T07 Program Analysis

Static Analysis and Dynamic Analysis

Dr. Marco Elver
Systems Research Group
https://dse.in.tum.de/

Tutorial outline

- **Part I: Lecture summary**
    - **Q&A for the lecture material**

- **Part II:** Programming basics

- **Part III:** Homework programming exercises (Artemis)

Lecture overview

- **Part I:** Faults and failures in software
  - Terminology and impact
- **Part II:** Program analysis trade-offs
  - Soundness, completeness, static vs. dynamic analysis
- **Part III:** Static analysis tools
  - Compiler warnings
  - Infer
  - SpotBugs
  - Clang Analyzer and Clang Tidy
- **Part IV:** Brief introduction to C
- **Part V:** Dynamic analysis tools
  - Undefined behavior
  - Dynamic binary instrumentation
  - Compiler-assisted instrumentation
  - Valgrind
  - Sanitizers
  - Program hardening

Faults and failures in software

Not all faults cause errors, and not all errors cause failures:

1. *Code coverage:* Depending on program inputs, not all code in a program may be executed.

2. *Transient errors:* The program enters an invalid state only briefly, and there are no observable (unexpected) side-effects; this may happen when the program performs work that is aborted due to an interruption, retried, or otherwise reset.

3. *Fault detection or protection:* Erroneous behavior is discovered and corrected before it affects system services.

Avoiding faults and failures in software

Fault-avoidance: Software design and implementation process uses approaches to avoid *programming errors*, minimizing faults introduced

📜 *Expressive type systems, advanced programming languages, formal methods*

Fault-detection: Verification and validation processes to discover and remove *program faults* before deploying to "production"

🔍 *Dynamic & static program analysis* (detect faults in specific executions)

Fault-tolerance: System detects faults in specific executions at runtime, mitigated on detection.

🔍 *Low-cost dynamic program analysis* 🏛️ *Fault-tolerant software architecture*

📖 I. Sommerville, "Reliability engineering" (Chapter 11), In "Software Engineering," 2015

Program analysis is essential for software quality

1. Too costly to audit large software systems manually.
2. Program analysis techniques required for automated analysis.
3. Enables faster feature development and rollout.
4. To pick appropriate program analysis must understand desired system *availability and reliability* along with cost trade-offs.

Reliability: probability of failure-free operation over a specified time

Availability: probability that a system will be operational and deliver its services

Static Analysis

- *Static program analysis* is about analyzing a piece of code “statically”: the analysis only inspects the source code without executing or running it
- Static analysis reports can point out *system faults*, *errors*, or resulting *failures*
  - *Tools are often designed to allow for high quality diagnosis*
- Can cover large sets of states very quickly and cheaply.
- *Completeness often traded against unsoundness:* tools prefer to produce useful signals (true positives) while keeping noise (false positives) low.

📖 H. Adkins et al., “Static Program Analysis” (Chapter 13), In “Building Secure & Reliable Systems,” 2020

Static Analysis: Compiler errors, warnings & Linters

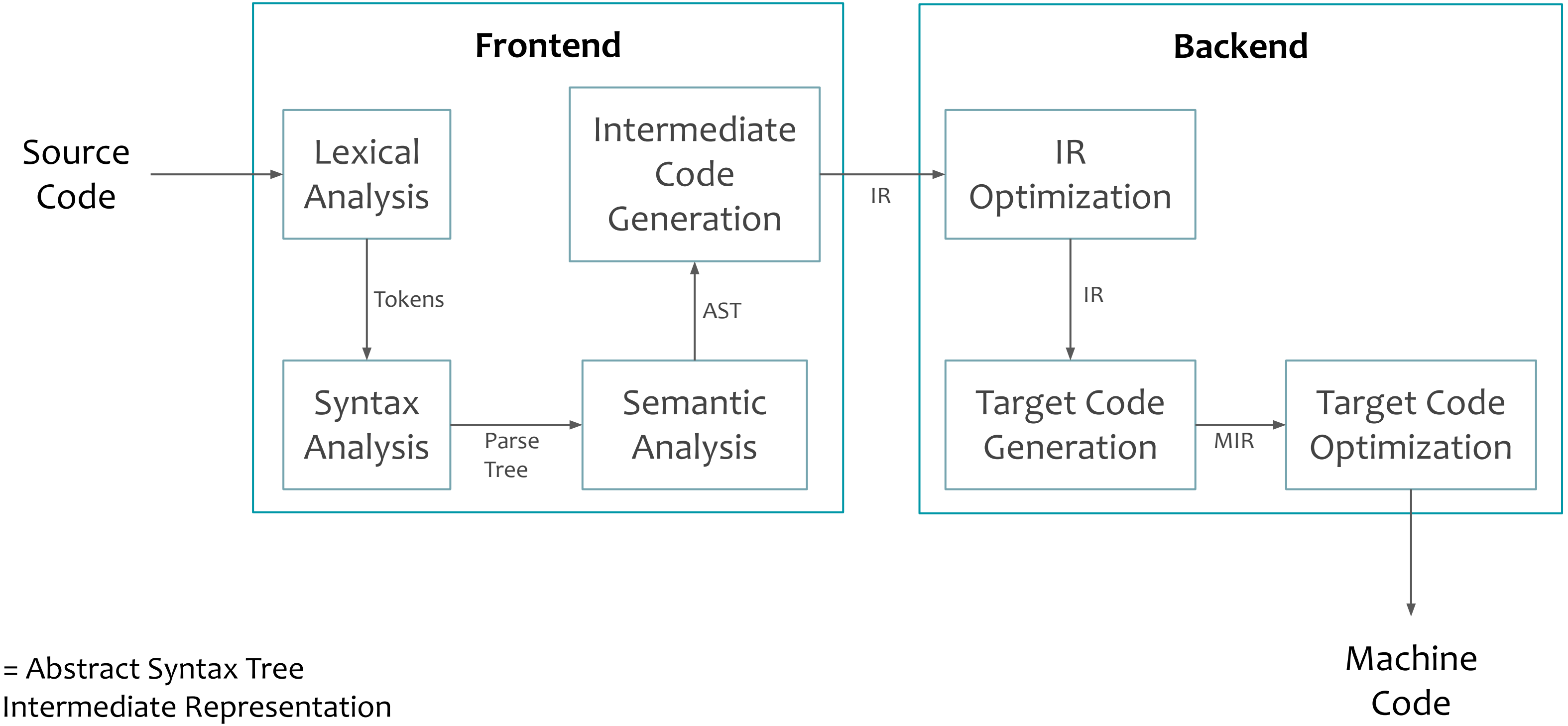
Built-in analyses by the language compiler and default toolchain:

