HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Static Analysis for C++ with Phasar

Philipp Schubert
philipp.schubert@upb.de

Ben Hermann
ben.hermann@upb.de

Eric Bodden
eric.bodden@upb.de

SOFTWARE
TECHNIK

Who are we?

Philipp Schubert

- Chief Developer of PHASAR
- Teaches C/C++ for several years
- PhD student at Paderborn University

Ben Hermann

- Committer for PHASAR
- Works on Soot and OPAL
- PostDoc at Paderborn University

Eric Bodden

- Chief Maintainer of Soot
- Teaches program analysis for several years
- Professor at Paderborn University

Who are you?

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN
Fraunhofer
IEM

Our static analysis frameworks

Soot

- De-facto Standard for Program Analysis of Java and Android bytecode and source code
- Used by >1500 research groups worldwide
- Foundation for most of our current analysis tools

FlowDroid

- De-facto Standard for taint analysis of Android apps
- Used by >1000 research groups worldwide
- Used in productive use as part of one of the world‘s largest app stores

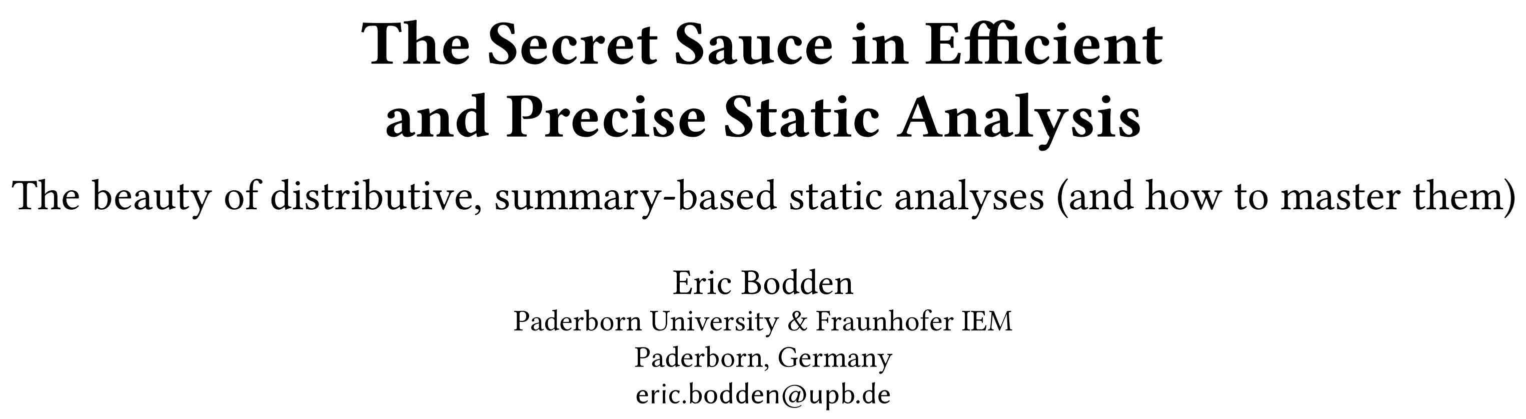
Phasar

- New framework based on LLVM
- Shares some design ideas with Soot
- Focus on C/C++ for now
- First release past week

Some design principles that we use

- Simple abstract domains:
  - Do not track numeric values
  - Generally do not attempt to interpret/correlate branches
- Instead use context-sensitivity and flow-sensitivity
- Make use of frameworks that support procedure summaries
- Hence typically try to encode problems in a distributive way

- See SOAP'18 paper



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 & Fraunhofer IEM
Paderborn, Germany
eric.bodden@upb.de

Concrete analysis tool

CogniCrypt

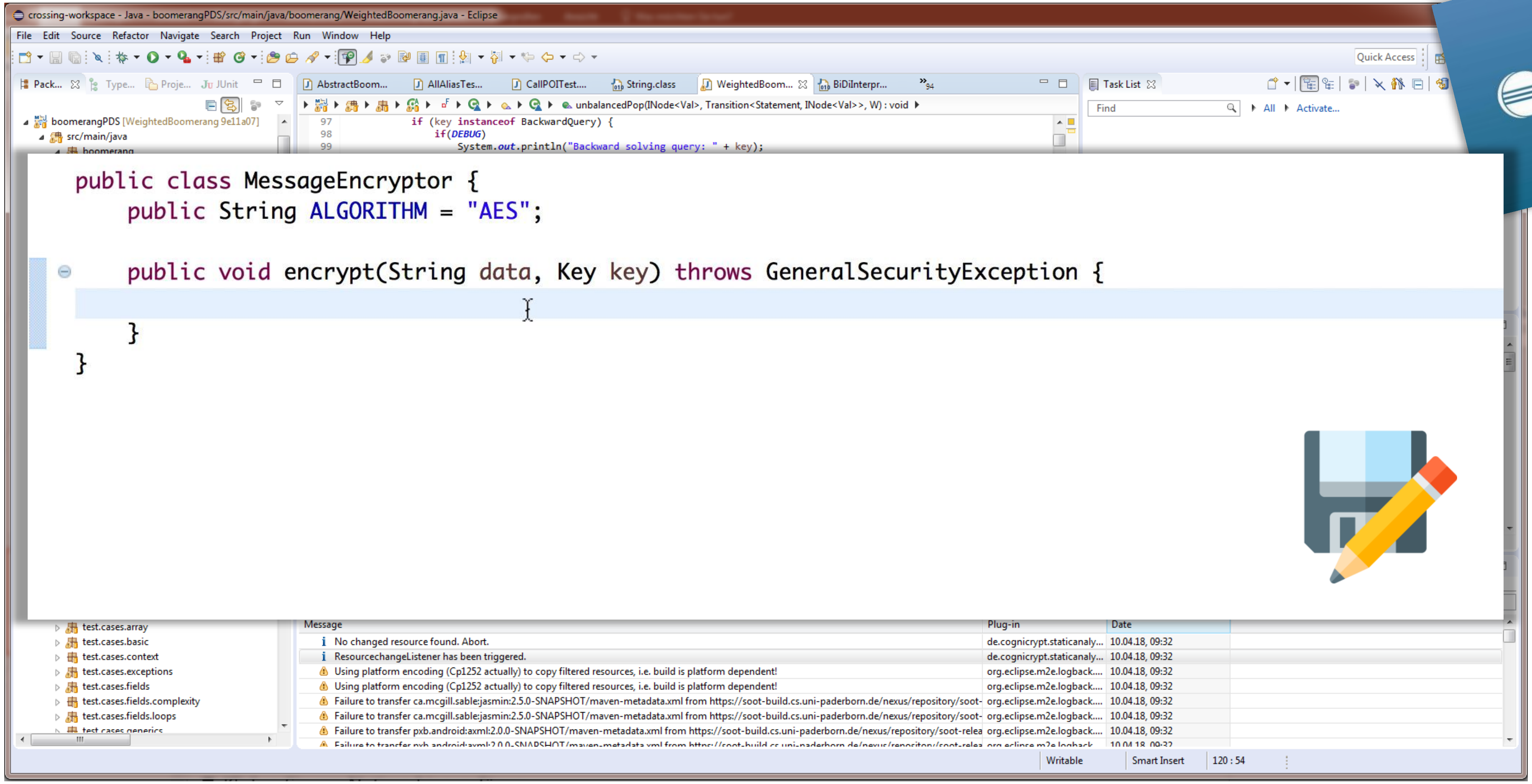
AES, RSA, CBC
…
Private Data,
Passwords,
Data Encryption
≠

