Masarykova univerzita
Fakulta informatiky

Extracting Parts of Programs into Separate Binaries

Master's thesis

Tomáš Mészaroš

Brno, 2018

Declaration

Hereby I declare, that this paper is my original authorial work, which I have worked out by my own. All sources, references and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

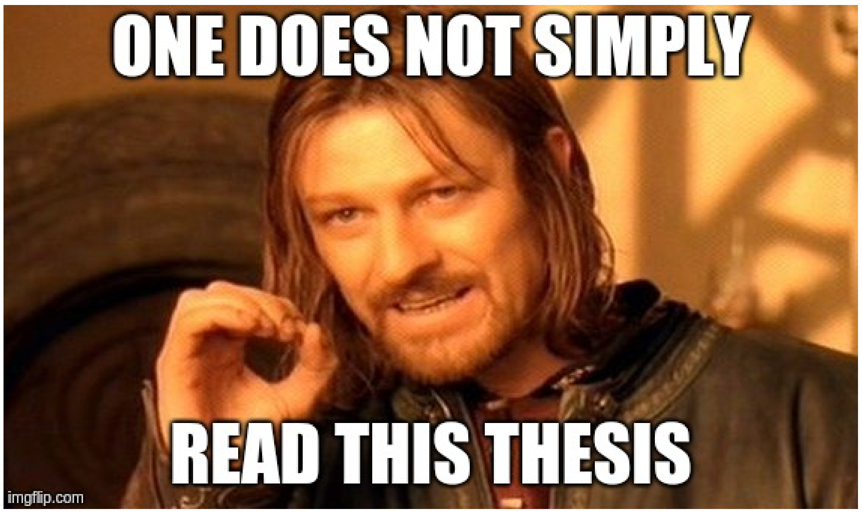
Tomáš Mészaroš

Advisor: Mgr. Marek Grác, Ph.D.

Acknowledgement

Dedicated to those who are brave enough to read this stuff.

You are heroes!



Abstract

abstract TBA

Keywords

keyword, keyword, keyword, keyword, keyword, keyword

Contents

1 Introduction: TODO

User wants to know the value of some variable in the program. He/she can run debugger of choice, set breakpoint at the selected variable location and let debugger execute input program step by step until it reaches the selected variable. Finally, debugger steps on the targeted variable and thus can extract its value and provide it back to the user.

The procedure described above is usually part of the standard standard approach when user want to get value of some selected variable during the program execution. Unfortunately, this approach is cumbersome in case when user want to execute above procedure many times. Procedure consists of many manual steps which is time consuming to perform. Ideally, there could be script that takes line of code (or variable name) as an input and produces output with the value of the selected target

Normally, this method would require to use debugger with the scripting support and write scripts that would instruct debugger what exactly to do, basically replicating the manual approach.

Instead of scripting debugger to do the extraction, we could write tool that would accept the same user input as the approach above (line of code/variable name), run analysis on where the execution flow in the program would occur to get to the target instruction and transplant subset of the input program into separate binary.

This way, user will have separate, executable that upon running would produce value of the targeted instruction, without having to manually step thought or script debugger.

This thesis aims to devise method and implement this method in a tool for statically transplanting a subset of a C program. Using the devised method, the selected program subset should be extracted from the original program provided by the user and synthesized as an independent, executable binary.

Proposed solution should be implemented in a tool having appropriate form, either as a standalone application or an LLVM plugin. It should easily accept user to provide their own input programs.

Finally, tool should be used to test at least two real-world open-source C programs in order to find where the room for improvements is and what could be improved in the future.

The following sections of this thesis are structured as follows. In the chapter 2 we will briefly introduce the LLVM compiler infrastructure. Explain what makes it so popular and why we picked this tool for our implementation. The following chapter 3, we will introduce method that is the basis of this thesis aim. We will devote chapter 4 for explaining specific details and intricacies of implementation. Experiments and their results will be discussed in the chapter chapter 5. Finally chapter 6 summarizes the results of this thesis and describes possible further research and development opportunities.

