Assignment 1 - Frontend

Deadline: See Canvas

The goal of the course assignments is to give you hands-on experience with all parts of a modern compiler. To keep things simple, in this course you will work on a compiler for a basic subset of C. At the end of the course, you should have a general idea how most parts of a compiler work and where to start if you want to make your own.

Rules

To ensure you receive a grade that reflects your understanding of the course material and the effort you put into the assignments, you have to follow the rules below. Breaking any of these rules may lead to you receiving 0 points for a given task/assignment and potentially being referred to the Examination Board.

1. The code you submit in your assignment has to be written by you.
2. Do not share your (partial) solutions with other students.
3. You can discuss the assignments with other students, but you should not share how you solved an assignment.
    - E.g., you can discuss whether a certain FenneC program is valid, what the program's output should be, or which parts of the program could be optimized.
4. Do not hardcode the full/partial solution for certain tasks into your assignment.
    - E.g., you can not submit a list of correct outputs for each test input and then print them out. Your assignment should behave as if it was a proper compiler.
    - In general, any logic in the form of `if (input_file_name == "...") special_logic();` will lead to a point reduction.
5. Do not try to cheat the grading system by e.g., modifying the grading script from within your compiler or faking any output.
6. You are explicitly allowed to look at any part of the LLVM source code. This includes the code that solves the same/similar problems as in the assignments.

Grading

All assignments are automatically graded using the provided grading script. You should run this grading script to check if your solution is correct. The point count printed by the grading script will match the actual points you receive when handing in your solution. The only exception to this is if you broke one of the rules above.

Environment

This course provides a docker container for this and nearly all following assignments. You have to get your lab code running within that docker container as this is how the grading system will run your code. The docker container also comes with a hand-in script that you have to use to create the hand-in zip files.

Frontend

This assignment lets you implement a *frontend* for a language called *FenneC*. This language is a small subset of C, meaning that all FenneC programs are also valid C programs. However, not all valid C programs are valid FenneC programs. The course resources provide a grammar file which describes the syntactic subset of C you have to support. All programs that can be expressed with this grammar behave identically to their C counterparts.

The purpose of a compiler frontend is to parse source code, verify the syntactic and semantic rules of the programming language, and then generate code suitable for the middle-end of the compiler.
Our frontend should parse FenneC source code, enforce its syntax and type system, and then generate LLVM IR from it. The tasks below provide you with more details on what exactly you are expected to do.

Note: We suggest you write your frontend in Python using the referenced tools/libraries so TAs can properly help you. However, you are free to use any other language and its equivalent tools for this assignment assuming you can run it on the provided docker container. You should reach out to the TAs or email coco-support@vusec.net before you start working with a different language.

Task 1: Lexing and parsing (1pt)

The first task is to turn FenneC source code into an abstract syntax tree (AST). This can be done using the appropriate tools, e.g., a parser generator. We provide you with the appropriate grammar file for the *Lark* parser generator as part of the course material. You can use this grammar file and Lark to automatically generate a parser. You are also allowed to write a recursive-descent parser by hand.

If you find a lexer/parsing error, your compiler frontend should return the exit code **`2`** from your frontend. In Python, this is done by calling **`sys.exit(2)`**.

Grading:
Run **`./grade --assignment 1.1`** which will print the points you got for this task.

Task 2: Type checking and other checks (4pts)

Using your parser from task 1, you now have to enforce the type system of FenneC. This is done by traversing each node in your AST and then checking the node-specific type constraints. For example, a division operation node (e.g., `4/2`) requires its two operands to be of the same integer type.

If you find a typing error, your compiler frontend should return the exit code **`2`** from your frontend.

Grading:

Run **`./grade --assignment 1.2`** which will print the points you got for this task.

Hints:

- Think about the order in which you need to traverse the AST to properly handle nested nodes.
- FenneC allows shadowing of variables.
- Ideally you would do all type checking in a dedicated type checking phase of the compiler that is independent of the code generation. However, to keep things simple, you will not get a point reduction for doing type checking while generating LLVM IR (see the next task).
- You can assume that `Print/PrintU` are built-in functions you hard-code in your compiler. You don't need to properly handle the FenneC.h include.
- There are no implicit (or explicit) type conversions allowed in FenneC. E.g., you can not multiply an `int` and an `unsigned` expression (such as **`1 * 1U`**).

Task 3: Generating LLVM IR (5pts)

The final task is to produce LLVM IR from the parsed AST. LLVM IR is effectively another programming language that is designed to be a middle-end for a modern compiler. Your job is to translate your parsed C program into an *equivalent* LLVM IR program.