Crypto Expert

Software Developer

Concrete analysis tool

CogniCrypt


crossing-workspace - Java - boomerangPDS/src/main/java/boomerang/WeightedBoomerang.java - Eclipse

```
        if (key instanceof BackwardQuery) {
            if(DEBUG)
                System.out.println("Backward solving query: " + key);
```

```
public class MessageEncryptor {
    public String ALGORITHM = "AES";

    public void encrypt(String data, Key key) throws GeneralSecurityException {

    }
}
```

Message	Plug-in	Date
No changed resource found. Abort.	de.cognicrypt.staticanaly...	10.04.18, 09:32
ResourcechangeListener has been triggered.	de.cognicrypt.staticanaly...	10.04.18, 09:32
Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!	org.eclipse.m2e.logback....	10.04.18, 09:32
Using platform encoding (Cp1252 actually) to copy filtered resources, i.e. build is platform dependent!	org.eclipse.m2e.logback....	10.04.18, 09:32
Failure to transfer ca.mcgill.sable:jasmin:2.5.0-SNAPSHOT/maven-metadata.xml from https://soot-build.cs.uni-paderborn.de/nexus/repository/soot-	org.eclipse.m2e.logback....	10.04.18, 09:32
Failure to transfer ca.mcgill.sable:jasmin:2.5.0-SNAPSHOT/maven-metadata.xml from https://soot-build.cs.uni-paderborn.de/nexus/repository/soot-	org.eclipse.m2e.logback....	10.04.18, 09:32
Failure to transfer pxb.android:axml:2.0.0-SNAPSHOT/maven-metadata.xml from https://soot-build.cs.uni-paderborn.de/nexus/repository/soot-relea	org.eclipse.m2e.logback....	10.04.18, 09:32
Failure to transfer pxb.android:axml:2.0.0-SNAPSHOT/maven-metadata.xml from https://soot-build.cs.uni-paderborn.de/nexus/repository/soot-relea	org.eclipse.m2e.logback....	10.04.18, 09:32

Writable | Smart Insert | 120 : 54

www.cognicrypt.org

The C++ Programming Language

C++ is easy.

It´s like riding a bike.

Except the bike is on fire,

and you´re on fire

and everything is on fire

because everything is hot lava.

The C++ Programming Language

C++ is easy.