- **Programming Languages:** many
- **False positives:** depends
- **False negatives:** depends
- **Cost:** very low

Availability:

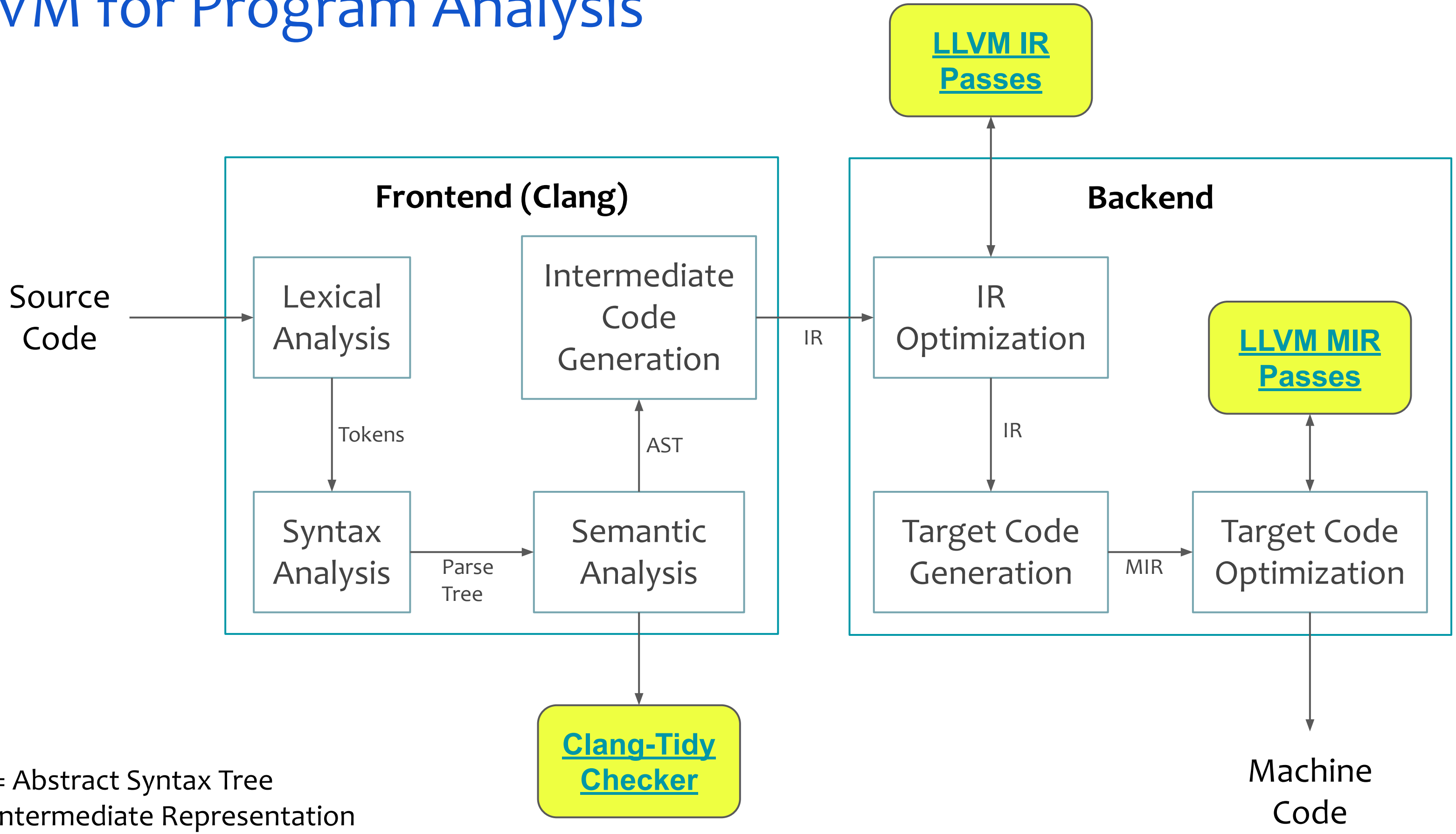
- Java (more with -Xlint)
- C/C++ with GCC, Clang, MSVC (more with -Wall, -Wextra, or /Wall respectively)
- Rust (more with Clippy)
- Many many more – check your favorite language compiler…

Compilation Pipeline



AST = Abstract Syntax Tree
IR = Intermediate Representation
MIR = Machine Intermediate Representation