Hints:

- You do not need to generate efficient LLVM IR from your frontend. For example, you can simply model all local variables as `alloca`'s and encode read/writing that variable as `load/store`. This is also what real compiler frontends do.
- There are no function/global declarations in FenneC. This means that every function/global is defined in the source before it is first called. You can rely on this to simplify your implementation.
- You can emit hard-coded function declarations for `Print/PrintU`. You can find their source in `framework/fennec/`. The way you declare a function in LLVM (and the Python wrapper) is to create the `Function` but not add any BasicBlocks to it.
- You are allowed to check what Clang (LLVM's own C frontend) is doing. You can also use a website such as godbolt for this. See the example here: https://godbolt.org/z/57n9P7rco .

Grading:
Run **`./grade --assignment 1.3`** which will print the points you got for this task.

Assignment 2 - Optimizations

Deadline: See Canvas

In the previous assignment, we parsed source code and translated it into LLVM IR. This allows us to lower it into machine code and execute it. This assignment is about another reason for why we generated LLVM IR: Optimizations. Your task in this assignment is to optimize a set of existing programs with the help of the compiler.

Grading

This assignment has no fixed point count associated with each task. Instead, there are a total of 10 points to gain in the whole assignment. Your point count is calculated on how fast the programs run after your compiler passes optimized them. **The faster these programs run after optimization, the more points you get.** The tasks we give you are designed to give you 10 points assuming you implement them correctly.

The grading script comes with several programs that we each assigned an 'optimal' runtime. If you can optimize each program to stay below this runtime, you will receive 10 points. If your optimized program runs as slow as the unoptimized version (or slower), you will receive 0 points. We linearly interpolate the points you receive based on how close your performance comes to the optimal runtime for all programs combined. The actual score is determined by the average improvements of all tests.

Note: You will get **2 points less** for **each** program that produces a different output after optimization. In other words, you have to preserve the behavior of the unoptimized program. We also added several tests in the 'benchmarks/csmith' folder that are not benchmarked for performance, but they are used to check your program for correctness. If one of those tests fails, your optimization is incorrect and the best course of action is to revert your breaking changes to your pass. Do **NOT** try to 'fix' broken passes later once you see csmith tests failing, as this will most likely not be easy.

As before, the grading script will tell you your expected point count. It will also give you various useful statistics about which programs have the largest remaining optimization potential and which programs were incorrectly compiled.

The respective grading script invocation is `./grade --assignment 2`.

The Benchmarking In-Order Machine Emulator (BIOME)

Because every student has different hardware, this assignment does not measure the actual CPU/wall-time of programs. Instead, we run each program in an emulator that ensures a level playing field for all students as well as consistent grading. This emulator is called BIOME and is part of the course framework.

BIOME runs your programs and outputs how many simulated 'CPU cycles' the program used. Each LLVM instruction is given a fixed cycle count that is summed up for the total

program execution. In a real machine, these LLVM instructions would have been lowered to actual hardware instruction during compilation, but we stick to counting the LLVM instructions to keep things simple and independent of your actual hardware.

In addition to a cost for execution, BIOME also simulates a (very primitive) instruction cache that only fits a fixed amount of 16 (LLVM) instructions. Each instruction has to be loaded into the instruction cache before being executed. If the instruction is not in the cache already, then the oldest executed instruction is evicted and the program slows down for 40 cycles. The cache is never flushed/emptied otherwise and persists across calls.

You can find the cycle cost for each instruction in the file:
`./framework/BIOME/CostModel.h` file.

Note: Not all LLVM instructions have a cost in this file. If an instruction is not listed, it means it is unsupported by our made-up CPU and you **can not use it**.

Setup

The `assign2` folder in the course contains several .cpp files that contain some of the boilerplate code for your compiler passes. You have to implement the different optimizations in the respective source files. You can also implement further optimizations, but that is not necessary to achieve 10 points in this assignment.

Optimization Pipeline

The `optimize.sh` script is called by the grading script to optimize each program. It contains example invocations for the no-op template passes you have to fill out. The result from one optimization pass is passed towards the next one. The order in which the passes are run matches the order in the file. E.g., the code below would run the LICM pass, then forward the optimized IR to the ADCE pass.

```
run_pass coco-licm
run_pass coco-adce
```

You can (and **should**) change the pass order in `optimize.sh` in which the passes run and how often you want to run them. There is no point deduction for running passes too often. However, it will slow down the compilation process and make your own lab work more tedious.