2 The LLVM Compiler Infrastructure: TODO

"The LLVM (FOOTNOTE:The name "LLVM" itself is not an acronym; it is the full name of the project.) Project is a collection of modular and reusable compiler and toolchain technologies." [LLV18b]

"an umbrella project that hosts and develops a set of close-knit low-level toolchain components (e.g., assemblers, compilers, debuggers, etc.), which are designed to be compatible with existing tools typically used on Unix systems"

"the main thing that sets LLVM apart from other compilers is its internal architecture." [6] Primary subprojects:

LLVM core clang ... Strengths: "A major strength of LLVM is its versatility, flexibility, and reusability"

2.1 Intermediate Representation

Introduction - IR AKA LLVM assembly language AKA LLVM

- "LLVM is a Static Single Assignment (SSA) based representation that provides type safety, low-level operations, flexibility, and the capability of representing ‘all’ high-level languages cleanly. It is the common code representation used throughout all phases of the LLVM compilation strategy."

- Aims: - "The LLVM representation aims to be light-weight and low-level while being expressive, typed, and extensible at the same time."

- Representations of IR: - as an in-memory compiler IR - as an on-disk bitcode representation (suitable for fast loading by a Just-In-Time compiler) - as a human readable assembly language representation

Example of the IR

We have the following C function add

```
int add(int a, int b) {
    return a+b;
}
```

When using clang compiler with -emit-llvm flag, we get the following representation in IR:

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
```

```
%a.addr = alloca i32, align 4
%b.addr = alloca i32, align 4
store i32 %a, i32* %a.addr, align 4
store i32 %b, i32* %b.addr, align 4
%0 = load i32, i32* %a.addr, align 4
%1 = load i32, i32* %b.addr, align 4
%add = add nsw i32 %0, %1
ret i32 %add
```

High Level Structure

- Module structure: - functions - global variables - symbol table entries

- using LLVM linker for module combination - we will use this in practice

- Functions: - "A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) for the function." - PHI nodes

2.2 Optimisations

LLVM uses the concept of Passes for the optimisations. Concrete optimisations are implemented as Passes that work with some portion of program code (e.g. Module, Function, Loop, etc.) to collect or transform this portion of the code. [LLV18a]

There are the following types of passes:

Analysis passes - Analysis passes collect information from the IR and feed it into the other passes. They can be also used for the debugging purposes, for example pass that counts number of functions in the module.

Examples:

- basiccg: Basic CallGraph Construction - dot-callgraph: Print Call Graph to “dot” file - instcount: Counts the various types of Instructions

Transform passes - Transform passes change the program in some way. They can use some analysis pass that has been ran before and produced some information.

Examples:

- dce: Dead Code Elimination - loop-deletion: Delete dead loops - loop-unroll: Unroll loops

Utility passes - Utility passes do not fit into analysis passes or transform passes categories.

Examples:

- verify: Module Verifier - view-cfg: View CFG of function - instnamer: Assign names to anonymous instructions

2.3 Clang

"The Clang project provides a language front-end and tooling infrastructure for languages in the C language family"

Features and Goals (some overview of clang): - End-User Features - Utility and Applications - Internal Design and Implementation

AST - what is AST - AST in clang - Differences between clang AST and other compilers ASTs - We will not use clangs AST, we will work directly with IR, it better suits this project

3 Extracting Program Subsets: WIP

TODO: weire intro about this chapter structure.
In this chapter, we introduce the method for extracting parts of programs from the provided user input.

We start with the method overview in the section 3.1 where we define what is the user input and briefly outline the method itself, what it does and what are the steps for achieving the final result.

We follow with the example of the method from the user perspective in the section 3.2.

After example, we present in detail each major step that is followed. Starting with computing data dependencies graph (section 3.3). Following with the procedure for finding connected components in the computed data dependencies graph (section 3.4). Next follows introduction of the call graph (section 3.5) and subsequently procedure for finding path from source to target in it (section 3.6). The chapter ends with the section describing methods on eliminating dead components and functions from the code (section 3.7).

3.1 Method Overview

Mandatory **input** for the method is a touple with the following definition:

$$input \equiv (code, target)$$

where:

- **code** is a C program source code compiled into the llvm bytecode.
- **target** is an integer value representing line of code from the C program source code.

We also define **source** as an entry to the C program (`main` function).

The method determines what parts of the *input* to extract according to the *source* and *target*. Procedure subsequently calculates possible execution path up to the *target* and extracts this execution path into the separate, functioning executable.

Barring implementation specific details (which are discussed in the chapter 4), the method can be summarized by the following five steps:

1. Compute data dependencies between instructions.
2. Find connected components in the computed data dependencies inside every function.
3. Construct call graph, mapping between connected components and functions that are being called from these components.
4. Find path from source to target in the call graph.

5. Eliminate dead components and functions that do not depend on the path.

Upon completion of the steps mentioned above, the llvm bytecode is produced as an *output*. We can define **output** as the extracted part of the original program according to the *source* and *target* while keeping the consistency of the code intact. By **consistency**, we mean that the *output* code is in a such state that it was possible to be compiled. *Output* is expected to be runnable the same way as the original. ****TODO? explain more here or in another chapter what it means for IR to be compiled without problems?****

3.2 Example

User provided us with the *input* in the form of the following C program source code that is stored in the file `example.c`:

`example.c`