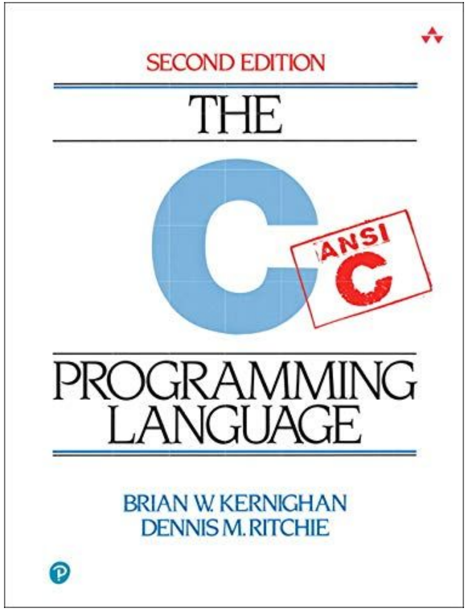
LLVM for Program Analysis



AST = Abstract Syntax Tree
IR = Intermediate Representation
MIR = Machine Intermediate Representation

Brief Introduction to C

- C is everywhere – lowest common denominator!
- Developed in the 1970s alongside UNIX.
- Access to low-level operating system facilities and libraries via C API.
- Foreign Function Interfaces (FFI) in terms of a C ABI.
- Many newer languages derived from C syntax.
- Latest version of C standard is C23.



ABI: Application Binary Interface – interface in terms of executed machine code (binary level)

API: Application Programming Interface – typically defines a source-level interface (programmer level)

Introduction to C: Arrays & Pointers

```c
// stack_arrays.c
#include <stdio.h>
#include <stdlib.h>  // provides atoi()

#define ARRAY_SIZE(x) (sizeof(x) / sizeof((x)[0]))

static void print_values(const int *values, size_t count)
{
    for (int i = 0; i < count; ++i)
        printf("value[%d] = %d, ", i, values[i]);
}
int main(int argc, char *argv[])
{
    if (argc < 2) return 1;
    const int mul = atoi(argv[1]);
    int values[32] = {};
    for (int i = 0; i < ARRAY_SIZE(values); ++i) {
        values[i] = i * mul;
        if (values[i] > 100)
            break;
    }
    print_values(values, ARRAY_SIZE(values));
    return 0;
}
```

```
$> cc -Wall -o stack_arrays stack_arrays.c
$> ./stack_arrays 5
value[0] = 0, value[1] = 5, value[2] = 10, value[3] =
15, value[4] = 20, value[5] = 25, value[6] = 30,
value[7] = 35, value[8] = 40, value[9] = 45,
value[10] = 50, value[11] = 55, value[12] = 60,
value[13] = 65, value[14] = 70, value[15] = 75,
value[16] = 80, value[17] = 85, value[18] = 90,
value[19] = 95, value[20] = 100, value[21] = 105,
value[22] = 0, value[23] = 0, value[24] = 0,
value[25] = 0, value[26] = 0, value[27] = 0,
value[28] = 0, value[29] = 0, value[30] = 0,
value[31] = 0,
```

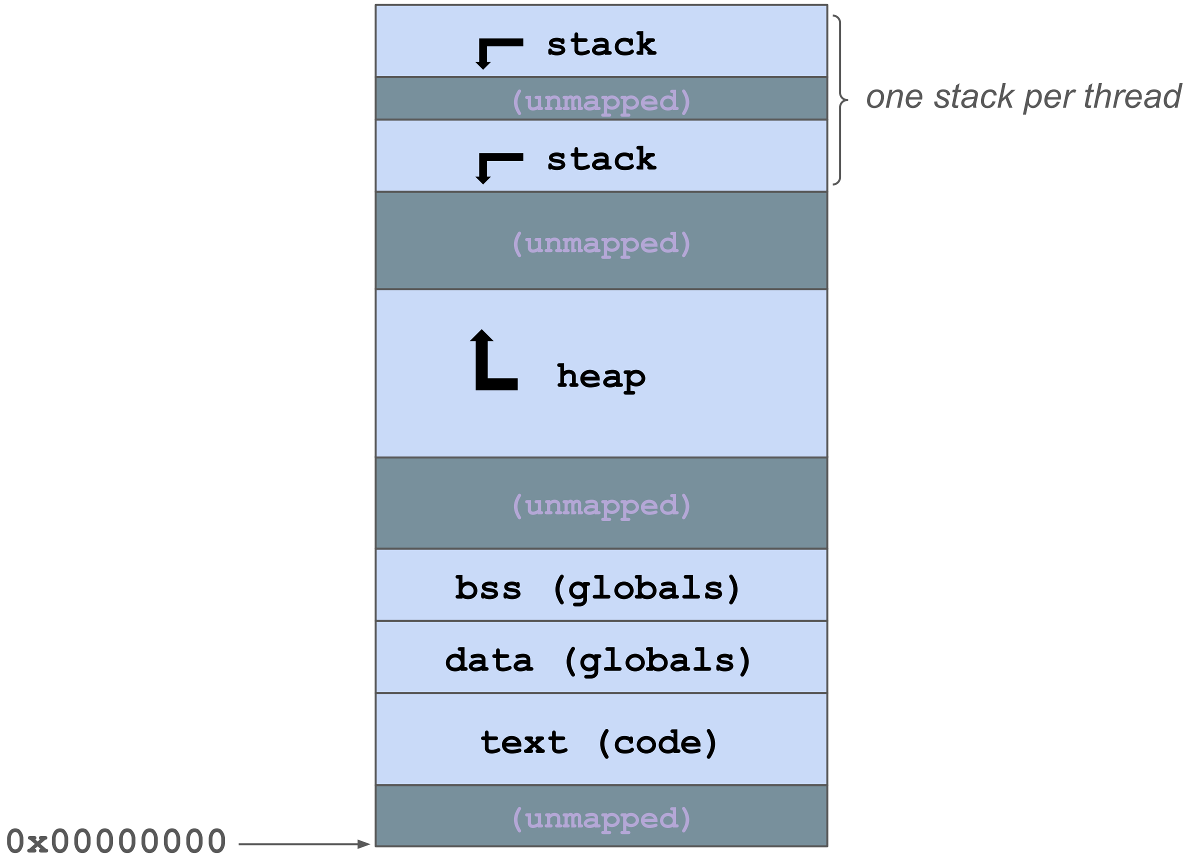
💡 C has complex initialization rules. If in doubt, explicitly initialize variables!

