B. Tech Project

End-Semester Evaluation

Security Analysis of Compiler Optimisations

Supervisor: Dr. Chandan Karfa

Dristiron Saikia
180101022

Outline

1. Introduction
2. Problem Statement
3. Literature Survey
4. Solution Approach
5. Efforts
6. Future Plans

What are Compiler Optimisations ?

- Reduce memory usage
- Increase performance

- Reasonable Compilation time

Example:

The compiler sees a hot path, so it alters the code guaranteeing correctness.

```c
// Code before optimizati
int secret = 0;
if(priv)
    secret = 0xFBADC0DE;
```

Code motion optimisation

```c
// After applying code m
int secret = 0xFBAC0DE;
if(!priv)
    secret = 0;
```

Why is Security Analysis in Optimisations needed ?

Example:

```c
// Code before optimizati
int secret = 0;
if(priv)
    secret = 0xFBADC0DE;
```

Code motion optimisation

```c
// After applying code mo
int secret = 0xFBAC0DE;
if(!priv)
    secret = 0;
```

The above optimisation is correct but is information leaky

Memory-safety violations

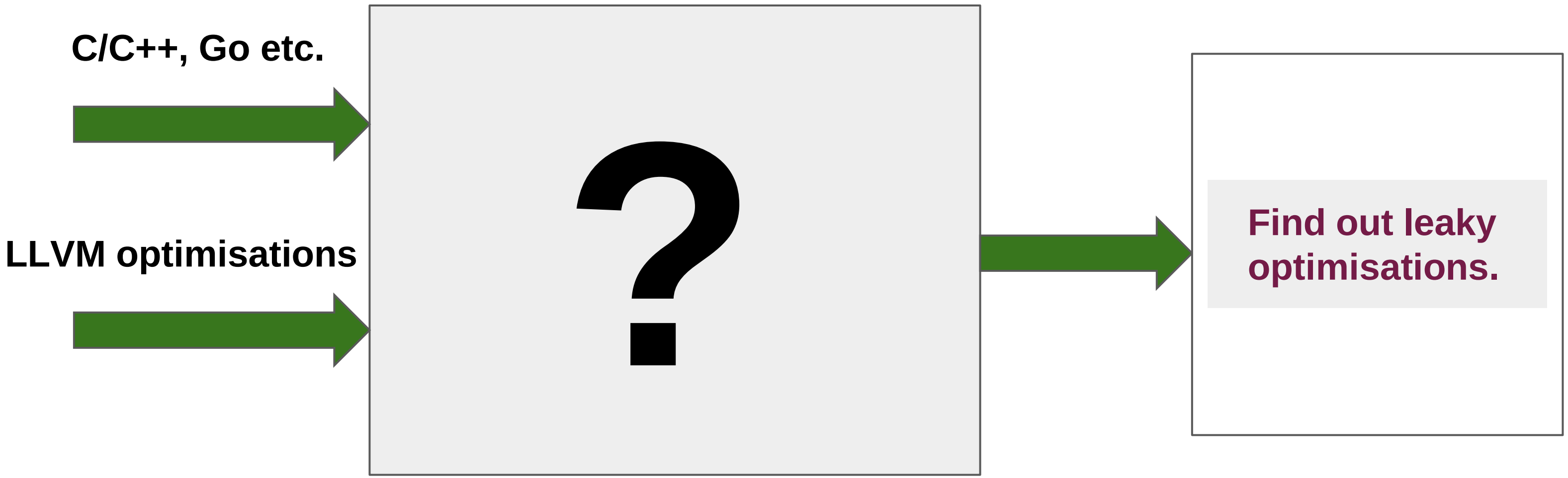
<u>Optimisation effects</u>

- Layout of stack frames
- Liveness of variables
- Timing of individual execution paths

Out-of-bound pointers

Problem Statement

Tracing information leakage at different Optimisation levels in LLVM compiler.