```c
int foo(int n) {
    int x = n + 10;
    return x;
}

int bar(void) {
    int y = 42;
    return y;
}

int main(void) {
    int some_int = 10;
    int foo_result = foo(some_int);
    int bar_result = bar();
    return 0;
}
```

Since our method does not work directly with the C source code but instead works with the LLVM Intermediate Representation (IR), lets use clang and emit IR from the presented C source code in order to demonstrate the procedure more clearly: [1]

```
clang -S -emit-llvm example.c -o example.s
```

1. Flag `-S` tells clang to only run preprocess and compilation steps, while `-emit-llvm` makes sure to use the LLVM representation for assembler and object files. For detailed description of various clang flags, visit `https://clang.llvm.org/docs/ClangCommandLineReference.html`.

Emitted LLVM IR is stored in the file `example.s` and has the following structure:[2]

example.s

```
 1 define i32 @foo(i32 %n) #0 {
 2 entry:
 3   %n.addr = alloca i32, align 4
 4   %x = alloca i32, align 4
 5   store i32 %n, i32* %n.addr, align 4
 6   %0 = load i32, i32* %n.addr, align 4
 7   %add = add nsw i32 %0, 10
 8   store i32 %add, i32* %x, align 4
 9   %1 = load i32, i32* %x, align 4
10   ret i32 %1
11 }
12
13 define i32 @bar() #0 {
14 entry:
15   %y = alloca i32, align 4
16   store i32 42, i32* %y, align 4
17   %0 = load i32, i32* %y, align 4
18   ret i32 %0
19 }
20
21 define i32 @main() #0 {
22 entry:
23   %retval = alloca i32, align 4
24   %some_int = alloca i32, align 4
25   %foo_result = alloca i32, align 4
26   %bar_result = alloca i32, align 4
27   store i32 0, i32* %retval, align 4
28   store i32 10, i32* %some_int, align 4
29   %0 = load i32, i32* %some_int, align 4
30   %call = call i32 @foo(i32 %0)
31   store i32 %call, i32* %foo_result, align 4
32   %call1 = call i32 @bar()
33   store i32 %call1, i32* %bar_result, align 4
34   ret i32 0
35 }
```

User also provided the line number **7** from the `example.c` as the target, which corresponds to the line **16** from the `example.s`. Source is the main function.

2. Strictly speaking, this is not exactly the IR code that would be emitted by the clang. We have stripped it out of the module info and comments to make it more readable. To see the unmodified `example.s`, please go to the Appendix C.

Procedure for finding mapping between C code referenced by the user input and its analogous IR instruction is implementation detail and is be described in the chapter 4.

When we apply the method on the contents of the `example.s` with respect to the source and target, we get the result stored in the file `example_extracted.s` with the following code:

example_extracted.s