💡 Declare functions and global variables `static` if they are "private" to the source file.

What is memory anyway?

- Each process gets its own *virtual address space*
  - Virtual memory mapping to physical memory managed by OS kernel
  - Prohibits access to other processes' address spaces
- For each process, memory is allocated for:
  - Code
  - Globals
  - Stack (for function-local variables spilled from CPU registers)
  - Heap (dynamic memory allocation)
- Addresses may not always be the same
  - Affected by ASLR (more later)
  - Order of allocations (heap)

Process Memory Layout



Memory Safety

Memory-safety error: An illegal access to unintended memory regions.

Two types of errors:

1. **Spatial errors:** unintended address
   - buffer overflow, stack overflow (out-of-bounds)
2. **Temporal errors:** unintended time
   - double free, dangling pointers (use-after-free, use-after-return)

Heartbleed in OpenSSL
(buffer overflow)

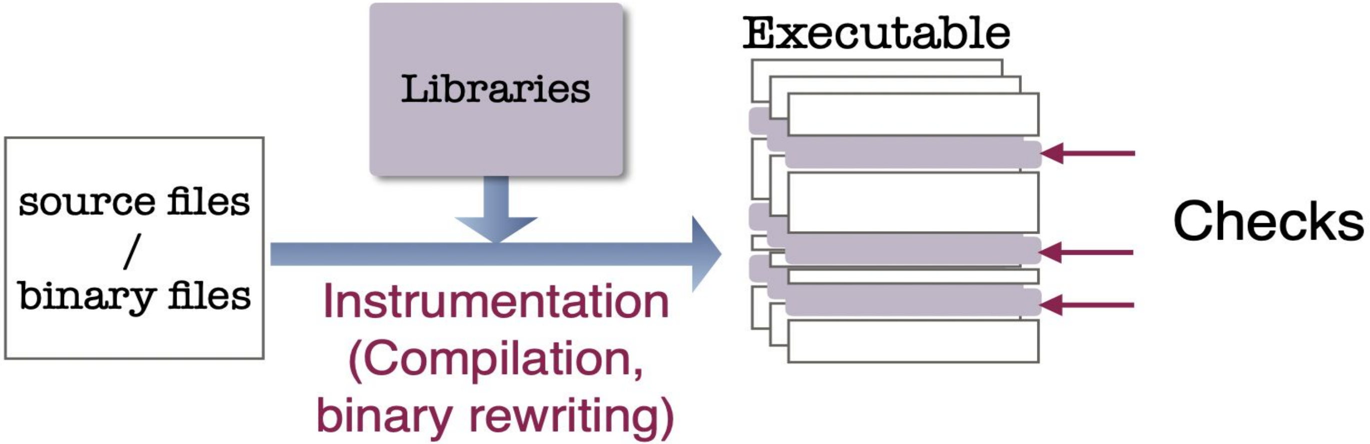
How to catch them at runtime? Can we catch them cheaply?

Dynamic Analysis

- *Dynamic program analysis* is about analyzing a piece of code "dynamically": the analysis observes the program as it is being executed.
- Dynamic analysis reports typically point out *system errors* or *failures.*
  - Can rarely deduce the underlying *system fault.*
  - Quality of diagnostics often inversely correlated with the performance of a tool.
- Only the state space that was *covered* during execution is analyzed.
- If covered state space is non-exhaustive, the analysis will always be *unsound*.

📖 H. Adkins et al., "Dynamic Program Analysis" (Chapter 13), In "Building Secure & Reliable Systems," 2020

