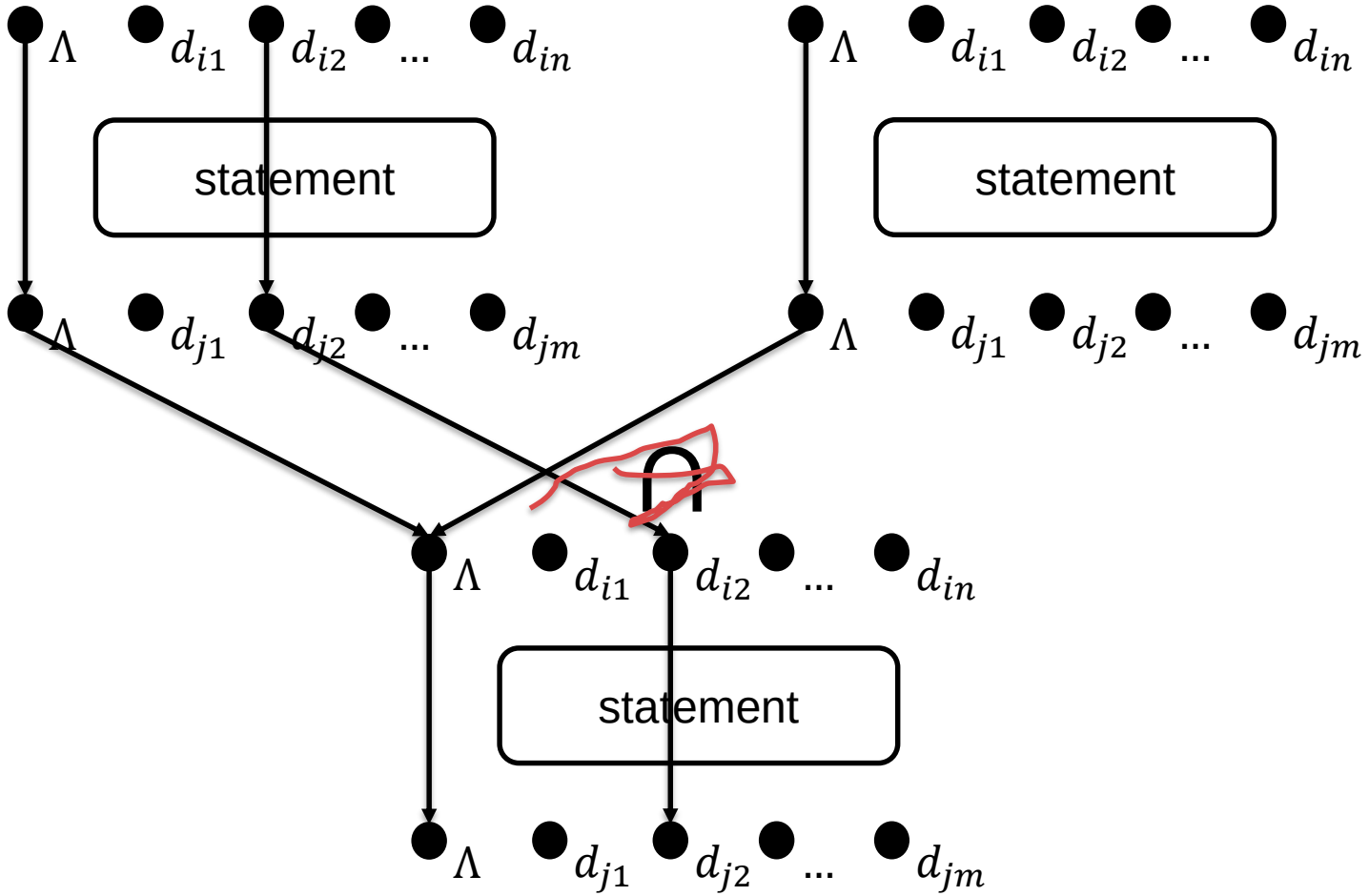
IFDS or inter-procedural finite distributive subset problems

- Flow functions describe the effect of a statement
- Inspect the statement and specify the effect
  - This is your task:
    - What type of statement?
      - Assignment
      - Arithmetic
      - …
    - What facts are generated, killed or id-ed?