```
define i32 @bar() #0 {
entry:
  %y = alloca i32, align 4
  store i32 42, i32* %y, align 4
  ret i32 %0
}

define i32 @main() #0 {
entry:
  %bar_result = alloca i32, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}
```

As we can see from the `example.c`, execution path from the program entry in the `main` function (which we will call **source**) to the **target** does not include function `foo` and its associated instructions, they can be removed. We are left only with function `bar` which contains target, and necessary instructions in the function `main` along with the `main` itself.

We can now take `example_extracted.s` and recompile it back into the functioning executable.[3]

3.3 Computing Data Dependencies

In order to identify what parts of the IR we can afford to remove, it is imperative to compute dependencies between instructions. When removing instructions, we need to preserve consistency of the remaining code so that it can be later compiled into a functional executable. To ensure this, we first compute dependencies among instructions.

We recognize two types of dependencies between IR instructions: control and data dependencies. The following terminology and procedures for computing dependencies that we use are due to the Marek Chalupa master's thesis *Slicing of LLVM Bitcode* [CHA25].

3. More about recompilation in the chapter 4.

- “**Control dependence** explicitly states what nodes are controlled by which predicate.“
- “A **data dependence** edge is between nodes n and m iff n defines a variable that m uses and there is no intervening definition of that variable on some path between n and m. In other words, the definitions from n reach uses in m.“

The crucial information comes from data dependencies. We need to make sure that the IR integrity will remain intact after we are done with removing IR instructions.

The following example demonstrates data dependencies for the previously presented `example.c` source code. Taking closer look specifically at the function `main`:

```
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
  %foo_result = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %some_int, align 4
  %0 = load i32, i32* %some_int, align 4
  %call = call i32 @foo(i32 %0)
  store i32 %call, i32* %foo_result, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}
```

Taking `main` code, we can construct graph G where V is set of vertices (in our case vertex is instruction) and E is set of edges (in our case, edge between vertices V1 and V2 represents data dependency between instruction V1 and V2). In order to compute data dependencies, we use `dg` library. [CHA25] [4]

Computed **data dependency graph** for instructions from the function `main` is presented in the Figure 3.1. This graph is stored and used in the next step of the method for finding connected components (section 3.4).

Taking a closer look at the instruction: `%some_int = alloca i32, align 4`. We see that the following instructions have data dependency on the `%some_int`

4. For more info please visit `https://github.com/mchalupa/dg` [CHA25]. We will take a closer look at `dg` in the chapter 4.

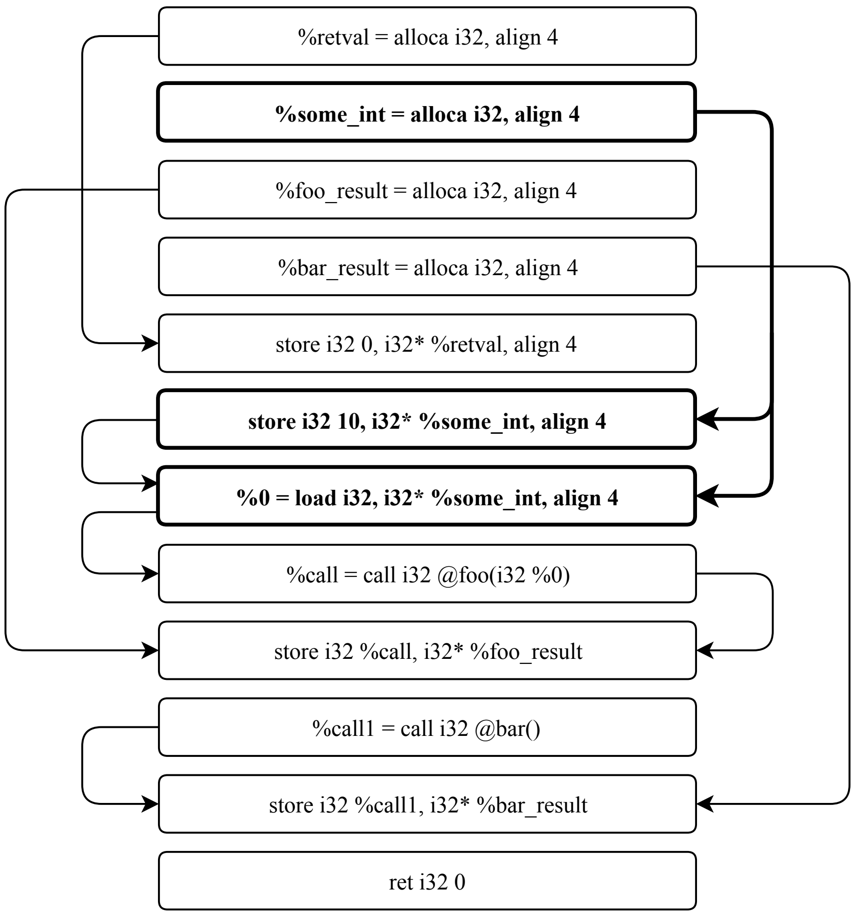

%retval = alloca i32, align 4
%some_int = alloca i32, align 4
%foo_result = alloca i32, align 4
%bar_result = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 10, i32* %some_int, align 4
%0 = load i32, i32* %some_int, align 4
%call = call i32 @foo(i32 %0)
store i32 %call, i32* %foo_result
%call1 = call i32 @bar()
store i32 %call1, i32* %bar_result
ret i32 0

Figure 3.1: Data dependencies graph of the `main` function instructions.