Dynamic Analysis

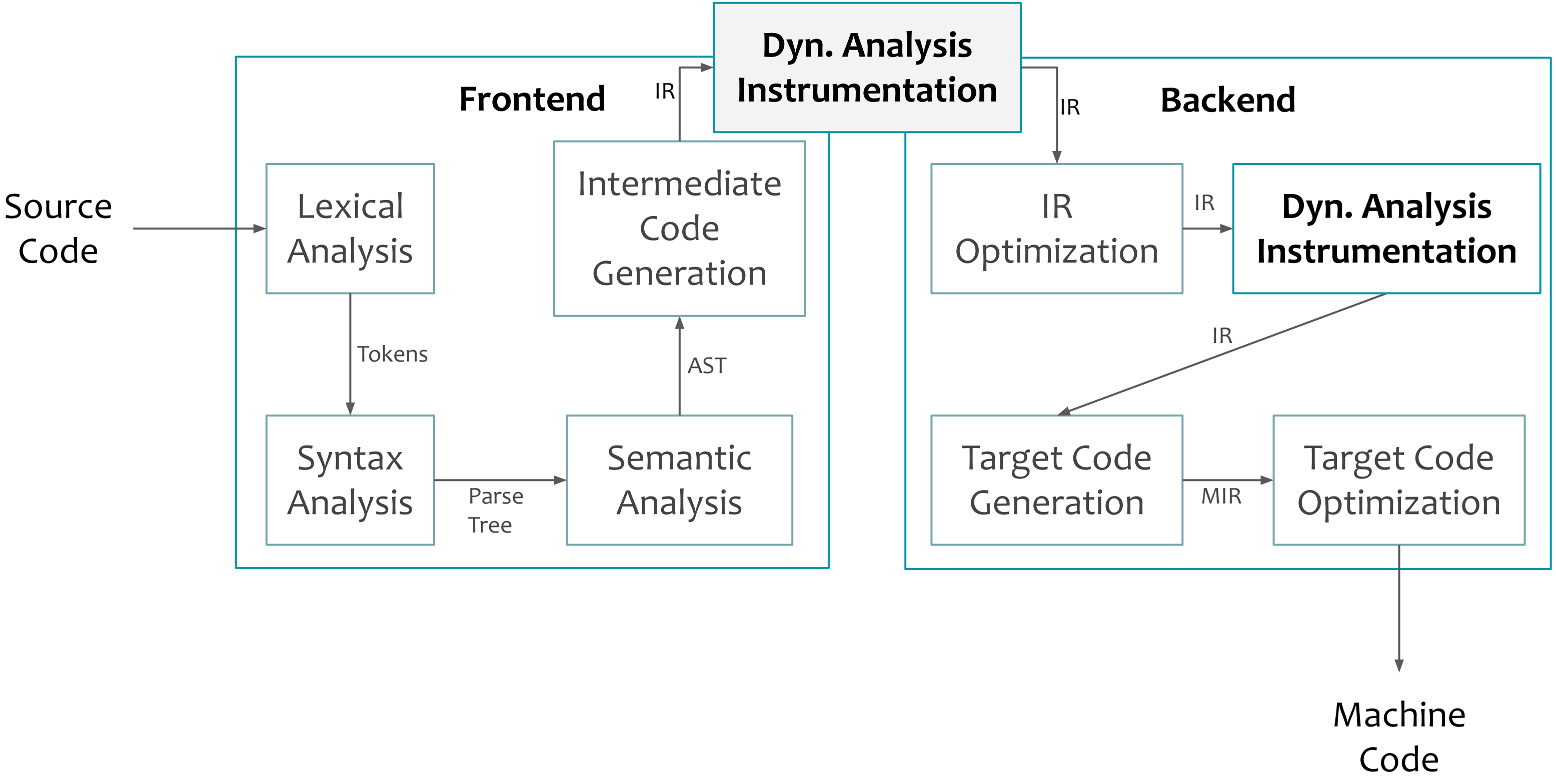


Dynamic Binary Instrumentation

- Instruments *unmodified* binary by inserting calls and/or emulating instructions.
- Usually results in very high runtime overheads.
- Unaware of source language semantics.
    - Analysis must be language-agnostic.
- Popular dynamic binary instrumentation frameworks:
    - Valgrind: valgrind.org
    - Intel Pin: www.intel.com/software/pintool
    - DynamoRIO: dynamorio.org
    - Can be used to build various dynamic program analysis.

Dynamic Binary Instrumentation

- Compiler-Assisted Instrumentation



Undefined Behavior bug categories

- Integer overflow → **UndefinedBehaviorSanitizer**

- Out-of-bounds accesses
- Heap use-after-free
- Stack use-after-return

Valgrind
AddressSanitizer

- Uses of uninitialized memory → **MemorySanitizer**

- Data Races → **ThreadSanitizer**

How to thwart security attacks?

Program hardening: "lightweight deterministic dynamic analysis", i.e. augment the original program with metadata (e.g. bounds of live objects or allowed memory regions) and insert access checks.

Software-based approaches:
compiler/runtime to transform applications to incorporate metadata management and access checking.

\+ No hardware support required

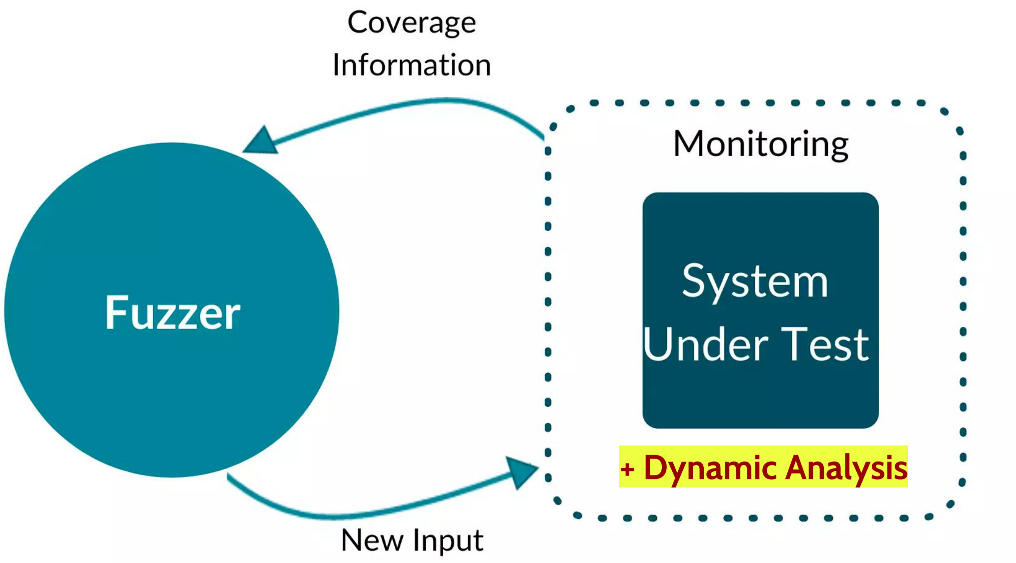
\- High-performance overheads

Hardware-based approaches:
HW support (registers/instructions) for metadata management and access checking.

\+ "Low" performance overheads

\- New hardware support required

Dynamic Analysis + Fuzzer ⇒ Find lots of bugs fast!