IFDS or inter-procedural finite distributive subset problems

- Merge operator is set union
- Set intersect cannot be encoded

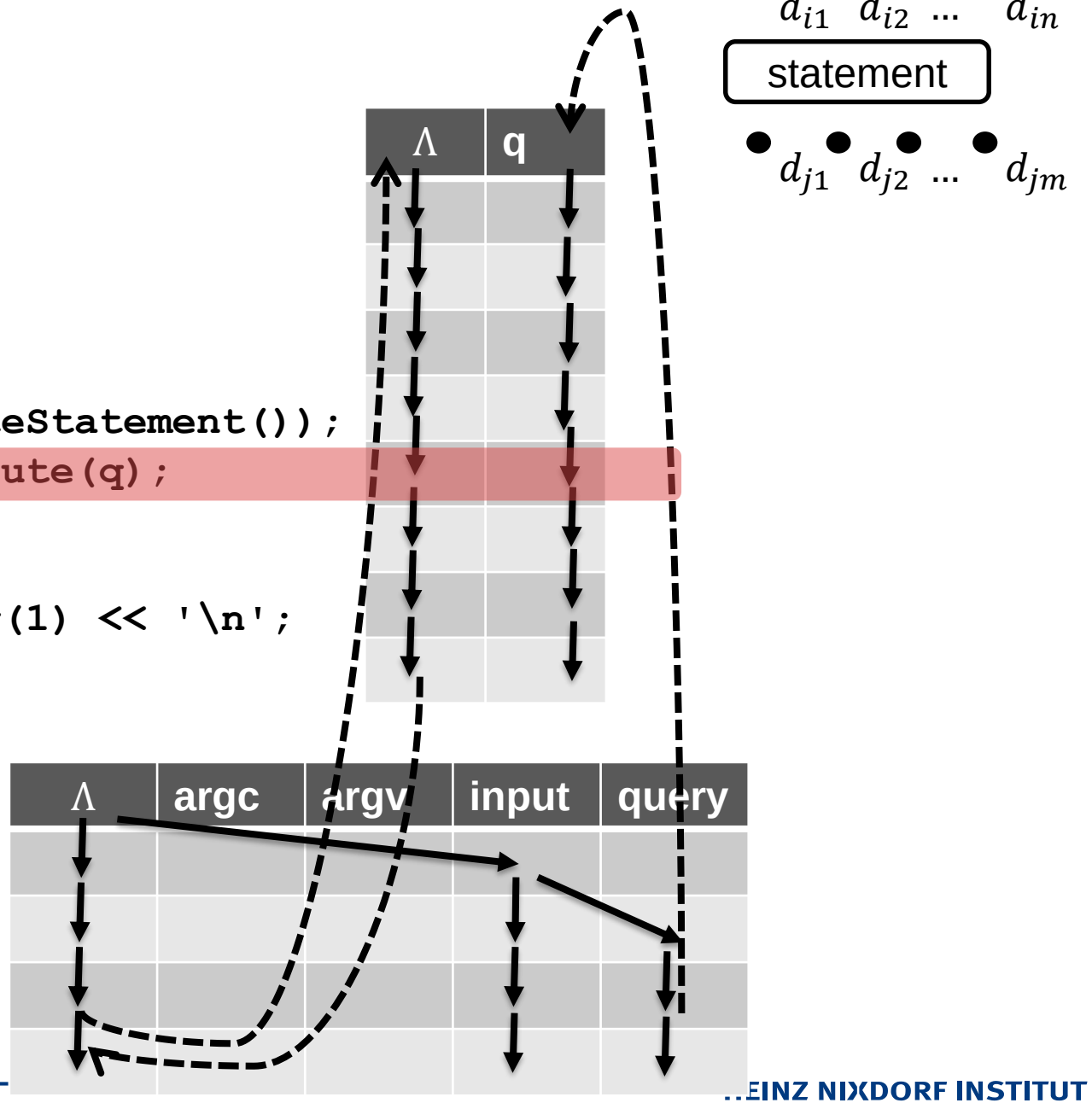


IFDS or inter-procedural finite distributive subset problems

- E.g. in our taint analysis
  - `string input = argv[1];`
    - Flow function would generate an edge from Λ to `input`
  - `sanitize(string &s);`
    - Flow function would not generate an outgoing edge from `s`
  - `Statement::execute(const SQLString &sql);`
    - If `sql` is reachable at this point ➔ report leak!

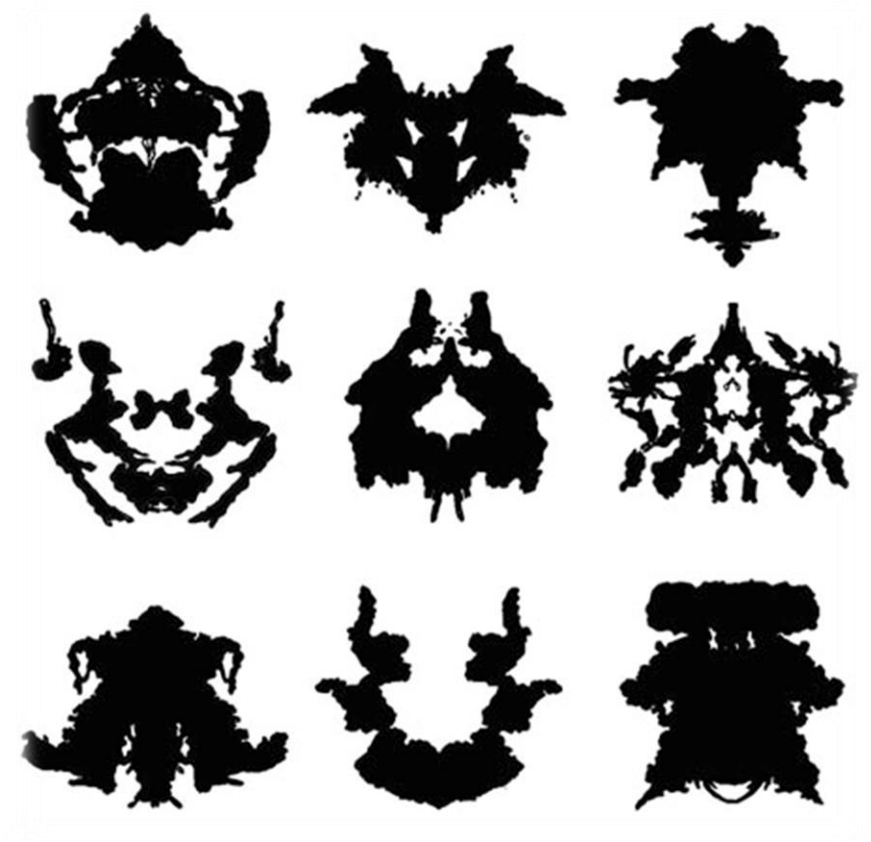
Taint Analysis

```
void executeSQL(string q) {
  Driver *d = get_driver_instance();
  unique_ptr<Connection> conn(
      d->connect(/* credentials */));
  conn->setSchema(/* db to be used */);
  unique_ptr<Statement> stmt(conn->createStatement());
  unique_ptr<ResultSet> res = stmt->execute(q);
  res->beforeFirst();
  while (res->next()) {
    cout << "Student: " << res.getString(1) << '\n';
  }
}
int main(int argc, char **argv) {
  string input = argv[1];
  string query =
    "SELECT Students.Name FROM Students
     WHERE Students.id='" + input + "'";
  executeSQL(query);
}
```