The `optimize.sh` script emulates what a real optimization pipeline in a compiler looks like. However, unlike a real optimization pipeline, the one in this assignment can be static (i.e., pick a list of passes and put them in the file). A real compiler could also dynamically schedule further optimizations.

Task 1: Aggressive Dead Code Elimination (ADCE)

Dead code elimination removes all code that is not reachable or unnecessary from a program. For example, an assignment of a variable that is never used can be eliminated without changing the semantics of a program.
Aggressive dead code elimination (ADCE) achieves this by assuming a computation is dead until it is proven to be live. A benefit of ADCE is that it allows the compiler to prove that an instruction is dead even if they are in a cycle in the def-use graph.

The algorithm starts by going through all basic blocks in depth-first order. This way we ensure we do not visit unreachable blocks. The algorithm then marks instructions as (trivially) alive (if it has side effects) or trivially dead (if it has no side effects and has no uses). We then mark the use-def chains of every alive instruction as alive as well, and finally we remove every instruction not marked as alive.

Since this is the first time you will be writing an LLVM pass, we list the exact steps, hints and the pseudocode of the algorithm below. While implementing this pass, try to think of why this approach works!

Approach

- Visit each Basic Block in depth-first order. This ensures we do not visit trivially dead blocks.
- Gather all instructions that are trivially live in a worklist and mark them as live. Remove trivially dead instructions.
- Propagate liveness to all (instruction) operands of the items in the worklist. If an operand of an element in your worklist is an instruction, mark it as live and add it to the worklist. Skip cases where you already marked the instruction as live.
- Drop all references to instructions not marked live, and then erase them.

Hints

- An instruction is trivially live if it may have side-effects (check with `Instruction::mayHaveSideEffects()`, or if it is a terminator, a volatile load, a store}, or a call.
- An instruction is trivially dead if it has no uses (use `I.use_empty()`).
- For the worklist you can use LLVM's built-in SmallVector type.
- You can iterate through basic blocks in depth-first order using `depth_first` or `depth_first_ext`. The latter stores reachable blocks in the `df_iterator_default_set<BasicBlock*>` passed as its second argument.
- Use a set (or `DenseMap`) for keeping track of live instructions.

Algorithm

```
// Initial pass to mark trivially live and trivially
// dead instructions. Perform this pass in depth-first
```

```
// order on the CFG so that we never visit blocks that
// are unreachable: those are trivially dead.
LiveSet = emptySet;
for (each BB in F in depth-first order)
    for (each instruction I in BB)
        if (isTriviallyLive(I))
            markLive(I)
        else if (I.use_empty())
            remove I from BB;
// Worklist to find new live instructions
while (WorkList is not empty) {
    I = get instruction at head of work list;
    if (basic block containing I is reachable)
        for (all operands op of I)
            if (operand op is an instruction)
                markLive(op)
}
// Delete all instructions not in LiveSet. Since you
// may be deleting multiple instructions that may be
// in a def-use cycle, you must call I.dropAllReferences()
// on all of them before deleting any of them
// because you cannot delete a Value that has users.
for (each BB in F in any order)
    if (BB is reachable)
        for (each non-live instruction I in BB)
            I.dropAllReferences();
for (each BB in F in any order)
    if (BB is reachable)
        for (each non-live instruction I in BB)
            erase I from BB;
```

Task 2: Combining Instructions

Your first task is to find sets of instructions that can be combined into a set of cheaper instructions. All optimizations you implement here can be done as peephole optimizations. In other words, you traverse your program and during every traversal step, you inspect just a small subset of the program (the *peephole*) to search for instructions to combine. You can limit yourself to just inspecting each instruction and its direct arguments for this task.

There are three groups of optimizations that you should implement here:

- **Constant propagation:** Computes expression involving only constants at compile time and replaces them with the result. For example, the compiler can replace `unsigned x = 1 * 2 - 4` with simply `unsigned x = 0`.
- **Algebraic simplification:** This involves removing operations that are algebraic identities, such as simplifying `x * 1` to just `x`.

- **Reduction-in-strength:** Find a cheaper set of instructions that behave the same as the original instructions. For example, `x * 2` is faster on BIOME when expressed as `` `x + x` `` (as the cycle cost for addition is lower).

Task 3: Loop Invariant Code Motion

Large part of a program's workload usually occurs within loops. Often there are computations within the loop body that always produce the same results (i.e, they are loop-invariant). In this task you implement loop-invariant code motion (LICM), which *hoists* (moves out) computations from a loop without changing the semantics of the program, usually saving a lot of unnecessary computation.