Literature Survey

1. Vijay D’Silva, Mathias Payer, Dawn Song, **“The Correctness-Security Gap in Compiler Optimization”**, *2015 IEEE Security and Privacy Workshops (SPW)*
2. Chaoqiang Deng, Kedar S. Namjoshi, **“Securing a compiler transformation”**, *Formal Methods in System Design, Volume 53, Issue 2, October 2018, pp 166-188*
3. Frēdēric Besson, Alexandre Dang, Thomas Jensen, **“Information-Flow Preservation in Compiler Optimisations”** *2019 IEEE 32nd Computer Security Foundations Symposium (CSF)*
4. Éléonore Goblé, **“Taint analysis for automotive safety using the LLVM compiler infrastructure”**, *Linköping University | Department of Computer and Information Science*
5. Marcelo Arroyo, Francisco Chiotta, F. Bavera, **“An user configurable clang static analyzer taint checker”**, *2016 35th International Conference of the Chilean Science Society (SCCC)*
6. Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. **Phasar: Aninter-procedural static analysis framework for c/c++**. *In Tom´a˘s Vojnarand Lijun Zhang, editors,Tools and Algorithms for the Construction and Analysis of Systems, pages 393–410, Cham, 2019. Springer International Publishing.*

Types of Information Leakage

```c
// Dead Store Elimination
char *getPWHAsh(){
    long i; char pwd[64];
    char *sha1 = (char*)malloc(41);
    // read password
    fgets(pwd, sizeof(pwd), stdin);
    // calculate sha1 of password
    ...
    //overwrite pwd in memory
    for(int i=0;i<sizeof(pwd);i++){
        pwd[i] = 0;
    }

    untrusted();

    return sha1;
}
```

Persistent State Leakage

Dead Store Elimination

```c
// Dead Store Elimination
char *getPWHAsh(){
    long i; char pwd[64];
    char *sha1 = (char*)malloc(41);
    // read password
    fgets(pwd, sizeof(pwd), stdin);
    // calculate sha1 of password
    ...
[redacted]
    untrusted();

    return sha1;
}
```

Types of Information Leakage

Signed Integer Overflow undefined.

```c
// Undefined behaviour
int *alloc(int nrelems) {
    // Potential overflow.
    int size = nrelems*sizeof(int);
    // Comparison that depends on
    // undefined behaviour.
    if (size < nrelems) {
        exit(1);
    }
    return (int*)malloc(size);
}
```

Undefined Behaviour Leakage

Compiler <u>removes overflow check</u> because undefined.

```c
// Undefined behaviour
int *alloc(int nrelems) {
    // Potential overflow.
    int size = nrelems*sizeof(int);
    // Comparison that depends on
[redacted]
    return (int*)malloc(size);
}
```

Types of Information Leakage

```
int crypt(int *k){
int key = 0;
if(k[0] == 0xC0DE){
      key = k[0]*15+3;
      key += k[1]*15+3;
      key += k[2]*15+3;
}     else{
      key = 2*15+3;
      key += 2*15+3;
      key += 2*15+3;
}
return key;
}
```

Side Channel Leakage

Common Subexpression Elimination

```
// CSE
int crypt(int *k){
int key = 0;
if(k[0] == 0xC0DE){
      key = k[0]*15+3;
      key += k[1]*15+3;
      key += k[2]*15+3;
}     else{
      //replaced by
      tmp = 2*15+3;
      key = 3*tmp;
}
return key;
}
```

Architecture of Optimising Compiler

- Front End (Syntactic and Semantic Analysis)