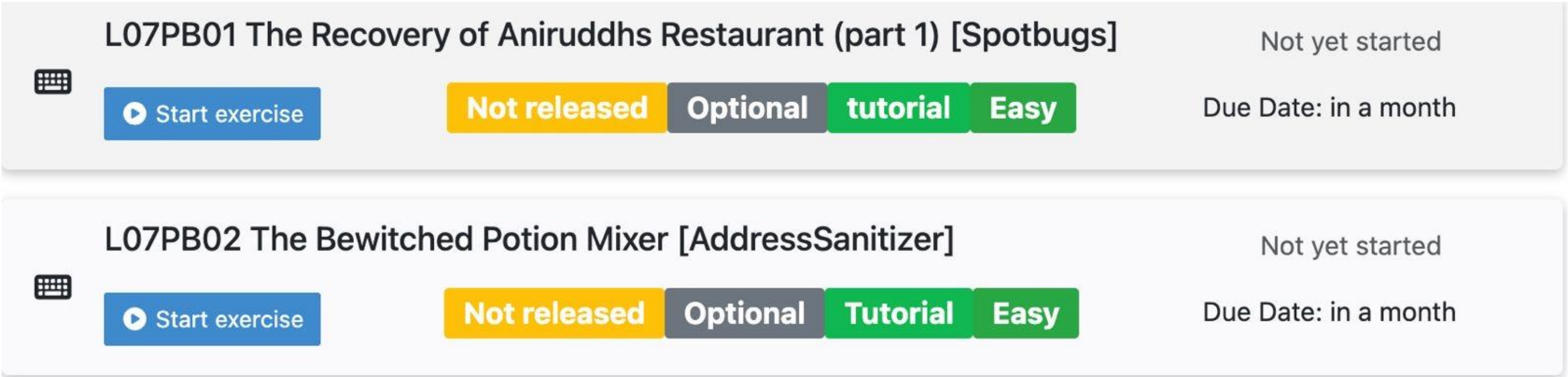
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- **Part II:** **Programming basics**

- **Part III:** Homework programming exercises (Artemis)

Programming Basics (PB) exercises

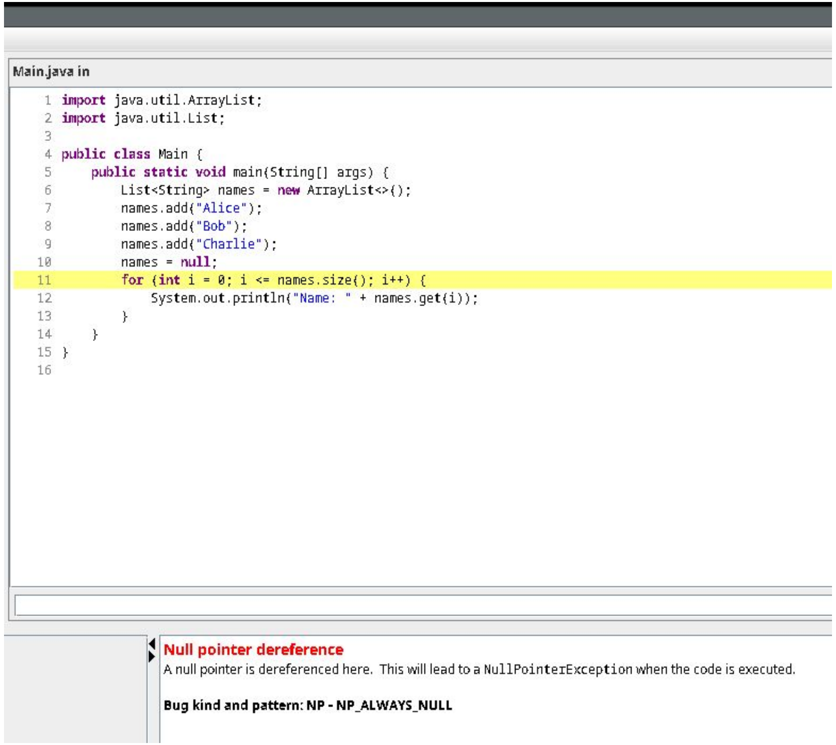


L07PB01 The Recovery of Aniruddhs Restaurant (part 1) [Spotbugs]

- Tasks:
  - Analyze a given Project and try to find problems in the code with Spotbugs
- Goals:
  - Learn how to use SpotBugs to locate and fix problems in a given Codebase

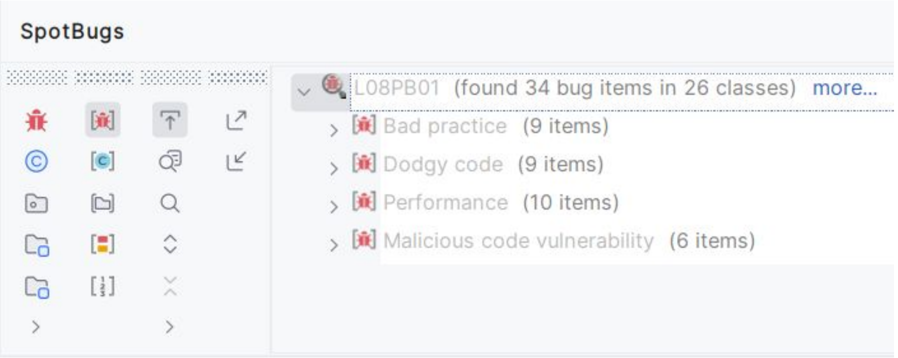
L07PB01 The Recovery of Aniruddhs Restaurant (part 1) [Spotbugs]