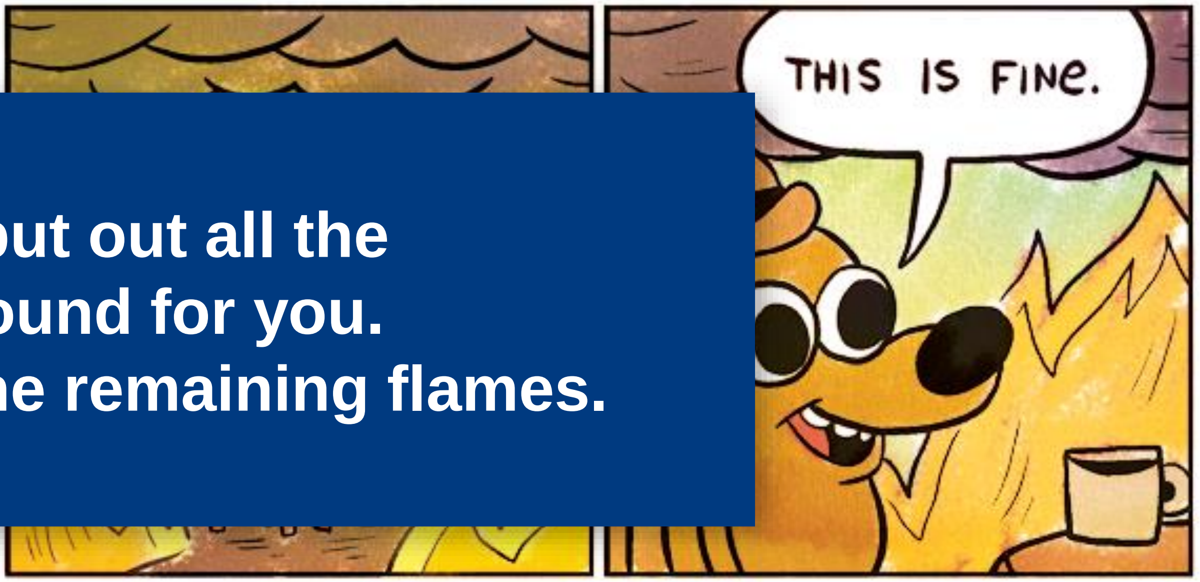
It´s like r

Except the

and yo

and every

because everything is hot lava.



We have put out all the fires we found for you.
Please excuse the remaining flames.

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Structure of this Tutorial

Block 1

- Introduction of PHASAR and LLVM
- Using the command-line
- Generating call graphs, etc.

Block 2

- Writing custom analyses
- Taint analysis

Block 3

- Data-flow analyses with IFDS
- Data-flow analyses with IDE

Block 4

- Analysis performance measurement
- Debugging
- Q&A
- Discussions

What you will need for this tutorial…

VirtualBox

We have prepared a Virtual Box image with a complete installation of PHASAR, all the code examples, and an editor. You should have already downloaded that. If not, here it is:

https://drive.google.com/open?id=1F4wehrMB7NyVgsI5ebTaIDaQSstffGJ1

Docker

If you prefer your own environment, we have prepared a docker container for you. Just run:

```
docker run -ti bhermann/phasar:pldi18
```

Compile your own

If you are working with Linux or MacOS, you can compile your own version of PHASAR. Just check out our Git repo and follow the instructions. We can give you the code examples.

https://github.com/secure-software-engineering/phasar

Check your installation now please.

You should be able to run the phasar executable.

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Static Analysis for C++ with Phasar

Block 1

Philipp Schubert
`philipp.schubert@upb.de`

Ben Hermann
`ben.hermann@upb.de`

Eric Bodden
`eric.bodden@upb.de`

SOFTWARE
TECHNIK

In this Block

1. What is PHASAR? What is it not?
2. Basics of LLVM
3. PHASAR – Architecture and Features
4. Analyzing programs from the command line

What is PHASAR?

- PHASAR is a static analysis framework for C/C++
- It is based on LLVM
- Its main focus is on data flow analysis
- But you can do other things with it too
- We aim to design it highly flexible to allow users to develop their own analyses

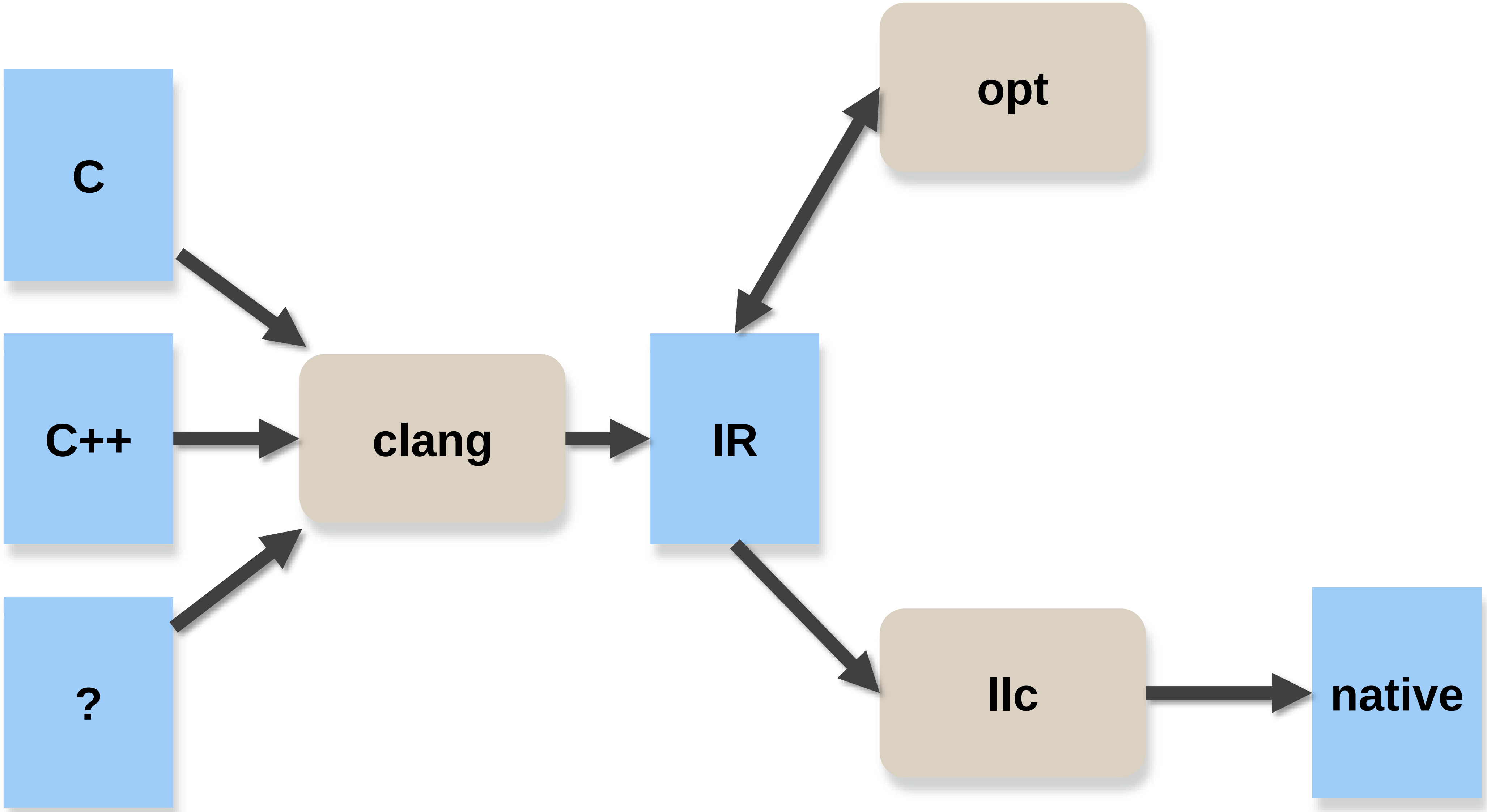
What PHASAR is not?