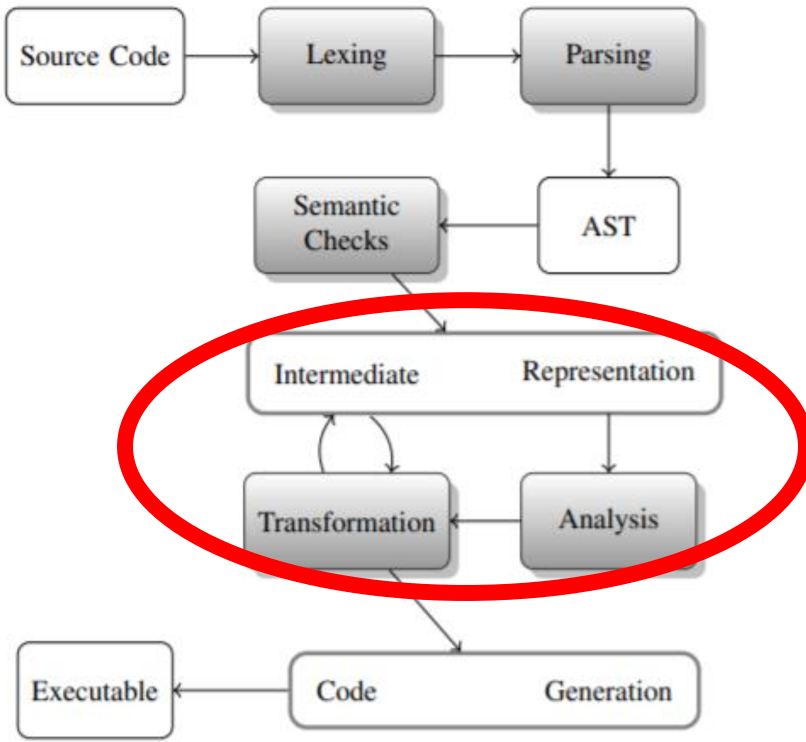
  C/C++ code.

- Middle End (**Code Optimisation**)

  Intermediate Representation.

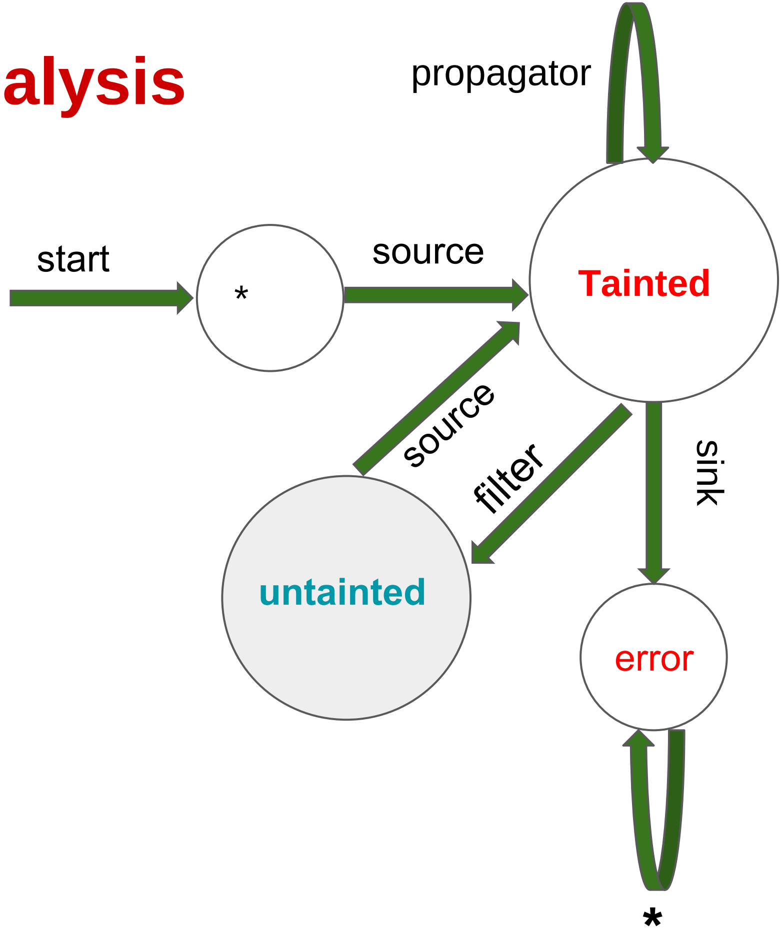
- Back End (Machine Code Generation)

  Executable file.