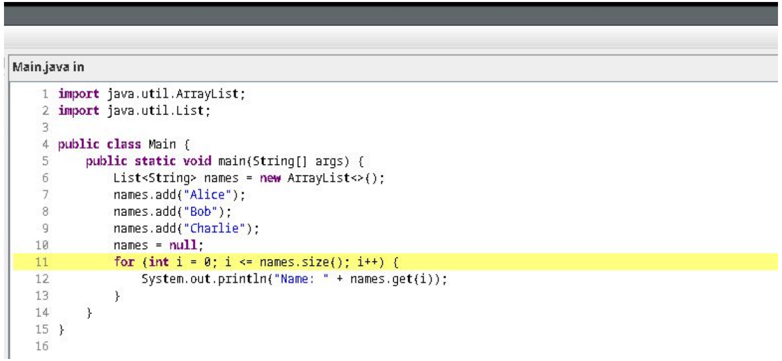
- SpotBugs is a static analysis tool for Java
- It checks Java code for common bad practices and potentially incorrect implementations
- A list of possible bugs that can be found here: https://spotbugs.readthedocs.io/en/latest/bugDescriptions.html



L07PB01 The Recovery of Aniruddhs Restaurant (part 1) [Spotbugs]

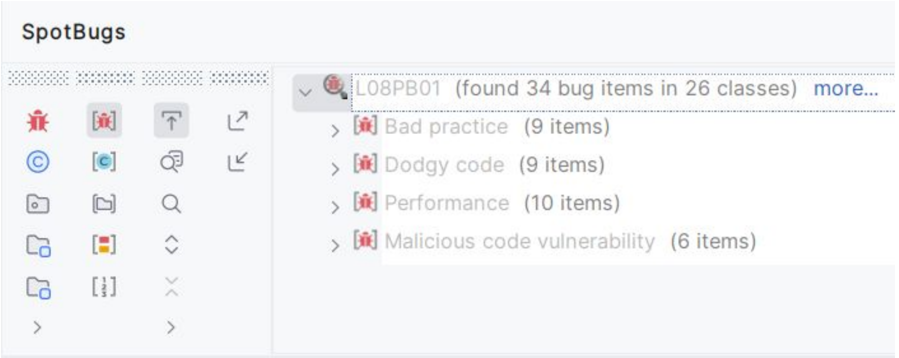
- SpotBugs can be installed via some IDEs (e.g. IntelliJ) or directly be used via its own guide (https://github.com/spotbugs/spotbugs/releases)

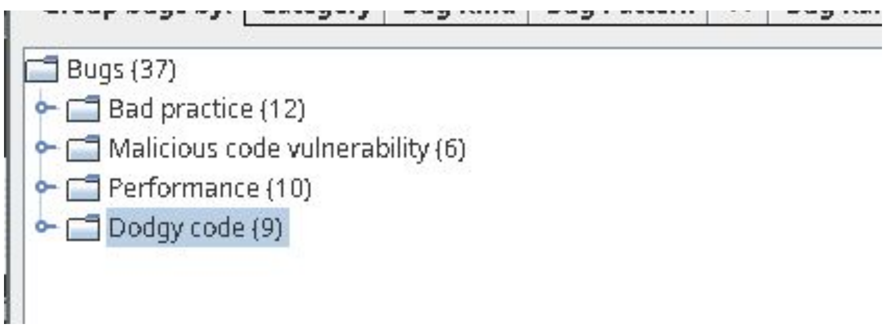




L07PB01 The Recovery of Aniruddhs Restaurant (part 1) [Spotbugs]

- Now that you have installed SpotBugs you can start with the main part of the exercise
  - Clone the repository from Artemis
  - And compile the Project (directly or with IDE of your choice)





L07PB01 The Recovery of Aniruddhs Restaurant (part 1) [Spotbugs]

- There are a multitude of Problems in the code. 34 in total
- Included Problems are:
  - String Comparison with !=

```
public void serveRice() {
    if(Table.currentTableRepresentation != Table.cleanTableRepresentation) {
        Table.cleanTable();
    }
```

  - Unread fields

```
public class Lassi extends Dishes {

    private final String name = "Lassi";
```

L07PB02 The Bewitched Potion Mixer [AddressSanitizer]

- Tasks:
  - Understanding memory errors and their consequences using AddressSanitizer for dynamic analysis
- Goals:
  - Learn how to use AddressSanitizer to locate and fix problems in a given Codebase

L07PB02 The Bewitched Potion Mixer [AddressSanitizer]

- AddressSanitizer and other sanitizers are part of compiler like gcc and clang
    - *i.e.* no explicit installation required
- By injecting additional information at compile time they enable to find bugs at runtime
- For AddressSanitizer these include:
    - Use after free
    - Buffer overflow
    - User after scope
    - Memory leaks

L07PB02 The Bewitched Potion Mixer [AddressSanitizer]

```c
void mix_initial_potion(const int first, const int second) {
    char *ingredient_one = ingredients[first];
    char *ingredient_two = ingredients[second];

    printf("Mixing ingredients %s and %s.\n", ingredient_one, ingredient_two);

    // Add a new ingredient, avoiding uninitialized memory use by initializing it
    char *volatileEssence = malloc(MAX_INGREDIENT_LENGTH);
    if (volatileEssence) {
        strncpy(volatileEssence, "Volatile Essence", MAX_INGREDIENT_LENGTH);
        volatileEssence[MAX_INGREDIENT_LENGTH - 1] = '\0';
        printf("Adding %s.\n", volatileEssence);
        free(volatileEssence);
    }
}
```

Correctly initialize volatileEssence before using it

L07PB02 The Bewitched Potion Mixer [AddressSanitizer]

```
char name[] = "Phantom Dust";
phantomDustName = (char *)malloc(MAX_INGREDIENT_LENGTH);
strncpy(phantomDustName, name, strlen(name)+1);

const char *description = "A mysterious, almost invisible stardust.";
phantomDustDescription = (char *)malloc(strlen(description) + 1);
if (phantomDustDescription) {
    strcpy(phantomDustDescription, description);
}

phantomDustInfo = malloc(sizeof (struct IngredientInfo));
phantomDustInfo->name = phantomDustName;
phantomDustInfo->description = phantomDustDescription;
```