```
store i32 10, i32* %some_int, align 4
%0 = load i32, i32* %some_int, align 4
```

It is apparent that both instructions need `%some_int` for their operand. If we removed `%some_int = alloca i32, align 4` without taking into consideration that there are two instructions that depend on it, we would get into inconsistent state and two dependent instructions would contain undefined values as their operand.[5] This would lead into unsuccessful recompilation of the modified code back into the executable.

3.4 Finding Connected Components

Having shown in the previous section what are inter-instruction data dependencies and how important they are in relation to the code consistency. However, they do not fully

5. More about undefined values at https://llvm.org/docs/LangRef.html#undefined-values, https://llvm.org/docs/FAQ.html#what-is-this-undef-thing-that-shows-up-in-my-code, https://llvm.org/doxygen/classllvm_1_1UndefValue.html

solve our problem of knowing when it is safe to remove instruction. Data dependencies between only two instructions do not reveal the whole picture. Since we have computed and stored graph of data dependencies, let us propose the idea of computing connected components of this graph.

We define **connected component** as an isolated subgraph, where each pair of vertices is connected by some path.

We use **data dependency graph** computed in the section section 3.3 to find its connected components by using the following algorithm.

```
finding components
0. Let G = data dependency graph
1. Run Breadth-first search [Cor+09] on G to visit each instruction
2. IF instruction not in any component:
       create new component and put instruction inside
   ELSE:
       go to next instruction
```

Running the above mentioned algorithm on the **data dependency graph** produces connected components for each function in the input. As an example, we present components for the `main` function in the Figure 3.2 (for the clarity, each component has its own color).

Having instructions within each function separated into connected components comes useful especially because we can answer the question if some particular instruction is in data dependency relationship with multiple other instructions (instructions that form a data dependency path, etc.).

3.5 Computing Call Graph