Analysis pattern

- Many parts of an analysis follow certain pattern
- Pattern describe how to accomplish things
  - Fact generation within flow functions
  - Parameter mapping
  - Modeling effects of a function
  - …
- These pattern will be mentioned as we go along
- Learning a pattern makes analysis development easier



Encode flow functions in C++

- Describing flow functions in C++
  - Intra-procedural flow

```
virtual FlowFunction<D> *getNormalFlowFunction(N curr, N succ) = 0;
```

    - This must be implemented

  - Definition of `FlowFunction`

```
template <typename D> class FlowFunction {
public:
    virtual ~FlowFunction() = default;
    virtual set<D> computeTargets(D source) = 0;
};
```

Writing a simple flow function

- Example: collect all stack allocations

```
virtual FlowFunction<D>
*getNormalFlowFunction(N curr, N succ) override {
   if (auto Alloca = dyn_cast<AllocaInst>(curr)) {
      struct MyFlowFunction : FlowFunction {
         set<D> computeTargets(D src) {
            return (src == zerovalue) ?
                   { src, Alloca } : { src };
         }
      };
      return new MyFlowFunction;
   }
   return Identity::getInstance();
}
```

- Target code

```
int main() {
    int i = 42;
    int j = 13;
    return 0;
}
```

➔ Translate to IR

```
define dso_local i32 @main() #0 {
   %1 = alloca i32, align 4
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 0, i32* %1, align 4
   store i32 42, i32* %2, align 4
   store i32 13, i32* %3, align 4
   ret i32 0
}
```

Flow functions: shortcuts

- Gen shortcut
  - Generates a value unconditionally

```
template<typename D> class Gen : public FlowFunction<D> {
private:
    D genValue;
    D zeroValue;
public:
    Gen(D genValue, D zeroValue) : genValue(genValue), zeroValue(zeroValue) {}
    virtual ~Gen() = default;
    set<D> computeTargets(D src) override {
        if (src == zerovalue) {
            return { src, genValue };
        }
        return { src };
    }
};
```

The identity flow function

- Identity shortcut

```
template<typename D> class Identity : public FlowFunction<D> {
private:
    Identity() = default;

public:
    virtual ~Identity() = default;
    Identity(const Identity &) = delete;
    Identity(Identity &&) = delete;
    Identity &operator=(const Identity &) = delete;
    Identity &operator=(Identity &&) = delete;
    set<D> computeTargets(D src) override { return { src }; }
    static *Identity getInstance() {
       static *Identity id = new Identity;
       return id;
    }
};
```

Flow function using short cuts

- Example: collect all stack allocations
  - Using shortcuts

```
virtual FlowFunction<D>
*getNormalFlowFunction(N curr, N succ) override {
   if (auto Alloca = dyn_cast<AllocaInst>(curr) {
      return new Gen<D>(curr, zeroValue);
   }
   return Identity::getInstance();
}
```

- Target code

```
int main() {
    int i = 42;
    int j = 13;
    return 0;
}
```

➔ Translate to IR

```
define dso_local i32 @main() #0 {
   %1 = alloca i32, align 4
   %2 = alloca i32, align 4
   %3 = alloca i32, align 4
   store i32 0, i32* %1, align 4
   store i32 42, i32* %2, align 4
   store i32 13, i32* %3, align 4
   ret i32 0
}
```