Copy the string instead of using the local version

L07PB02 The Bewitched Potion Mixer [AddressSanitizer]

```
void handle_phantom_dust(void) {
    const struct IngredientInfo *reappearedDust = get_phantom_dust();
    printf("Ingredient Description: %s\n", reappearedDust->description);

    free(phantomDustInfo->name);
    free(phantomDustInfo->description);
    free(phantomDustInfo);
}
```

Remove second free(phantomDustInfo)

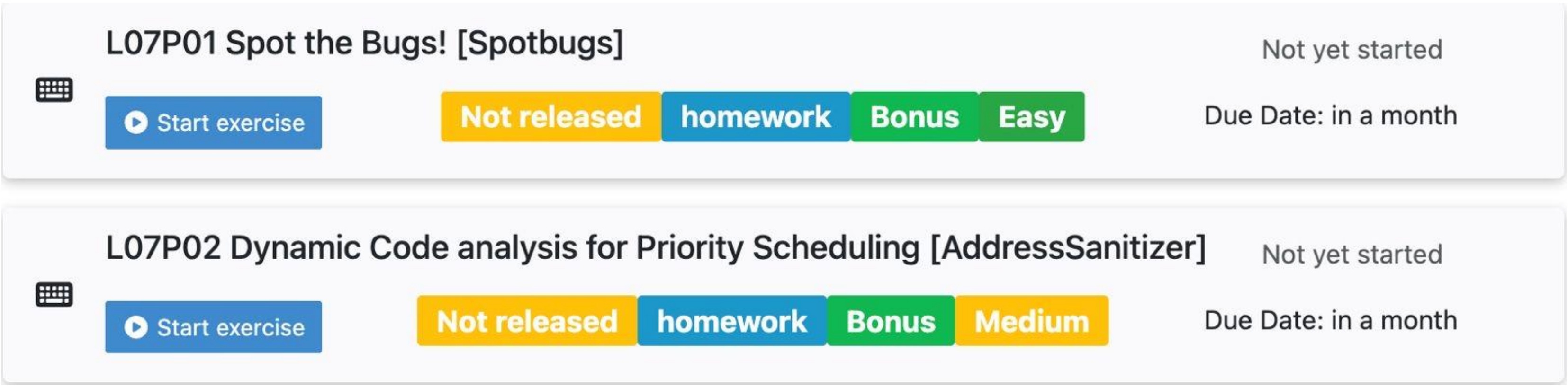
Tutorial outline

- ~~**Part I:** Lecture summary~~
  - ~~Q&A for the lecture material~~

- ~~**Part II:** Programming basics~~

- **Part III:** **Homework programming exercises (Artemis)**

Programming (P) exercises



L07P01 Spot the Bugs! [Spotbugs]

- Tasks:
  - Analyze a given Project and try to find problems in the code with Spotbugs
- Goals:
  - Learn how to use SpotBugs to locate and fix problems in a given Codebase

L07P02 Dynamic Code analysis for Priority Scheduling [AddressSanitizer]

- Tasks:
  - Find bugs in the scheduling algorithm using dynamic analysis with AddressSanitizer
- Goals:
  - Identify common memory-related issues such as Buffer Overflows, Use-After-Free and Memory Leaks

Programming Extras (PE) exercises

L07PE01 The Recovery of Aniruddhs Restaurant (part 2) [Infer]

Not yet started

Start exercise

Not released | Optional | homework | Easy

Due Date: in a month

L07PE02 Playlist Manager: Memory Debugging with MemorySanitizer [MemorySanitizer]

Not yet started

Start exercise

Not released | Optional | homework | Medium

Due Date: in a month

L07PE03 Gym Threading [ThreadSanitizer]

Not yet started

Start exercise

Not released | Optional | homework | Medium

Due Date: in a month

L07PE01 The Recovery of Aniruddhs Restaurant (part 2) [Infer]

- Tasks:
  - Analyze a given Project and try to find problems in the code with Infer
- Goals:
  - Build and Setup an environment to use infer
  - Using infer to find and fix bugs in a existing code

L07PE01 The Recovery of Aniruddhs Restaurant (part 2) [Infer]

- Infer is a static analysis tool made by Facebook
- It support multiple languages and can find different problems with them:
  - C/C++/Objective-C and Java

```
root@dbae59bf6270:/src# infer -- java Infer.java
Capturing in javac mode...
Found 1 source file to analyze in /src/infer-out
1/1 [############################################################] 100% 28.675m

Infer.java:9: error: Null Dereference
  object `s` last assigned on line 8 could be null and is dereferenced at line 9
.
   7.            int test() {
   8.                    String s = mayReturnNull(0);
   9. >                  return s.length();
  10.            }
  11.    }


Found 1 issue
          Issue Type(ISSUED_TYPE_ID): #
  Null Dereference(NULL_DEREFERENCE): 1
root@dbae59bf6270:/src#
```

L07PE02 Playlist Manager: Memory Debugging with MemorySanitizer [MemorySanitizer]

- Tasks:
  - Your task is to identify and fix any memory-related bugs in this program using MemorySanitizer
- Goals:
  - Enhance your understanding of memory-related issues in C programs and familiarize yourself with the use of MemorySanitizer, a tool for detecting memory errors

L07PE03 Gym Threading [ThreadSanitizer]

- Tasks:
  - Utilize ThreadSanitizer and synchronization mechanisms to identify and resolve bugs present in the code
- Goals:
  - Understanding the importance of analysis and optimization in concurrent programming by using ThreadSanitizer to remove memory leaks, data races, and synchronization problems

Homework programming exercises

- For more information, please check out the task descriptions on Artemis.

https://artemis.cit.tum.de/