For example, the following loop:

```
for (int i = 0; i < n; i++) {
  x = y + z;
  a[i] = 6 * i + x * x;
}
```

can be rewritten to reduce the total number of computations needed as follows:

```
x = y + z;
t1 = x * x;
for (int i = 0; i < n; i++) {
    a[i] = 6 * i + t1;
}
```

Your task is to hoist loop-invariant computations out of loops. To keep things simple we focus on register-to-register computations (i.e., we ignore computations that read or write to memory). We focus only on computations that are actually used within the loop. You can assume that the -loop-simplify pass has been run before your pass, so that all loops have a proper preheader.

Approach

- Obtain the current loop header.
- Get every BasicBlock dominated by the current loop header. Use the DominatorTree class to find out which blocks are dominated.
- Find the blocks that are within the current loop (i.e., not a sub-loop).
- Move every loop-invariant instruction of these blocks to the preheader of the loop.

We leave it up to you to find out which instructions are loop-invariant. In general, side-effects mean that variables are not loop-invariant (you can check for that via the LLVM function called `isSafeToSpeculativelyExecute`). Also, dependencies on non-constants or values produced in the loop are also (usually) not loop-invariant.

Task 4: Inlining Calls

Inlining replaces a function call in the IR with the body of the called function (and all arguments substituted). Unlike the previous optimizations in this assignment, inlining a function call does not always make your program faster. It only guarantees to eliminate the cost of the call itself and can enable other optimizations. However, it might increase your code size and slow down the program by causing additional pressure on the instruction cache.

There is no simple algorithm that can decide if inlining a specific is beneficial (the problem is NP-hard), so in this assignment you have to come up with your own heuristic.

Hints:

- The heuristic that gives you a good grade is not very complicated.
- You can only consider direct calls for this assignment. I.e., you don't need to inline calls using function pointers for an optimal score.

FAQ

Hi all, some FAQ and updates regarding assignment 2. I'll update them when I get more questions, so maybe check this page from time to time.

How do I know where the cycles are lost?

There is a profiler in the framework (and part of the simulator). If you run

```
PROFILE_CODE=1 ./grade -a 2
```

it will show you a heat map of where cycles are spent for each program. I also enabled it for the submissions on CodeGrade (even though you don't have color output there).

Do we need to optimize the most optimal program to get a 10?

No. The 10pt cycle count is based on a (suboptimal) solution. You can outperform it in several ways (this includes also implementing other minor peephole optimizations)

Why does the grading script round down?

The raw score is just interpolated between unoptimized code and the optimized code from the incomplete solution. The rounding and scoring is picked so there is an appropriate grading curve.

I got a cool inliner heuristic but it produces terrible performance. Why?

You most likely inline something that you shouldn't. And you probably do this because your heuristic is wrong. Note that you shouldn't just copy an heuristic from the internet (which most likely will not work as it assumes different optimizations and code cache structure).

Can we get any tips for the inliner heuristic?

- You probably have to inspect the contents of a function to make a good heuristic. This goes beyond just counting the instructions in it.
- Inspecting the call site(s) you are inlining can also help in some cases.
- Basing your heuristic on whether the call site is in a loop (and how big that loop is) can help. Note that loops can be very slow if they don't fully fit into the I-Cache.

We also have office hours on purpose. While we can't tell you a good heuristic, the TAs can help you with figuring out why your heuristic is bad.

My inliner solution slows down the code so much I get a 0 (even though I have other passes)?

Just disable the inliner in the optimize.sh pipeline and submit like that. Or remove your inliner implementation.

I noticed that loop-simplify is not run before our pass?

The loops you have to optimize should already be in the loop-simplify format (one preheader, etc.). If they are not, you don't have to optimize them and can just skip them. You might encounter those in the csmith/ tests (which are not benchmarked).

Can I call a certain function from our pass or are there functions that are off-limits?

In general, anything that does a significant part of the job of your pass is off-limits. If the function sounds very specific to what you're supposed to implement, it's most likely off-limits. If you're unsure whether you can use a function, ask coco-support@vusec.net . Note that this check isn't automated yet, so when I manually check your submissions I'll try to be not too strict.

Assignment 3 - Security

Deadline: See Canvas

By default, C programs use manual memory management and rely on the programmer to perform manual bounds checking for buffers and free memory after its last use. When done incorrectly, the resulting bugs are often subtle and have no negative effects until they start being exploited by an attacker.

In reaction to this, various memory error detection tools such as *valgrind* or *AddressSanitizer* have been developed. These tools *instrument* a target program so these subtle errors become visible and are reported in a deterministic way. Instrumentation means in this context to add code to the program that performs additional logic such as error checking. However, the original program logic is preserved and still produces the same results as without instrumentation (assuming the program is free of errors).