- PHASAR is not a substitute for LLVM, it complements LLVM
- It is not meant to work as a compiler pass

LLVM

- The LLVM compiler infrastructure is a modular compiler toolchain
- It began as a research project at the University of Illinois
- Now it is a very active open source project with dozens of related projects
- It is also the standard compiler for C/C++ on XCode.

LLVM



LLVM Representation

```
int factorial (int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
  %1 = alloca i32, align 4
  %r = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %r, align 4
  br label %2

; <label>:2                                  ; preds = %5, %0
  %3 = load i32* %1, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                                  ; preds = %2
  %6 = load i32* %1, align 4
  %7 = load i32* %r, align 4
  %8 = mul nsw i32 %7, %6
  store i32 %8, i32* %r, align 4
  %9 = load i32* %1, align 4
  %10 = add nsw i32 %9, -1
  store i32 %10, i32* %1, align 4
  br label %2
```

LLVM Representation

```
int factorial (int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
  %1 = alloca i32, align 4
  %r = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %r, align 4
  br label %2

; <label>:2                                       ; preds = %5, %0
  %3 = load i32* %1, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                                       ; preds = %2
  %6 = load i32* %1, align 4
  %7 = load i32* %r, align 4
  %8 = mul nsw i32 %7, %6
  store i32 %8, i32* %r, align 4
  %9 = load i32* %1, align 4
  %10 = add nsw i32 %9, -1
  store i32 %10, i32* %1, align 4
  br label %2
```

Data Layout (dash separated)
little-endian
ELF mangling
64-bit integers
80 to 128-bit floats
native CPU integers are 8, 16, 32, and 64 -bit
Stack alignment is 128-bits

LLVM Representation

```
int factorial (int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
  %1 = alloca i32, align 4
  %r = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %r, align 4
  br label %2

; <label>:2                                       ; preds = %5, %0
  %3 = load i32* %1, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                                       ; preds = %2
  %6 = load i32* %1, align 4
  %7 = load i32* %r, align 4
  %8 = mul nsw i32 %7, %6
  store i32 %8, i32* %r, align 4
  %9 = load i32* %1, align 4
  %10 = add nsw i32 %9, -1
  store i32 %10, i32* %1, align 4
  br label %2
```

Target Triple (dash separated)
Architecture (e.g., x86_64, ARM, PowerPC)
Vendor (e.g., pc, apple)
Operating System (e.g., linux, macosx10.7.0)
Environment (e.g., gnu)

LLVM Representation

```
int factorial (int n) {
    int r = 1;
    while (n > 0) {
        r *= n;
        n--;
    }
    return r;
}
```

```
; ModuleID = 'factorial.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @factorial(i32 %n) #0 {
  %1 = alloca i32, align 4
  %r = alloca i32, align 4
  store i32 %n, i32* %1, align 4
  store i32 1, i32* %r, align 4
  br label %2

; <label>:2                              ; preds = %5, %0
  %3 = load i32* %1, align 4
  %4 = icmp sgt i32 %3, 0
  br i1 %4, label %5, label %11

; <label>:5                              ; preds = %2
  %6 = load i32* %1, align 4
  %7 = load i32* %r, align 4
  %8 = mul nsw i32 %7, %6
  store i32 %8, i32* %r, align 4
  %9 = load i32* %1, align 4
  %10 = add nsw i32 %9, -1
  store i32 %10, i32* %1, align 4
  br label %2
```

Function Definition
Return type (here: i32 —> 32-bit Integer)
Function name (@-prefixed —> global identifier)
Argument list (%-prefixed —> local identifier)
Attribute reference

Down to Machine Code

```
	.text
	.file	"factorial.c"
	.globl	factorial
	.align	16, 0x90
	.type	factorial,@function
factorial:                              # @factorial
	.cfi_startproc
# BB#0:
	pushq	%rbp
.Ltmp0:
	.cfi_def_cfa_offset 16
.Ltmp1:
	.cfi_offset %rbp, -16
	movq	%rsp, %rbp
.Ltmp2:
	.cfi_def_cfa_register %rbp
	movl	%edi, -4(%rbp)
	movl	$1, -8(%rbp)
.LBB0_1:                                # =>This Inner Loop Header: Depth=1
	cmpl	$0, -4(%rbp)
	jle	.LBB0_3
# BB#2:                                 #   in Loop: Header=BB0_1 Depth=1
	movl	-4(%rbp), %eax
	imull	-8(%rbp), %eax
	movl	%eax, -8(%rbp)
	movl	-4(%rbp), %eax
	addl	$4294967295, %eax       # imm = 0xFFFFFFFF
	movl	%eax, -4(%rbp)
	jmp	.LBB0_1
```