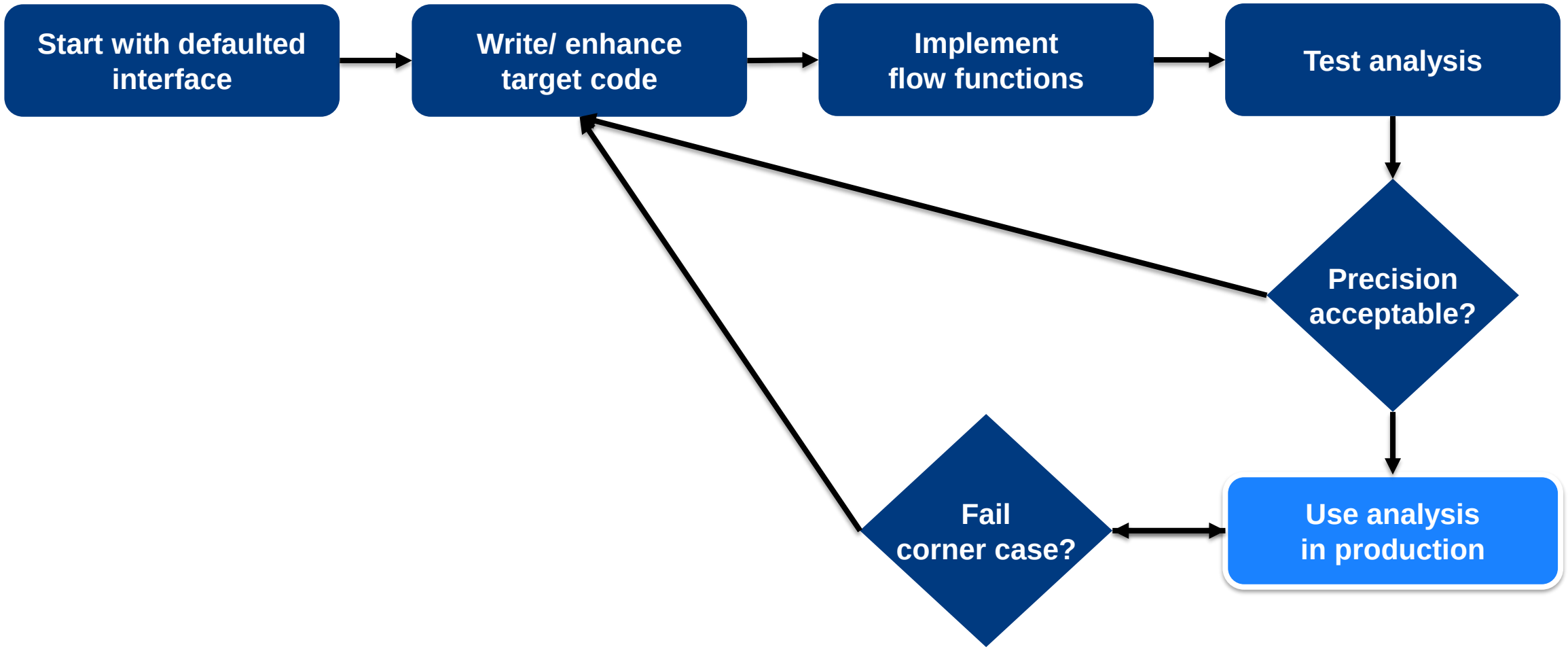
IFDS or inter-procedural finite distributive subset problems

- Model intra-procedural flows
  - `getNormalFlowFunction()`
- Model parameter mapping (actuals to formals)
  - `getCallFlowFunction()`
- Model flows along a call-site
  - `getCallToRetFlowFunction()`
- Model return value(s) (caution pointer/ reference parameters)
  - `getReturnFlowFunction()`
- Model special semantics (later on)
  - `getSummaryFlowFunction()`

Write your own simple taint analysis

- Track tainted values through a program
- Find undesired data-flows
- Here C programs (absence of name mangling)
  - Source
    - `taint()`
  - Sinks
    - `print()`
    - `log()`

Writing an analysis: general approach



Debugging cycle

- Currently no automated code generator, yet
- Pick an existing template and adjust it to your needs
  - Default all interface functions
  - What is to be found?
  - Write some small target code (containing issues)
  - Specify the preliminary flow functions

Analysis template

- **`MyIFDSProblem`** can be found in **`plugins/`**
- **`MyIFDSProblem.h`** and **`MyIFDSProblem.cxx`** are already set-up
- Provide implementations for the currently defaulted flow function factories
  - **`getNormalFlowFunction()`**
  - **`getCallFlowFunction()`**
  - **`getCallToRetFlowFunction()`**
  - **`getReturnFlowFunction()`**
- Use **`$ make`** to compile your analysis
- Run your analysis using **`$ phasar --config <conf>`**
  - where ***`<conf>`***
  - **`../target/*.conf`**
  - e.g.: **`$ phasar --config ../target/assignment_2.c.ll.conf`**

Target code

- Target code can be found in **`target/`**
  - Use configuration files
- First example
  - C

```
int taint() { return -13; }

void print(int i) { /* sink */ }

int main() {
    int i = 0;
    int j = taint();
    print(i);
    print(j);
    return 0;
}
```

- LLVM IR

```
define i32 @taint() {
  ret i32 -13
}
define void @print(i32) {
  %2 = alloca i32, align 4
  store i32 %0, i32* %2, align 4
  ret void
}
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %4 = call i32 @taint()
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  call void @print(i32 %5)
  %6 = load i32, i32* %3, align 4
  call void @print(i32 %6)
  ret i32 0
}
```

Dump of the analysis results

```
### DUMP LLVMIFDSSolver results
--- IFDS START RESULT RECORD ---
N:   store i32 0, i32* %2, align 4, !phasar.instruction.id !6, ID: 4 in function: main
D:   @zero_value = constant i2 0, align 4    V:  BOTTOM
--- IFDS START RESULT RECORD ---
N:   store i32 1, i32* %3, align 4, !phasar.instruction.id !7, ID: 5 in function: main
D:   @zero_value = constant i2 0, align 4    V:  BOTTOM
--- IFDS START RESULT RECORD ---
N:   %8 = load i32, i32* %3, align 4, !phasar.instruction.id !12, ID: 10 in function: main
D:   @zero_value = constant i2 0, align 4    V:  BOTTOM
--- IFDS START RESULT RECORD ---
N:   ret i32 0, !phasar.instruction.id !23, ID: 21 in function: main
D:   @zero_value = constant i2 0, align 4    V:  BOTTOM
```

Modelling the source

- Use this common pattern to model a function's effect
  - `getCallFlowFunction`
    - Avoid following called function(s)
    - Kill all facts when source is called
      - `KillAll`
  - `getCallToRetFlowFunction`
    - Plug-in the effects of source functions
    - Generate return value of `taint()`
    - Pass everything else as identity

```
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %4 = call i32 @taint()
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  call void @print(i32 %5)
  %6 = load i32, i32* %3, align 4
  call void @print(i32 %6)
  ret i32 0
}
```