In this assignment, you will develop your own LLVM-based tool that can catch both spatial errors (i.e., buffer under/overflows) as well as temporal memory errors (use-after-free errors).

Task 1: Spatial Heap Safety (4pts)

The first task is to protect each program against under/overflows that affect buffers on the heap. For the tests in this task (and this task only) you can assume that all vulnerable buffers are located on the heap (i.e., they are created via `malloc`).

While we don't require you to use a specific implementation, the recommended solution is to create your own `malloc`/`free` functions which track all heap allocations. Then, replace all existing calls to the custom `malloc`/`free` functions you created. You should use the `assign3/Runtime.cpp` file to implement your custom malloc and free and `assign3/Sanitizer.cpp` to implement your pass.

Your custom `malloc`/`free` functions should allocate additional space around each memory buffer. This additional space is considered not valid for any memory access and is usually called a *red zone*. Note that the program should still get a valid memory buffer of the requested size when calling malloc.

The next step is to instrument (i.e., prefix with additional instruction) all memory accesses to call a new function that checks if the accessed address is valid. If a memory access is detected to read/write memory in a red zone, you should abort the program with exit code 66 (e.g., by calling `exit(66)`). You can implement this check in `Runtime.cpp`, but be aware that a real tool would generate LLVM IR for the check for performance reasons.

Grading:
Run **`./grade --assignment 3.1`** which will print the points you got for this task.

Hints:

- All out of bounds memory accesses in this task (and the ones below) are within 32 bytes of the original buffer. You don't need to detect invalid accesses beyond these 32 bytes.
- You do not need to detect when pointers go out of bounds unless they are used to access memory. I.e., you do not need to report when pointer arithmetic causes a pointer to go out of bounds unless that pointer is dereferenced.
- Your custom `malloc` should have a different name than just `malloc` to avoid linking issues (same for `free`). You can declare a function with the same name in your pass and generate a CallInst to it to call the respective C++ function in your runtime.
- You can get the byte size of a type with `ModuleVar->getDataLayout().getTypeAllocSize(type).getFixedValue();`

Task 2: Spatial Stack Safety (3pts)

There are also vulnerable buffers on the stack that could underflow or overflow. In this task you have to check those too for potential under/overflows. You shouldreturn exit code 66 (as above) when you detect an under/overflow.

The general approach here is to replace all `AllocaInst` instances in your program. AllocaInst is in the case of Clang/LLVM the representation of a stack buffer. The replacement should be a larger AllocaInst that reserves a red zone before/after the actual buffer. Note that you should not create several AllocaInst instances, but instead one large one that contains both the red zone as well as the original buffer.

Hints:

- You can reuse the utility functions of the previous task for this one.
- Note that AllocaInst has both a type and also an element size that you need to consider to get the stack buffer size (`A->getArraySize()).`
- There is no equivalent of a `free` call for stack buffers. They still have a lifetime though (think about when the buffer is no longer used) and their redzone might be valid memory in subsequent function call.

Grading:
Run **`./grade --assignment 3.2`** which will print the points you got for this task.

Task 3: Temporal Heap Safety (3pts)

The final task is to detect temporal memory errors. In other words, you need to detect memory accesses to buffers that have been `free`'d before. The general approach for this is to *quarantine* memory when it is deallocated. In other words, you keep the free'd memory around for a while before reusing it. While in the quarantine, any access to it should be reported like an access to a red zone.

To avoid leaking memory, a buffer is usually removed from quarantine after a certain amount of `free` calls. However, for this assignment you can just assume that all programs are short-lived and you do not need to free up quarantined memory.

Grading:
Run **`./grade --assignment 3.3`** which will print the points you got for this task.

Hints:

- You can once again reuse some of the utility functions of the previous tasks for this one.

Assignment 4 - Optimization Verification

Deadline: See Canvas

Note: For BSc students, this assignment is optional. You will not get any points for doing it and it will not influence your grade.

This assignment is mandatory and graded like a normal assignment for MSc students.
Note: There is an alternate Assignment 4 you can do. You only have to do one of the two Assignment 4 we provide.

In this assignment you are given 3 incorrect compiler passes. Each pass contains a bug that causes subtle miscompilations in the optimized programs. In other words, the passes change the behavior of the optimized programs under some circumstances.
Your task is to fix all 3 bugs and provide a minimal test case that demonstrates the bug.