LLVM Intermediate Representation (IR)

- RISC-like instruction set, but
  - Strongly typed — every instruction has typed arguments
  - Explicit control flow
  - Explicit data flow (Static Single Assignment form)

Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

Target register

Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

Instruction

Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

No signed wrap

Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

Type

Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

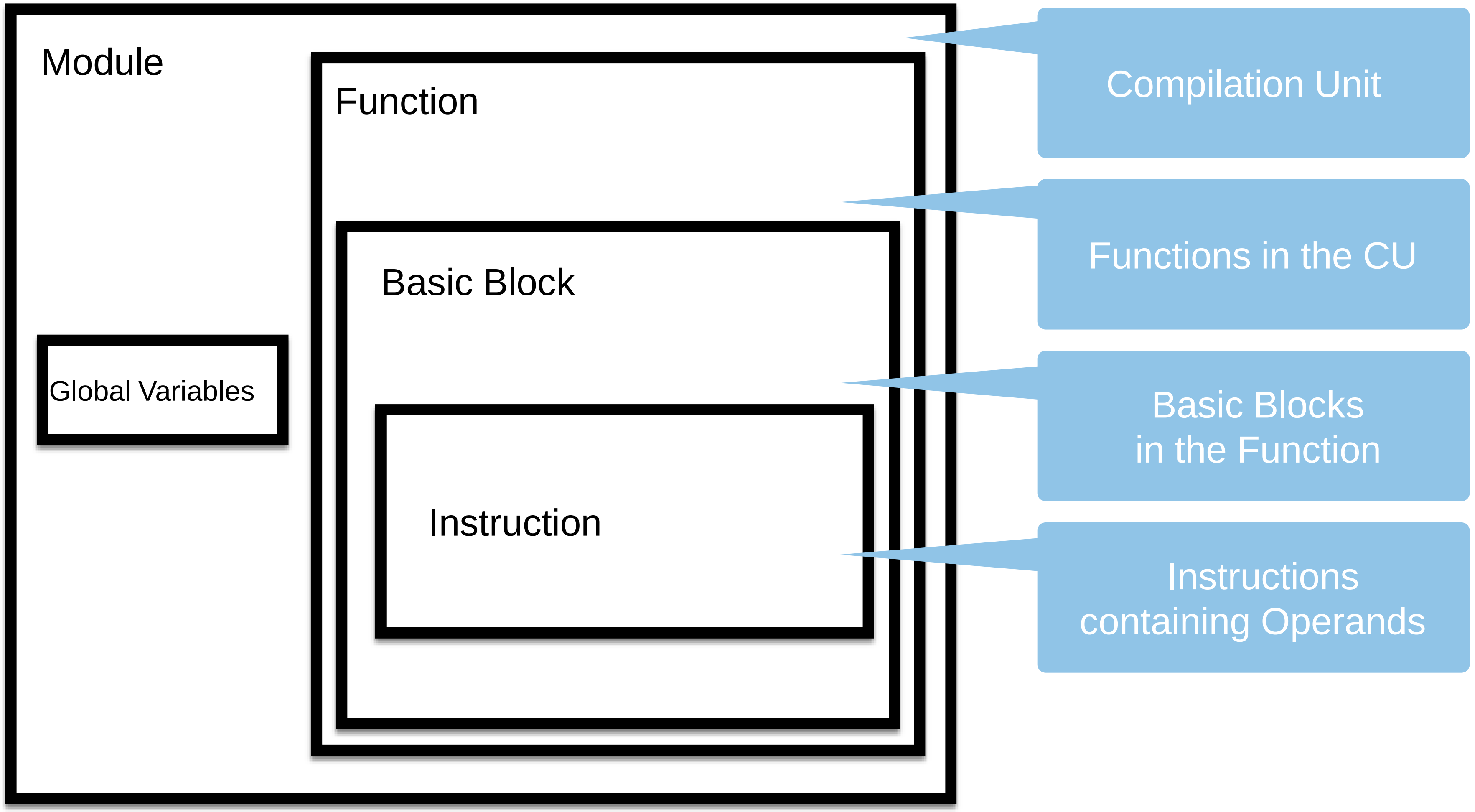
Operand 1
here source register

Instructions in LLVM IR

```
%10 = add nsw i32 %9, -1
```

Operand 2
here a constant

Scopes in LLVM



Using LLVM

- As PHASAR is build on top of LLVM
- It relies on its IR and its API
- Therefore, we would like to familiarize you with some LLVM tools

Compilation

Try this in your PHASAR environment Folder: analysis/target/

- Compiling a C file to LLVM IR

```
clang -S -emit-llvm factorial.c -o -
```

Only run preprocess and compilation steps (outputs human readable format)

- Taking a look at the structure of LLVM‘s C frontend

```
clang -cc1 -ast-dump factorial.c
```

- Try this out on your environment. Inspect the resulting output and compare with the input file

Compilation

Try this in your PHASAR environment Folder: analysis/target/

- Compiling a C file to LLVM IR

```
clang -S -emit-llvm factorial.c -o -
```

Produce LLVM IR

- Taking a look at the structure of LLVM‘s C frontend

```
clang -cc1 -ast-dump factorial.c
```

- Try this out on your environment. Inspect the resulting output and compare with the input file

Compilation

Try this in your PHASAR environment Folder: analysis/target/

- Compiling a C file to LLVM IR

```
clang -S -emit-llvm factorial.c -o -
```

Input file

- Taking a look at the structure of LLVM‘s C frontend

```
clang -cc1 -ast-dump factorial.c
```

- Try this out on your environment. Inspect the resulting output and compare with the input file