Taint Analysis

- **Source**: Generate tainted data
- **Propagator**: Propagate or transform tainted data into tainted data
- **Filter**: Checks if data is safe
- **Sink**: Critical operation that consume the data



State transition system for taint analysis

Low level Virtual Machine (LLVM)

Transformation Pass

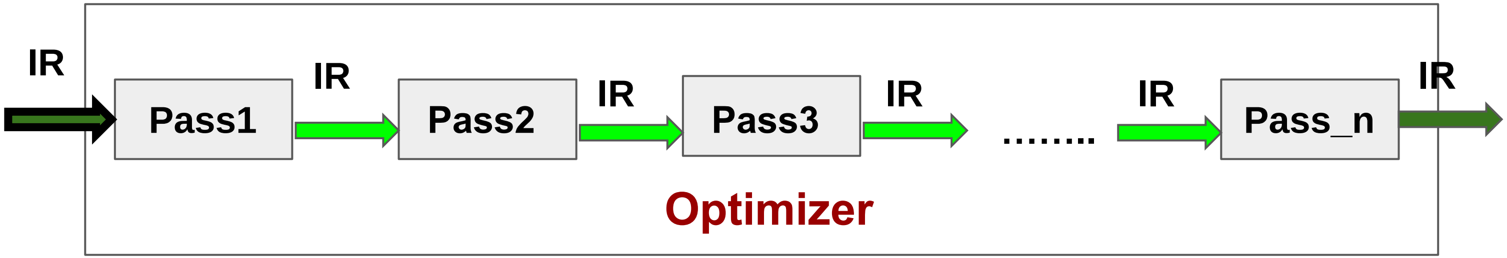
1. Modifies the IR.
2. To optimise the resulting IR.
3. Inherits from *PassInfoMixin<PassT>*

LLVM Pass Manager

1. Schedules passes in a specific order.
2. Caches Analysis results.

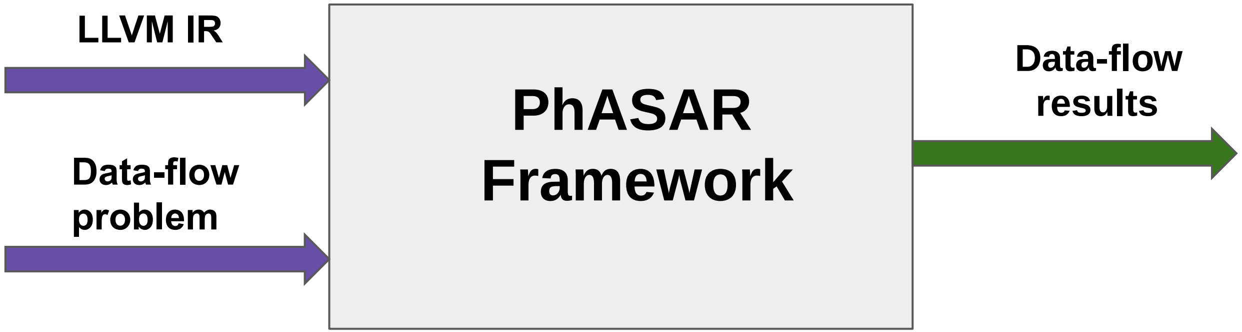
Analysis Pass

1. Makes no change to IR.
2. To extract information for static analysis.
3. Inherits from *AnalysisInfoMixin<AnalysisT>*



PhASAR

- **LLVM-based** static analysis framework.
- Solves arbitrary data-flow problems in **LLVM-IR**.
- Inbuilt context sensitive **IFDS solver** for data-flow analysis.