The code you're given are real LLVM optimization passes, so you will have a hard time spotting the bugs with the bare eye. This assignment will teach you how you can still find each bug easily using random differential testing.

Task 1: Differential testing script (1 pt)

Your first task is to write a testing script that takes a program and decides whether it exposes an optimization bug. This can be done by comparing the output (i.e., the `stdout` or exit code) of the program compiled with and without optimizations. For any well-defined deterministic program, a changed output implies an optimization pass is misbehaving.

For this task, we give you a trivially broken optimization pass called ExampleBrokenPass. You can find its source code in assign4/ExampleBrokenPass.cpp. You're also given a simple program that demonstrates the optimization bug in `assign4/tests/simple-load.c` as well as an undefined program in the `invalid-test.c` file in the same folder.

You can find the placeholder script in `assign4/is-interesting.py`. Adjust the file so that it returns the exit code 0 if the program demonstrates an optimizer bug. Otherwise return the exit code 1. If the program contains undefined behavior, you should also return exit code 1.

Grading:
Run **`./grade --assignment 4.1`** which will print the points you got for this task. The tests will check the example test programs above and the respective exit code.

Hints:

- The `assign4/optimize.sh` script allows compiling a program with only a certain compilation pass. You can invoke the script like this: **`optimize.sh input.c output.bin ExampleBrokenPass`** to generate an **`output.bin`** executable. In this example, **`ExampleBrokenPass`** is the name of a pass. You can pass **`None`** as a pass name to not use any optimizations during compilation.

Debugging Tasks

The next three tasks are all concerned with three optimization passes: GVN, GlobalOpt and InstSimplify. For each pass, you are expected to find and fix a bug we inserted into its implementation. You also have to provide a small example test case that demonstrates the found bug.

You can find the source of each pass in the `assign4/` folder (e.g., `GVN.cpp`). You can build all three passes by running make in the `assign4/` folder, and the grading script will build them for you automatically. You can use the `assign4/optimize.sh` script to run each of the provided passes on a source file and produce a binary.

Finding a test case

The intended way for you to find a test case for each bug is as follows:

1. Write a simple Python script that runs the csmith program generator to produce a well-defined random C program.
2. Use the `is-interesting.py` script from Task 1 to check if the generated program demonstrates an optimizer bug.
3. If the `is-interesting.py` script indicates an optimizer bug, you can stop and save the generated program. Otherwise go back to step 1 and repeat.

Once you find this good test case, you should put it in a file in the assign4/tests/ folder that has the name of the pass. E.g., `assign4/tests/GVN.c` for the GVN pass. This file is checked by the grading script.

Reducing the test case

You now have a program that demonstrates the bug, but you still don't know where the optimizer bug is nor is the giant random program useful for debugging. In the next step, you should minimize the program so you can use it to understand what bug the optimizer has. **For the sake of (automated) grading, we consider this task done when the test input is smaller than 300 bytes.**

We installed `creduce` on the docker setup that can automatically reduce a random C program. This is done by repeatedly removing a small part of the C program. Each time the program is modified, creduce checks if the modified version is still 'interesting'. If the modified version is not interesting, creduce will revert the change. If it is interesting, creduce will keep shrinking the program. You should read the manual page for creduce to understand better how it works: https://manpages.debian.org/testing/creduce/creduce.1.en.html

The definition of 'interesting' is left to the user of creduce (you) who has to provide a small script that is called by creduce after each reduction. In this task, interesting means that the smaller program still needs to be well-defined and demonstrate an optimizer bug. You can therefore reuse the `is-interesting.py` script from the first task.

You should be able to get a minimal test case by running creduce on your found test case and providing a small script that reuses the `is-interesting.py` script. However, creduce is not deterministic and might break the test case by introducing some kind of non-determinism. If the reduced test case is empty or non-deterministic, you should re-run creduce until you find a test case that looks right. You can also find out if there is a compiler warning (or some other utility) that can detect the non-determinism and enable this in the `has-ub.py` script.

Once you have a minimal test case, replace the respective program in the `assign4/tests/` folder with your smaller version. This file is checked by the grading script.

Finding the optimizer bug

The final step is to find and fix the optimizer bug in each pass. How you approach this is up to you, but we recommend that you first understand the expected behavior of the (unoptimized) minimal test case you generated. Then, try to understand what changed in the optimized version and what the pass is supposed to do.

Once you understand what behavior changed, you can start searching the pass source code for the respective logic that breaks. We recommend looking at the generated/modified LLVM instruction in the test case and then search for that in the pass. E.g., if you see that the pass generates a bogus **`load`**, then look for places where a **`LoadInst`** is created in the pass.