Compilation

Try this in your PHASAR environment Folder: analysis/target/

- Compiling a C file to LLVM IR

```
clang -S -emit-llvm factorial.c -o -
```

Output to stdout

- Taking a look at the structure of LLVM‘s C frontend

```
clang -cc1 -ast-dump factorial.c
```

- Try this out on your environment. Inspect the resulting output and compare with the input file

Compilation

Try this in your PHASAR environment
Folder: analysis/target/

- Compiling a C file to LLVM IR

```
clang -S -emit-llvm factorial.c -o -
```

- Taking a look at the structure of LLVM‘s C frontend

Use only compiler front-end

```
clang -cc1 -ast-dump factorial.c
```

- Try this out on your environment. Inspect the resulting output and compare with the input file

Compilation

Try this in your
PHASAR environment
Folder: analysis/target/

- Compiling a C file to LLVM IR

```
clang -S -emit-llvm factorial.c -o -
```

- Taking a look at the structure of LLVM‘s C frontend

Dump the complete
AST on the console

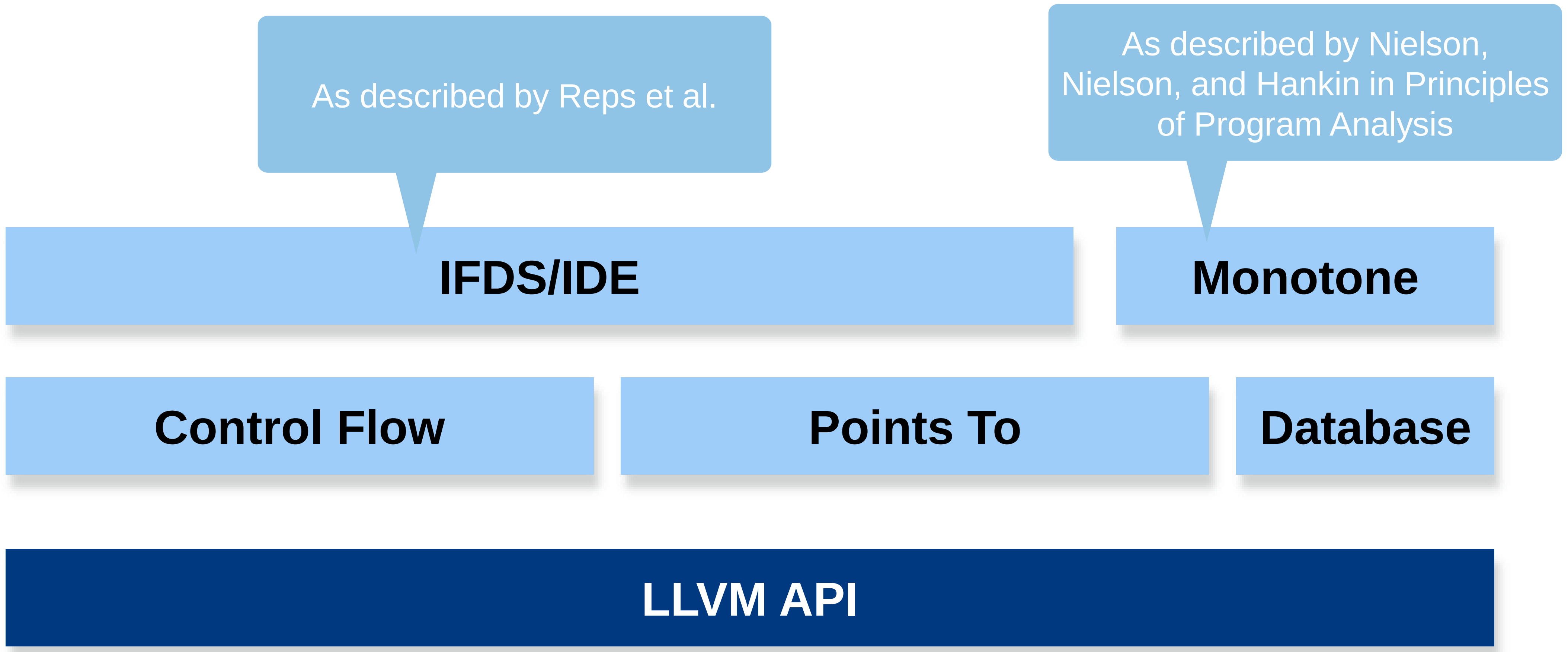
```
clang -cc1 -ast-dump factorial.c
```

- Try this out on your environment. Inspect the resulting output and compare with the input file

PHASAR

- PHASAR is a LLVM-based static analysis framework written in C++.
- It allows users to specify arbitrary **data-flow problems** which are then solved in a fully-automated manner on the specified LLVM IR target code.
- Computing points-to information, call-graphs, etc. is done by the framework.
- Before we look into data-flow problems in the next two block, we will look at:
  - PHASAR’s architecture
  - Using PHASAR from the command-line to extract some data structures

Architecture of the Framework


As described by Reps et al.
As described by Nielson, Nielson, and Hankin in Principles of Program Analysis
IFDS/IDE
Monotone
Control Flow
Points To
Database
LLVM API

How can you use it?