Reads and writes

- `getNormalFlowFunction()`
- Loads and stores …
  - always have a pointer operand
    - The memory address
  - Store has a value operand
    - The thing to be stored
  - Storing a tainted value taints the memory
  - Loading from tainted memory leads to a tainted value

```
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %4 = call i32 @taint()
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  call void @print(i32 %5)
  %6 = load i32, i32* %3, align 4
  call void @print(i32 %6)
  ret i32 0
}
```

Modelling the sinks

- How to model the sinks?
  - Familiar pattern
  - Use `getCallFlowFunction()` and `getCallToRetFlowFunction()`
    - `KillAll`
    - Detect the leak!

```
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %4 = call i32 @taint()
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  call void @print(i32 %5)
  %6 = load i32, i32* %3, align 4
  call void @print(i32 %6)
  ret i32 0
}
```

Modelling a leak

- `getCallToRetFlowFunction()`
- What can be done?
  - Report leak and abort immediately
    - Usually not wise
  - Collect leak and report at the end
    - `map<const llvm::Instruction *, set<const llvm::Value *> leaks;`

```
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  store i32 0, i32* %2, align 4
  %4 = call i32 @taint()
  store i32 %4, i32* %3, align 4
  %5 = load i32, i32* %2, align 4
  call void @print(i32 %5)
  %6 = load i32, i32* %3, align 4
  call void @print(i32 %6)
  ret i32 0
}
```

Modelling binary operations

- What must be done when a tainted value is used in a binary operation?
  - Highly depends on your needs
    1. Tainted value in bin op leads to tainted result
    2. Tainted value in bin op leads to sanitized result
    3. Resulting taint depends on specific binary operation
  - How to detect the specific operation?
    - Downcast to **`llvm::BinaryOperation`**
    - **`getOpcodeName()`**
    - **`getOpcode()`**
      - Returns an enum that can be checked
  - Iterating the operands
    - **`for (auto &Operand : Inst->operands()) { /* */ }`**

Make it inter-procedural: Modelling function calls

- Calling a function `getCallFlowFunction()`
  - Map actual parameters into formals
  - Use common pattern
    - Collect actuals at call-site in a `std::vector`
    - Collect formals in callee in a `std::vector`
    - Generate facts holding at call-site

```
int taint() { return - 13; }
int foo(int i, int j) {
   return i;
}
int main() {
   int a = taint();
   int b = taint();
   int c = foo(a, b);
   return 0;
}
```

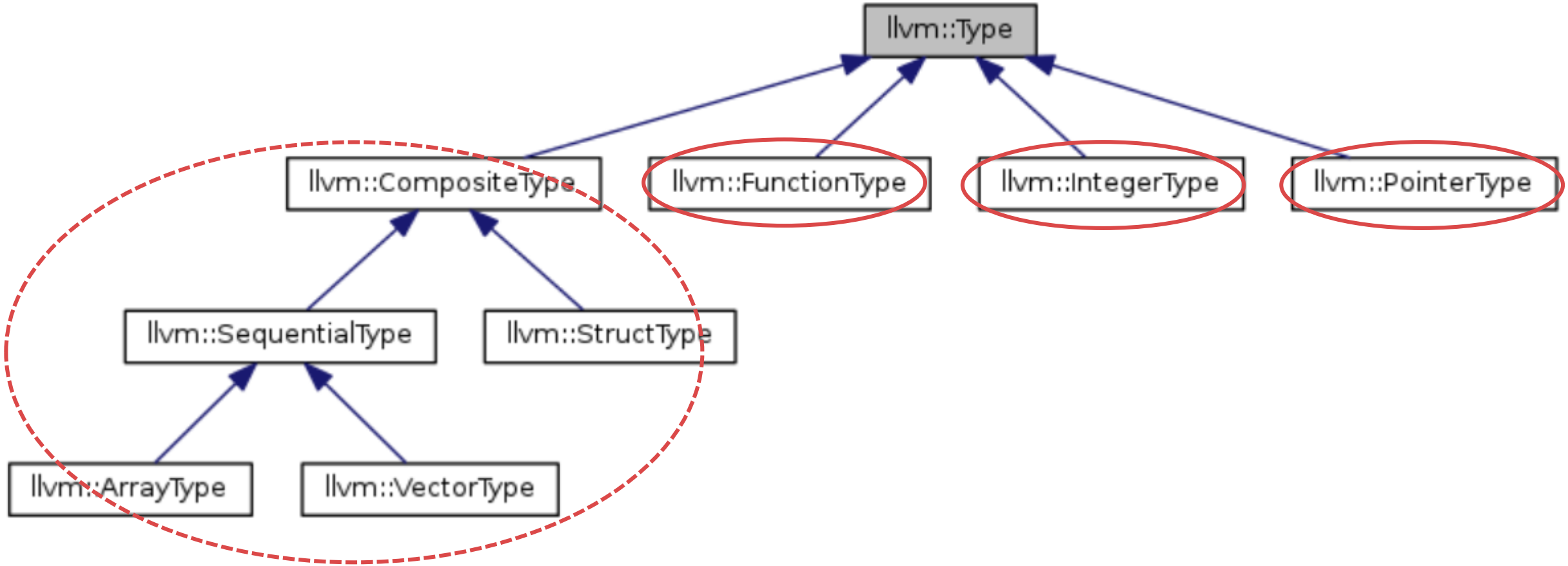
```
define i32 @foo(i32, i32) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  ret i32 %5
}
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %5 = call i32 @taint()
  store i32 %5, i32* %2, align 4
  %6 = call i32 @taint()
  store i32 %6, i32* %3, align 4
  %7 = load i32, i32* %2, align 4
  %8 = load i32, i32* %3, align 4
  %9 = call i32 @foo(i32 %7, i32 %8)
  store i32 %9, i32* %4, align 4
  ret i32 0
}
```