Notes:

- The `assign4/original-src/` folder has a copy of the pass source code that you must **NOT** modify. The contents of this folder are reset on the grading environment. You can run these passes by passing the pass name with a .original suffix. E.g., `./optimize.sh input output GVN.original` .
- You need to actually fix the pass. Not just disable part of it (or all of it) by commenting out some chunk of code. If you ever disable/remove 6 or more lines of code, you are doing something wrong.
- All bugs inserted into the passes consist of small source changes. You only need to modify/remove a few lines and don't need to write additional code from scratch. No change is larger than 5 lines of code.
- CSmith programs require the following include paths to compile: `-I /usr/include/csmith/`

Task 2: InstSimplify (3pts)

The provided `InstSimplify` pass does what you suspect from its name. It takes instructions and tries to find a simpler way to express them. For example, `X := A + 0` could just be expressed as `X := A`. You already implemented a somewhat similar version of this in assignment 2.

You should find and fix the hidden bug in the `assign4/InstSimplify.c` and put a minimal C program that demonstrates this bug in `assign4/tests/InstSimplify.c`.

Grading:
Run **`./grade --assignment 4.2`** which will print the points you got for this task.

Task 3: GlobalOpt (3pts)

The provided `GlobalOpt` pass optimizes global variables and their uses. E.g., if a global variable is never assigned a value aside from its initial one, it's assumed to be constant and all uses are replaced with the initial value.

You should find and fix the hidden bug in the `assign4/GlobalOpt.c` and put a minimal C program that demonstrates this bug in `assign4/tests/GlobalOpt.c`.

Grading:
Run **`./grade --assignment 4.3`** which will print the points you got for this task.

Task 4: GVN (3pts)

The provided `GVN` pass performs Global Value Numbering, which removes redundant computations in a program. This is similar to CSE, but not tied to any syntactic/structural similarities in the code.

You should find and fix the hidden bug in the `assign4/GVN.c` and put a minimal C program that demonstrates this bug in `assign4/tests/GVN.c`.

Grading:
Run **`./grade --assignment 4.4`** which will print the points you got for this task.

Assignment 4 - Backend

Deadline: See Canvas

Note: For BSc students, this assignment is optional. You will not get any points for doing it and it will not influence your grade.

This assignment is mandatory and graded like a normal assignment for MSc students.
Note: There is an alternate Assignment 4 you can do. You only have to do one of the two Assignment 4 we provide.

Yet Another Note: There is no framework support this alternate Assignment 4. We'll leave you mostly to your own devices here. But you can ask about build help with LLVM.

In the previous assignments you have seen how to take a source file, transform it into IR, and finally do some optimizations on it. So far, actually converting the IR to executable machine instructions has been happening in the magic backend of LLVM. In this assignment you will have to add security features to the LLVM backend, allowing you to get familiar with the deepest parts of LLVM.

Code randomization

In modern control-flow hijacking (or code reuse) attacks, the attacker needs to know the exact layout of the code the hijacked control flow jumps into. Code randomization seeks to make the code layout of a protected program randomized and thus unpredictable to the attacker, raising the bar for such attacks. Many code randomization techniques are described in the literature (e.g., https://bit.ly/2O6rALO and https://bit.ly/2U3SQ1y). In this assignment, we will focus on register assignment randomization and code pointer hiding, two popular and practical code randomization techniques.

Register assignment randomization

Register assignment randomization ensures that register assignments are randomized in the final binary, making the registers used by any given instruction unpredictable and overall introducing significant randomness throughout the code. In this assignment, we will focus on a simple implementation that randomizes the order of all the general-purpose registers assigned by the register allocator on a per-function basis.

Code pointer hiding

Code pointer hiding ensures that if attackers learn the memory address of a given function (i.e., a function pointer) or call instruction (e.g., a return address)—for instance, by leaking information from memory—they still have no information about the neighboring instructions. This property can be enforced in a number of ways. In this assignment, we will focus on a simple implementation that inserts a random number of NOP instructions for padding at function entry/exit points and before/after every call instruction.

Spectre-RSB mitigation

A clever mitigation for Spectre variant #2 is given by the use of a retpoline in place of indirect jumps or indirect calls. Another Spectre variant targets the Return Stack Buffer (RSB) to cause a misspeculation (https://bit.ly/2tVl6bK). In this case, a variation of 1 retpoline, that we're going to call ret-retpoline, can be used to trap speculative execution in an infinite loop. This mitigation is explained in the paper at page 11.