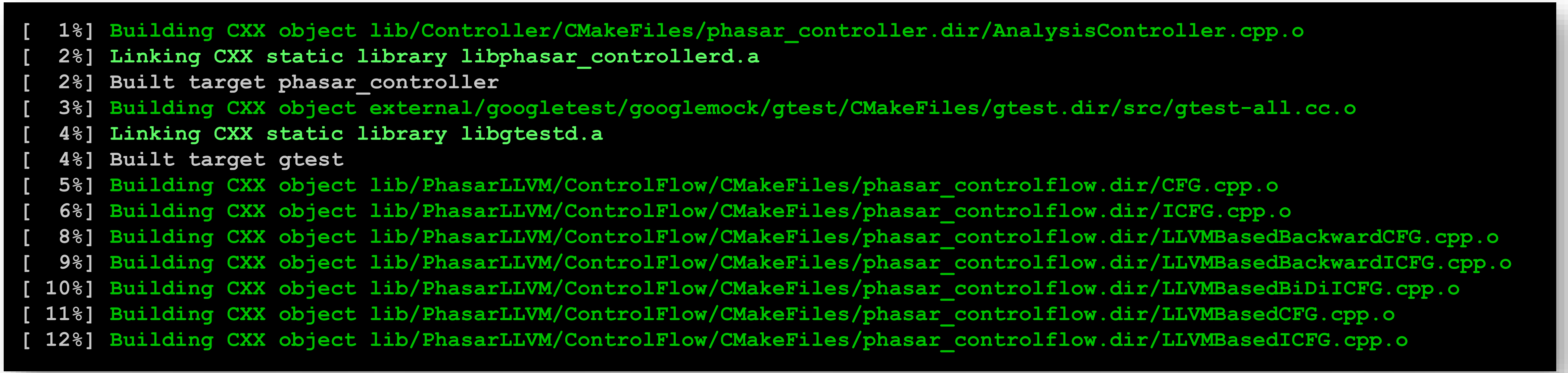
- Library – Use PHASAR functionality in your program (when you build PHASAR as shared object libraries)
- Framework – Extend it with your own analysis
- Runtime – Write plugins for PHASAR
- Executable – Run it from the command line and directly use the output

A Quick Word on Compilation

- Compiling PHASAR ***should*** be as easy as 1, 2, 3…
- However, it still is a C++ project… So in your environment it might be more difficult.
- Dependencies you will need:
  - LLVM/Clang 5.0 (llvm-5.0-dev clang-5.0 libclang-5.0-dev)
  - SQLite 3.11.0 or newer (libsqlite3-dev)
  - MySql Connector (libmysqlcppconn-dev)
  - LibCurl (libcurl4-openssl-dev)
  - Zlib (zlib1g-dev)
  - Boost 1.63.0 or newer (for common Linux no stable package available, has to be self compiled, for Homebrew 1.66.0 is available and it works)
  - Python 3 (helpful, but not necessary)
  - CMake
- On a Mac, you just need homebrew and then type **`brew bundle`** in the project directory

A Quick Word on Compilation

- Afterwards, it should be as easy as:
  - `mkdir build`
  - `cd build`
  - `cmake ..`
  - `make`



```
[  1%] Building CXX object lib/Controller/CMakeFiles/phasar_controller.dir/AnalysisController.cpp.o
[  2%] Linking CXX static library libphasar_controllerd.a
[  2%] Built target phasar_controller
[  3%] Building CXX object external/googletest/googlemock/gtest/CMakeFiles/gtest.dir/src/gtest-all.cc.o
[  4%] Linking CXX static library libgtestd.a
[  4%] Built target gtest
[  5%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/CFG.cpp.o
[  6%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/ICFG.cpp.o
[  8%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/LLVMBasedBackwardCFG.cpp.o
[  9%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/LLVMBasedBackwardICFG.cpp.o
[ 10%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/LLVMBasedBiDiICFG.cpp.o
[ 11%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/LLVMBasedCFG.cpp.o
[ 12%] Building CXX object lib/PhasarLLVM/ControlFlow/CMakeFiles/phasar_controlflow.dir/LLVMBasedICFG.cpp.o
```

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Try for Yourself

/home/phasar/Programs/phasar

- Go to the folder of the PHASAR source code.
- Type in:
  - `cd build`
  - `make clean`
  - `cmake ..`
  - `make`

Using the command-line interface

- The easiest way to use PHASAR is using it from the command line
- However, you are then limited to the provided, pre-configured analyses

Command-Line Arguments – Basic Functionality

```
  --config arg                          Path to the configuration file, options
                                        can be specified as 'parameter = option'
  --silent                              Suppress any non-result output
```

- Configuration files contain key value pairs for any command-line argument
- We will be using configuration files later
- If you want to pipe the output to another program the **`silent`** option leaves out any unnecessary output

Command-Line Arguments – Input Definition

```
-f [ --function ] arg                 Function under analysis (a mangled
                                      function name)
-m [ --module ] arg                   Path to the module(s) under analysis
-p [ --project ] arg                  Path to the project under analysis
-E [ --entry_points ] arg             Set the entry point(s) to be used
```

- With these options you can form the input to the analysis.
- If you want to limit your analysis to specific functions you can use the `--function` option.
  - We expect a so-called mangled function name here. That is a function name after the LLVM process. For example: `_Z5printi`
- The `--module` option is the most commonly used option. Here you have to provide a list of LLVM IR module files (.ll-files)
- If you do not want to create a call-graph from the main function you can use the `--entry-points` option to specificy the functions you want to start.

Command Line Arguments – Output Definition

```
-O [ --output ] arg (=results.json)  Filename for the results
```

- By default PHASAR outputs any information gathered by the three basic analyses in a file named **`results.json`**.
- You can change this by providing the **`--output`** option.
- We will later provide more information on the file’s contents.