Solution Approach

What we want ?

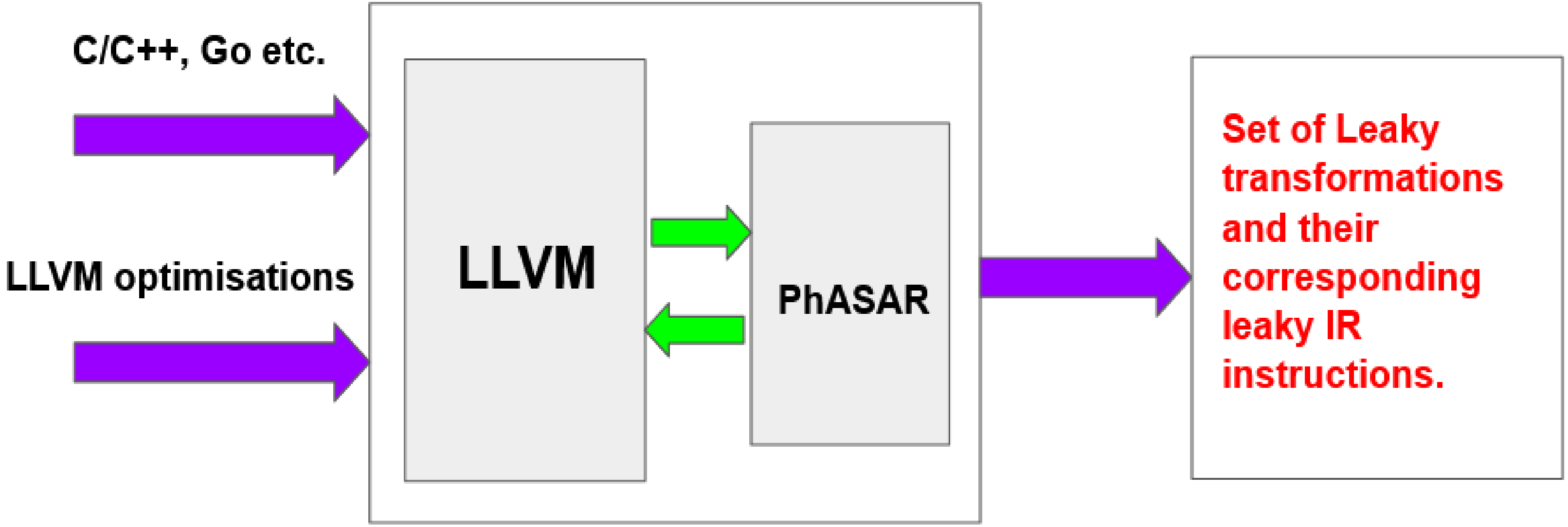
We want an information leakage tracing system for every existing optimisations in LLVM and flexible to accommodate any future optimisations in LLVM.

How to achieve this ?