Assignment Objectives

In this assignment you have to delve into the CodeGen (lib/CodeGen) and the targetdependant (lib/Target) parts of LLVM. In the former you will be able to implement compile-time register randomization and code pointer hiding, while in the latter you can implement ret-retpoline for x86.

Compiling LLVM

For this assignment you have to delve into the LLVM source tree and build LLVM yourself.

You can find the source code for LLVM 10 here: https://github.com/llvm/llvm-project/releases/download/llvmorg-10.0.1/llvm-10.0.1.src.tar.xz . The official LLVM build instructions can be found here: https://llvm.org/docs/CMake.html. The simplest way to build LLVM is by first creating a build directory via

```
mkdir build ; cd build
```

Then you can run these commands in the build directory to build LLVM:

`cmake -DLLVM TARGETS TO BUILD="X86" ..` and then `make -j8`.

Note: You should replace the '8' in the make command with the number of CPU cores/hardware threads on your machine. Note that the first build might take quite some time depending on your machine. You can do a (much faster) incremental build after the initial build via `make -j8` in the build directory you created above. You can find your own built version of LLVM tools in the `build/bin/` directory.

1 Register Assignment Randomization (3 pts)

In this part of the assignment you will implement register assignment randomization. To achieve this goal, you will add a new register allocator to LLVM. We split this into two different steps:

1.1 Adding a custom register allocator

Start by adding a new allocator that can be invoked with the llc command in the following manner: `llc -regalloc=coco` . Adding a register allocator can be quite tricky! Start by copying the file implementing the basic register allocator and changing all mentions of basic,

to coco (i.e., call the allocator RegAllocCoco). You need to change some header files (found in **`include/llvm/CodeGen`**). Hint: search for RABasic.

Also remember to add your new file implementing the RegAllocCoco allocator to the CMakeLists.txt in the lib/CodeGen directory. Build LLVM using **`cmake --build . --target install`**. You can check whether the allocator has been added by executing llc --help and looking for the **`-regalloc`** flag (it should now mention the coco register allocator).

1.2 Randomizing the registers

Once you have added your custom allocator you can start implementing the actual randomization. To improve the reproducibility of builds you should use the seed from the RandomNumberGenerator class. Get the random number generator seed for your allocator in the following manner: **`mf.getFunction()->getParent()->createRNG(this);`**

Hints

- You should implement the register assignment randomization in the lib/CodeGen directory.
- The amount of code required to implement this is minimal. The hard part is finding where to implement it.
- Use tools such as gdb or objdump -D to verify that you are actually randomizing the registers.

2 Code Pointer Hiding (3 pts)

For this part, as an additional diversification strategy, you have to insert a random number of NOP instructions for padding at certain locations in the program.

2.1 Adding a MachineFunctionPass

Start by copying the MachineFunctionPrinterPass.cpp in the **`lib/CodeGen`** folder. This will give you a bare-bone MachineFunctionPass on which to base your implementation.

Your task is now to insert a random number (between 0 and a certain N) of NOP instructions at certain locations in the program.

MachineFunctionPasses are analogous to normal FunctionPasses, but are executed in the LLVM backend, where IR has already been transformed into machine code.

Refer to http://llvm.org/doxygen/classllvm_1_1MachineFunctionPass.html and the following blogpost (http://www.kharghoshal.xyz/blog/writing-machinefunctionpass) for more information on how to write machine passes.

2.2 Requirements

The implementation should be target-independent. Moreover, you should add a compiler flag that represents the largest size of a single NOP sled. Finally, you should add NOP padding at the following locations:

- Before the first/last instruction in a function
- Before each call site
- After each call site

3 Spectre-RSB Mitigation (4 pts)

In this part of the assignment you have to implement ret-retpoline as a mitigation to Spectre-RSB attacks. In particular, you have to replace every RET instruction with a 3 pre-determined snippet that will trap speculative execution in an infinite loop.

3.1 Adding a Target-Specific MachineFunctionPass

Create a new MachineFunctionPass inside lib/Target/X86 and, for each RET instruction you find, replace it with the following snippet:

```
call return_new
spec_trap:
pause
jmp spec_trap
return_new:
add rsp, 8
ret
```

3.2 Requirements

- You have to implement ret-retpoline only for x86 64
- To build the instructions you should use the MachineInstrBuilder and helpers like BuildMI().

Grading and Support

- The assignment is intentionally left open-ended. Do not expect much help. For instance, it is entirely up to you to figure out which files need to be modified in the LLVM backend. No hints will be given.
- Submit your solution as one (or three) patches for LLVM-10.0.1. Also include a (short) README file detailing what you did.