… and returns

- Map formals into actuals
- Pattern is almost the same
  - Pass by pointer/ reference
    - Collect actuals at call-site in a `std::vector`
    - Collect formals in callee in a `std::vector`
  - Special treatment: return value
  - Generate facts holding at function´s return at the return-site

```
define i32 @foo(i32, i32) {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  store i32 %1, i32* %4, align 4
  %5 = load i32, i32* %3, align 4
  ret i32 %5
}
define i32 @main() {
  %1 = alloca i32, align 4
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %5 = call i32 @taint()
  store i32 %5, i32* %2, align 4
  %6 = call i32 @taint()
  store i32 %6, i32* %3, align 4
  %7 = load i32, i32* %2, align 4
  %8 = load i32, i32* %3, align 4
  %9 = call i32 @foo(i32 %7, i32 %8)
  store i32 %9, i32* %4, align 4
  ret i32 0
}
```

Types in LLVM



C/C++ arrays and structs in LLVM

- Arrays and structs are similar
  - Memory object
  - + Offset
- Arrays are indexed with integer values
- Structs are indexed with their field names
  - Names are hidden offset sizes
- Both are indexed in the same manner
  - `llvm::GetElementPointerInst`

```
struct S {
    int x;
    int y;
};

typedef int[2] S;
```

Structs and fields

- How to?
  - `llvm::GetElementPointerInst`

```
struct S {
    int x;
    int y;
};

int main() {
   S s;
   s.x = 4;
   s.y = 5;
   int z = s.x + s.y;
   return 0;
}
```

```
%struct.S = type { i32, i32 }

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca %struct.S, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %4 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 0
  store i32 4, i32* %4, align 4
  %5 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 1
  store i32 5, i32* %5, align 4
  %6 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 0
  %7 = load i32, i32* %6, align 4
  %8 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 1
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %7, %9
  store i32 %10, i32* %3, align 4
  ret i32 0
}
```

Structs and fields

- GEP computes addresses
- First operand indexes through the pointer
- Second operand indexes the field

```
struct S {
    int x;
    int y;
};

S s;

int *x = &s[0].x;

*x = 5;
```

  - GEP does not access memory, it only computes a memory address

```
%struct.S = type { i32, i32 }

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca %struct.S, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %4 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 0
  store i32 4, i32* %4, align 4
  %5 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 1
  store i32 5, i32* %5, align 4
  %6 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 0
  %7 = load i32, i32* %6, align 4
  %8 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 1
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %7, %9
  store i32 %10, i32* %3, align 4
  ret i32 0
}
```

Modelling fields

- We need a field aware data-flow fact
- We need special treatment for loads and stores
- Distinguish by base variable and field / index
- `nullptr` denotes that `V` is a plain variable

```
class FieldAwareFact {
 private:
  llvm::Value *V;
  llvm::GetElementPtrInst *G = nullptr;

 public:
 FieldAwareFact(llvm::Value *V) : V(V) {}
 FieldAwareFact(
       llvm::Value *V,
       llvm::GetElementPtrInst *G)
              : V(V), G(G) {}
};
```

```
%struct.S = type { i32, i32 }

define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca %struct.S, align 4
  %3 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  %4 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 0
  store i32 4, i32* %4, align 4
  %5 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 1
  store i32 5, i32* %5, align 4
  %6 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 0
  %7 = load i32, i32* %6, align 4
  %8 = getelementptr inbounds %struct.S,
            %struct.S* %2, i32 0, i32 1
  %9 = load i32, i32* %8, align 4
  %10 = add nsw i32 %7, %9
  store i32 %10, i32* %3, align 4
  ret i32 0
}
```

Modelling fields

- Distinguish fields by index only works for static indices
  - For the following code we are doomed

```
struct S { int x, int y };

int main(int argc, char **argv) {
   if (argc >= 3) return 1;
   S s;
   int *ptr = static_cast<int*>(&s);
   int idx = argc - 1;
   ptr[idx] = 13;
   ptr[++idx] = 42;
   return s.x + s.y;
}
```