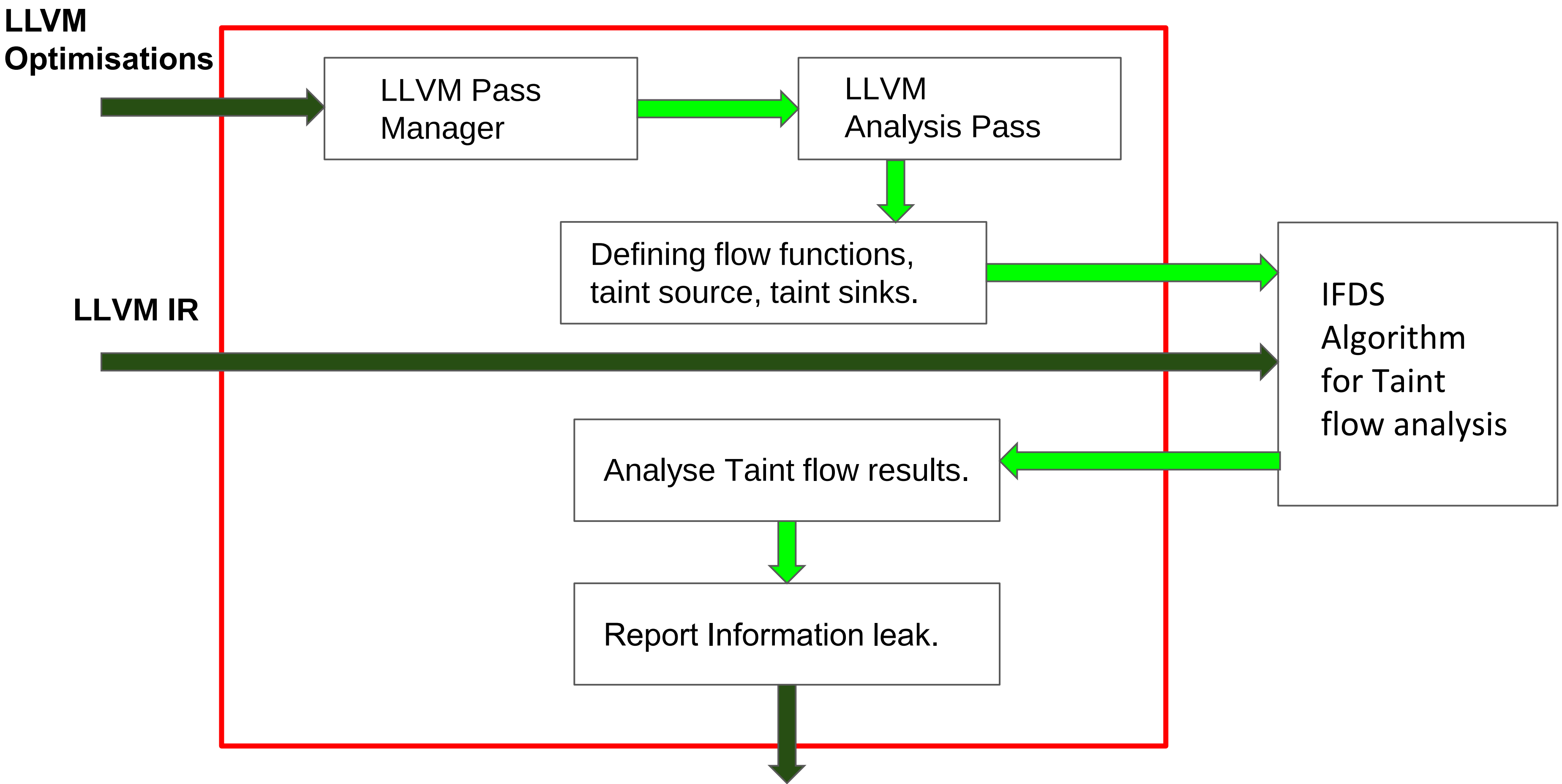
The solution can be divided into two phases

- Find a way to check for information leakage for a given program. PhASAR
- Integrate the information leakage checker into the LLVM compilation process. LLVM Pass

High Level Design



Design Details



LLVM Analysis Pass

Implement the run function

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
struct Dristiron : public FunctionPass {
    static char ID;

    Dristiron() : FunctionPass(ID) {}

    bool runOnFunction(Function &F) override {
        errs() << "Dristiron: ";
        errs().write_escaped(F.getName()) << '\n';
        return false;
    }
}; // end of struct Dristiron
}  // end of anonymous namespace

char Dristiron::ID = 0;

static RegisterPass<Dristiron> X("Dristiron", "Dristiron Pass",
                                 false /* Only looks at CFG */,
                                 false /* Analysis Pass */);
```

Inherit from Base class.

Register the LLVM Pass

Defining flow functions