Command Line Arguments – Controlling Upstream Analyses

```
-P [ --pointer_analysis ] arg        Set the points-to analysis to be used
                                     (CFLSteens, CFLAnders)
-C [ --callgraph_analysis ] arg      Set the call-graph algorithm to be used
                                     (CHA, RTA, DTA, VTA, OTF)
-H [ --classhierachy_analysis ] arg  Class-hierarchy analysis
-V [ --vtable_analysis ] arg         Virtual function table analysis
```

- For pointer analysis and call-graph analysis, we provide different implementations.
- Depending on the client analysis you would like to implement, you will prefer a different algorithm.
- The **`--classhierarchy_analysis`** and **`--vtable_analysis`** are boolean and accept either '**`on|off`**', '**`yes|no`**', '1|0' or '**`true|false`**' depending on your preference.
- The **`--classhierarchy_analysis`** option produces a class hierarchy for C++ projects.
- The **`--vtable_analysis`** option reconstructs a virtual function table during this process.

Command Line Arguments – Analysis Options

`-W [ --wpa ] arg (=1)`	`Whole-program analysis mode (1 or 0)`
`-M [ --mem2reg ] arg (=1)`	`Promote memory to register pass (1 or 0)`
`-R [ --printedgerec ] arg (=0)`	`Print exploded-super-graph edge recorder (1 or 0)`

- In some cases it might be sufficient not to analyze the whole program.
  In these cases, you can set **`--wpa`** to **`0`**.
- The **`--mem2reg`** option activates the mem2reg pass from LLVM, which promotes memory references to be register references. This makes downstream analyses easier.
- The **`--printedgerec`** option can be activated to print out the exploded super graphs for debugging data-flow analyses.

Command line arguments – Plugin Mechanism

```
  --analysis_plugin arg                Analysis plugin(s) (absolute path to the
                                       shared object file(s))
  --callgraph_plugin arg               ICFG plugin (absolute path to the shared
                                       object file)
```

- We provide two interfaces to include shared-object library plugins into the PHASAR workflow.
- For security purposes these parameters must be given as absolute paths.
- An analysis plugin is a general purpose plugin. We will write such a plugin later.
- A call graph plugin can be used for any other analysis and replaces the standard implementations from PHASAR.

Generating Class Hierarchies

Try it out in:
analyses/target/

- A class hierarchy is a data structure for object-oriented programs.
- Thus, for PHASAR you will need a program in C++.
- Nodes in this graph are classes.
- Edges are supertype-subtype relations between these classes
- You can generate such a class hierarchy by simply calling:

```
phasar –m *.cpp.ll
```

- The class hierarchy is written to the output file. The default file is results.json.
- You can alter the target file by using the `--output` option.
- It will be overwritten every time.

The results.json file

```
[
 {
  "TypeHierarchy": …
 },
 {
  "CallGraph": …
 },
 {
  "PointsToGraph": …
 }
]
```

- The output of these previous three analyses are stored in the results file.
- Data is presented in the JavaScript Object Notation (JSON) format.
- It is a general format to interchange data between systems.
- It supports objects, arrays, and attribute-value pairs.

Type Hierachy Results

```
"TypeHierarchy": {
  "struct.S": [
   "struct.T"
  ],
  "struct.T": null
 }
```

- In the TypeHierarchy object all struct and class types are listed.
- In the array on the right side of the attribute-value pair are the subtypes of the types on the left side.
- Here, the struct T is a subtype of struct S.

Generating Call Graphs

Try it out in:
analyses/target/

- A call graph is one of the most fundamental data structures for static analysis.
  - In this graph methods/functions are nodes
  - And calls are the edges between these nodes
- It records every call from one function to other functions.
- PHASAR creates a call graph for a module if you start it with:

```
phasar –m *.cpp.ll –C CHA
```

- The call graph is printed to the standard output.
- It is also stored in `results.json`.

Our supported algorithms are:

- `CHA (Class Hierachy Analysis)`
- `RTA (Rapid Type Analysis)`
- `DTA (Declared Type Analysis)`
- `VTA (Variable Type Analysis)`
- `OTF (On-the-fly)`
- … or your call graph plugin

Call Graph Results

```
"CallGraph": {
  "_Z5printi": null,
  "_Z5taintv": null,
  "main": [
   "_Z5taintv",
   "_Z5printi"
  ]
 }
```

- In the CallGraph object, all methods/functions in the analysis scope are listed.
- On the right side in the array, all methods are listed, that are called from the method on the left side

Generating Points-to Sets

Try it out
Folder: pldi18_tutorial/analyses /target/simple_taint_analysis

- A points-to set contains all objects and a list of the objects that point to the same memory location.
- This set is used to determine aliasing objects for other analyses.
- It can help to provide a more accurate, precise result.
- PHASAR produces points-to sets with the following call:

```
phasar -m *.cpp.ll -P CFLAnders
```

- The results of the points-to analysis are printed to the console
- The are also stored in the output file

Our supported algorithms are:

- `CFLSteens`
- `CFLAnders`

Points-To Results

```
 "PointsToGraph": {
   "  %10 = getelementptr inbounds %struct.T, %struct.T* %6, i32 0, i32 0,
!phasar.instruction.id !13, ID: 15": [
    "  %6 = alloca %struct.T, align 4, !phasar.instruction.id !5, ID: 7",
    "  %8 = getelementptr inbounds %struct.T, %struct.T* %6, i32 0, i32 0,
!phasar.instruction.id !10, ID: 12",
    "  %9 = getelementptr inbounds %struct.S, %struct.S* %8, i32 0, i32 0,
!phasar.instruction.id !11, ID: 13",
    "  %11 = getelementptr inbounds %struct.S, %struct.S* %10, i32 0, i32 1,
!phasar.instruction.id !14, ID: 16"
   ], ...
```

- In the Points-To object, all allocation sites are listed.
- On the right side in the array, all instructions are listed, that alias with the object on the left side

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Thank you for your attention

Do you have any questions so far?