    - This is an odd way of indexing anyway

Modelling arrays

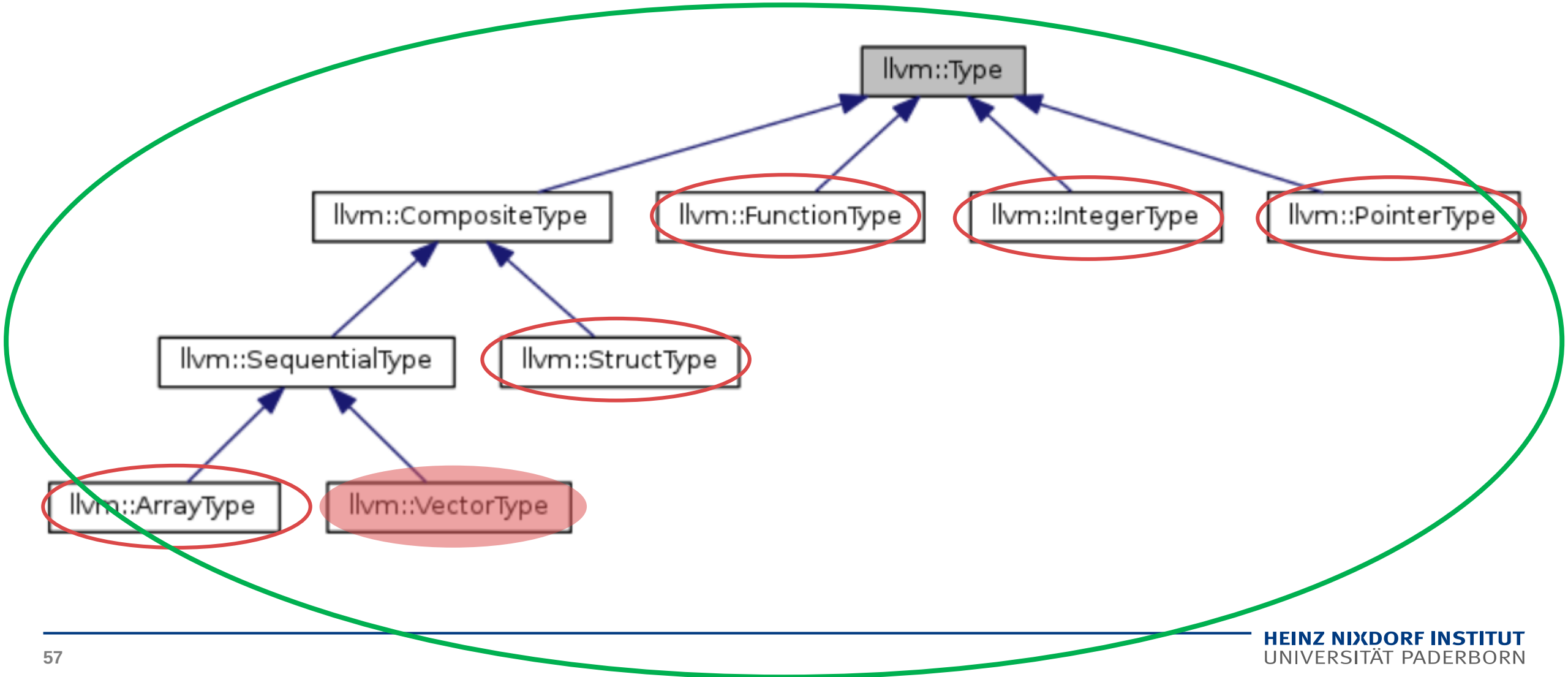
- Arrays: `[10 x i32]`
- Pretty much the same

```
int main() {
   int mem[10];
   mem[3] = 42;
   mem[4] = 13;
   return 0;
}
define i32 @main() #0 {
  %1 = alloca i32, align 4
  %2 = alloca [10 x i32], align 16
  store i32 0, i32* %1, align 4
  %3 = getelementptr inbounds [10 x i32], [10 x i32]* %2, i64 0, i64 3
  store i32 42, i32* %3, align 4
  %4 = getelementptr inbounds [10 x i32], [10 x i32]* %2, i64 0, i64 4
  store i32 13, i32* %4, align 16
  ret i32 0
}
```

Special treatment of loads and stores

```
if (auto Store = llvm::dyn_cast<llvm::StoreInst>(&I)) {
  // find where the memory was defined following the use-def chain
  for (auto &Use : Store->getPointerOperand()->uses()) {
    if (auto GEP = llvm::dyn_cast<llvm::GetElementPtrInst>(Use)) {
     // try to find a static index
     int Idx = -1;
     if (auto CI = llvm::dyn_cast<llvm::ConstantInt>(GEP->getOperand(2)))
       Idx = CI->getZExtValue();
     llvm::outs() << "index: " << Idx << '\n';      // print the index found
     GEP->getPointerOperand()->print(llvm::outs()); // print the base variable
     llvm::outs() << '\n';
     // check for array or structure type
     if (GEP->getPointerOperand()->getType()->getPointerElementType()->isArrayTy()) {
       llvm::outs() << "is store to array\n";
     } else if (GEP->getPointerOperand()->getType()->getPointerElementType()->isStructTy()) {
       llvm::outs() << "is store to struct\n";
     }
    } else {
     // accessing plain memory
     llvm::outs() << "plain store\n";
    }
```

Types in LLVM



Special functions: LLVM intrinsics and libc

- Intrinsics functions prefixed with **`llvm.`**
- Describing semantics using functions (rather than instructions)
- Less effort than inserting a new instruction
- Sufficient for most purposes
- Semantics only
  - Code generator handles intrinsics
    i. Software implementation
    ii. Hardware implementation
  - **`llvm.memcpy`**
  - **`llvm.sqrt`**
  - **`llvm.lifetime.start`**
  - **...**

- libc functions
  - **`malloc`**
  - **`free`**
  - **`sin`**
  - **`printf`**
  - **`sqrt`**
- Next level of lowering
- Functions boil down to system calls
- Assembly at the very end of this chain
- Calls into libc are not followed
- Model the semantics of these functions in your flow functions
- Phasar models them as identity (default)

Setting up the analysis domain

- Phasar allows to switch types of analysis domain
  - In our taint analysis we used
    - `const llvm::Value *` as domain for our data-flow facts
    - `const llvm::Instruction *` as domain for our control-flow nodes
      - Is fixed when analyzing LLVM IR
    - `const llvm::Function *` as domain for our procedures
      - Is fixed when analyzing LLVM IR
    - `LLVMBasedICFG &` as domain for our inter-procedural control-flow graph
      - Is a sane default
      - Can be replaced by
        - `LLVMBasedBackwardsICFG &`
        - Or your own implementation

Choosing the target data types

- Types of the analysis domain
  - M
    - Function type
  - N
    - Node type
  - D
    - Data-flow fact type
  - V
    - Value fact type
  - I
    - Control-flow graph type
- Types depend on
  - IR and compiler framework
  - Inter-procedural control-flow graph
  - Concrete analysis
    - Data-flow fact domain
    - Value domain
- Types can be chosen via
  i. Templates (compile-time)
  ii. Wrapper (run-time)

data type

noun COMPUTING

a particular kind of data item, as defined by the values it can take, the programming language used, or the operations that can be performed on it.