```cpp
TaintAnalysisProblem(LLVMBasedICFG &icfg, const LLVMTypeHierarchy &th,
                     const ProjectIRDB &irdb, TaintSensitiveFunctions TSF,
                     std::vector<std::string> EntryPoints = {"main"})
    : LLVMDefaultIFDSTabulationProblem(icfg, th, irdb),
      SourceSinkFunctions(TSF),
      EntryPoints(EntryPoints) {
  TaintAnalysisProblem::zerovalue = createZeroValue();
}

virtual ~TaintAnalysisProblem() = default;

std::shared_ptr<FlowFunction<const llvm::Value *>> getNormalFlowFunction(
    const llvm::Instruction *curr, const llvm::Instruction *succ) override {
  // TODO: implement
  return Identity<const llvm::Value *>::getInstance();
}

std::shared_ptr<FlowFunction<const llvm::Value *>> getCallFlowFunction(
    const llvm::Instruction *callStmt,
    const llvm::Function *destMthd) override {
  // TODO: implement
  return Identity<const llvm::Value *>::getInstance();
}

std::shared_ptr<FlowFunction<const llvm::Value *>> getRetFlowFunction(
    const llvm::Instruction *callSite, const llvm::Function *calleeMthd,
    const llvm::Instruction *exitStmt,
    const llvm::Instruction *retSite) override {
  // TODO: implement
  return Identity<const llvm::Value *>::getInstance();
}

std::shared_ptr<FlowFunction<const llvm::Value *>> getCallToRetFlowFunction(
    const llvm::Instruction *callSite, const llvm::Instruction *retSite,
    std::set<const llvm::Function *> callees) override {
  // TODO: implement
  return Identity<const llvm::Value *>::getInstance();
}
```

Constructor

Intra-procedural flow

Parameter mapping

Flow along call-sites

Return values

IFDS Report

```
  void printIFDSReport(
      std::ostream &os,
      SolverResults<const llvm::Instruction *, const llvm::Value *,
                    BinaryDomain> &SR) override {
    os << "\n----- Found the following leaks -----\n";
    if (Leaks.empty()) {
      os << "No leaks found!\n";
    } else {
      for (auto Leak : Leaks) {
        os << "At instruction\nIR  : " << llvmIRToString(Leak.first) << '\n';
        os << llvmValueToSrc(Leak.first);
        os << "\n\nLeak(s):\n";
        for (auto LeakedValue : Leak.second) {
          os << "IR  : ";
          // Get the actual leaked alloca instruction if possible
          if (auto Load = llvm::dyn_cast<llvm::LoadInst>(LeakedValue)) {
            os << llvmIRToString(Load->getPointerOperand()) << '\n'
               << llvmValueToSrc(Load->getPointerOperand()) << '\n';

          } else {
            os << llvmIRToString(LeakedValue) << '\n'
               << llvmValueToSrc(LeakedValue) << '\n';
          }
        }
        os << "-------------------\n";
      }
    }
  }
};
```

Efforts

- LLVM codebase

```
formal@formal-CELSIUS-R940power:~$ du -hs ./llvm-project/
59G      ./llvm-project/
```

- Adding LLVM analysis pass.

```
formal@formal-CELSIUS-R940power:~/llvm-project/build$
formal@formal-CELSIUS-R940power:~/llvm-project/build$ cat ../../Dristiron/hello.c
#include<stdio.h>
void Dristiron(){
}
int main(int argc, char** argv){
  printf("Hello World\n");
}
formal@formal-CELSIUS-R940power:~/llvm-project/build$ opt -load ./lib/LLVMHello.so -hello < ../../Dristiron/hello.ll > /dev/null
Hello: Dristiron
Hello: main
formal@formal-CELSIUS-R940power:~/llvm-project/build$
```

Future Plans

- Implement the mentioned design.
- Running units tests.
- Benchmark the proposed system.
- Test on compilation time and look for efficiency.
- Develop a good documentation so that it could be easily used.

Thank You