Caution with intermediate representations

- Clang(++) front-end generates naïve IR
  - This kind of IR is generated using **`$ clang -emit-llvm -S myfile.cpp`**
- The IR becomes clever and sophisticated through application of passes
  - Semantically they *should* be equivalent
  - Passes may *completely* change what the IR looks like!
  - Advice
    1) Start writing analysis for naïve IR
    2) Apply passes and revise the IR and your analysis
    3) Run your analysis on IR compiled with production flags

Caution with intermediate representations

- Compile debug information into the code `-g`

```
define i32 @main(i32, i8**) #0 !dbg !7 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  %6 = alloca i32, align 4
  %7 = alloca i32, align 4
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  call void @llvm.dbg.declare(metadata i32* %4, metadata !14, metadata !15), !dbg !16
  store i8** %1, i8*** %5, align 8
  call void @llvm.dbg.declare(metadata i8*** %5, metadata !17, metadata !15), !dbg !18
  call void @llvm.dbg.declare(metadata i32* %6, metadata !19, metadata !15), !dbg !20
  store i32 42, i32* %6, align 4, !dbg !20
  call void @llvm.dbg.declare(metadata i32* %7, metadata !21, metadata !15), !dbg !22
  store i32 13, i32* %7, align 4, !dbg !22
  ret i32 0, !dbg !23
}
declare void @llvm.dbg.declare(metadata, metadata, metadata) #1
!14 = !DILocalVariable(name: "argc", arg: 1, scope: !7, file: !1, line: 1, type: !10)
!15 = !DIExpression()
!16 = !DILocation(line: 1, column: 14, scope: !7)
!17 = !DILocalVariable(name: "argv", arg: 2, scope: !7, file: !1, line: 1, type: !11)
!18 = !DILocation(line: 1, column: 27, scope: !7)
!19 = !DILocalVariable(name: "i", scope: !7, file: !1, line: 2, type: !10)
!20 = !DILocation(line: 2, column: 6, scope: !7)
!21 = !DILocalVariable(name: "j", scope: !7, file: !1, line: 3, type: !10)
!22 = !DILocation(line: 3, column: 6, scope: !7)
!23 = !DILocation(line: 4, column: 2, scope: !7)
```

```
int main(int argc, char **argv) {
   int i = 42;
   int j = 13;
   return 0;
}
```

- This IR is truncated!

Caution with intermediate representations

- Compile with optimizations enabled `-O2`

```
int main(int argc, char **argv) {
   int i = 42;
   int j = 13;
   return 0;
}
```

```
define i32 @main(i32, i8** nocapture readnone) {
  ret i32 0
}
```

Caution with intermediate representations

- Compile with optimizations enabled `-O2`

```
int main(int argc, char **argv) {
    int i = 42;
    int j = 13;
    return i + j;
}
```

```
define i32 @main(i32, i8** nocapture readnone) {
  ret i32 55
}
```

Caution with intermediate representations

- Compile with optimizations enabled `-O2`

```
int sum(int to) {
    int sum = 1;
    for (int i = 1; i <= to; ++i) {
        sum += i;
    }
    return sum;
}
```

```
define i32 @_Z3sumi(i32) {
  %2 = icmp slt i32 %0, 0
  br i1 %2, label %11, label %3

; <label>:3:
; preds = %1
  %4 = zext i32 %0 to i33
  %5 = add i32 %0, -1
  %6 = zext i32 %5 to i33
  %7 = mul i33 %4, %6
  %8 = lshr i33 %7, 1
  %9 = trunc i33 %8 to i32
  %10 = add i32 %9, %0
  br label %11

; <label>:11:
; preds = %3, %1
  %12 = phi i32 [ 0, %1 ], [ %10, %3 ]
  ret i32 %12
}
```

What IR to use?

- Specify preliminary analysis on simple targets
- No optimization used
- Test/ refine the analysis on IR obtained using the production flags

- Handle larger projects with **`compile_commands.json`**
  - Makefile
    - **`$ bear make`**
  - Cmake
    - **`$ cmake ... a bunch of flags ... -DCMAKE_EXPORT_COMPILE_COMMANDS=ON`**
- Easily toolable with some python scripts
  - Just one loop

```
[
    {
        "arguments": [
            "c++",
            "-c",
            "-std=c++14",
            "-Wall",
            "-Wextra",
            "-o",
            "main",
            "main.cpp"
        ],
        "directory": "/tmp",
        "file": "main.cpp"
    },
...
```

Compile production code to IR

- Naïve and simplified

```
with open(compile_db) as json_db:
    json_data = json.load(json_db)
    for compilation_unit in json_data:
        file = compilation_unit["file"]
        ...
        command = ... compile file to LLVM IR ...
        output = subprocess.check_output(command, shell=True, stderr=subprocess.STDOUT)
        if output:
            print(output.decode("utf-8"))
```

```
[
    {
        "arguments": [
            "c++",
            "-c",
            "-std=c++14",
            "-Wall",
            "-Wextra",
            "-o",
            "main",
            "main.cpp"
        ],
        "directory": "/tmp",
        "file": "main.cpp"
    }
...
```

Thank you for your attention

Questions?