In respect to the program execution flow, user provided the target and we know the source. In order to proceed further, it is needed to compute possible paths from source to target that could be used by the execution of the program. Computing and walking the call graph of the program can produce for us this piece of information.

A **call graph**[6] is a control flow graph that represents relationship between program procedures in respect to control flow.[SSK09] Having call graph $G = \{V, E\}$, set of vertices V typically represents functions and set of edges E represents transfer of control flow from one function to another.

In our context, call graph represents relationship between individual connected components (computed in the section 3.4) and functions that are being called from these

6. More technically, *call multigraph* [SSK09].

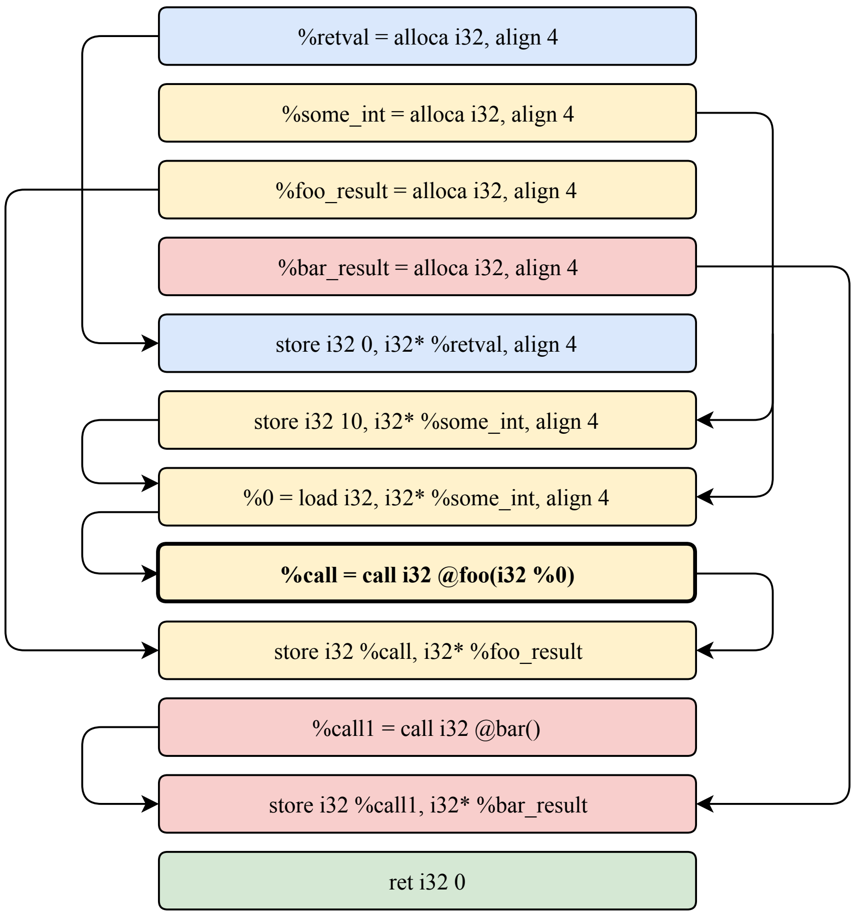

%retval = alloca i32, align 4
%some_int = alloca i32, align 4
%foo_result = alloca i32, align 4
%bar_result = alloca i32, align 4
store i32 0, i32* %retval, align 4
store i32 10, i32* %some_int, align 4
%0 = load i32, i32* %some_int, align 4
%call = call i32 @foo(i32 %0)
store i32 %call, i32* %foo_result
%call1 = call i32 @bar()
store i32 %call1, i32* %bar_result
ret i32 0

Figure 3.2: Connected components of the data dependencies graph for the `main` function instructions. Individual components are differentiated by the color.

components by one of its instructions. In other words, our call graph is a set of mappings from components to the function (or functions).

We construct the call graph using the following algorithm:

```
constructing call graph
0. Let FS = set of functions in the code
1. Let CS(f) = set of components inside function f
2. FOR EACH function F in the set FS:
      FOR EACH component C in the set CS(F):
         FOR EACH instruction I in the component C:
            IF instruction I is a call instruction to some function X:
               store information X gets called from the C
```

Running the presented algorithm on the **data dependence graph** of the `main` function that we computed earlier (Figure 3.2) produce call graph structure shown in the Figure 3.3. We know that in the context of the `main` function, there are four distinct components. We can see from the computed call graph, that `i32 @foo(i32 %n)` is being called from the *yellow* component, and `i32 @bar()` is being called from the *red* component.

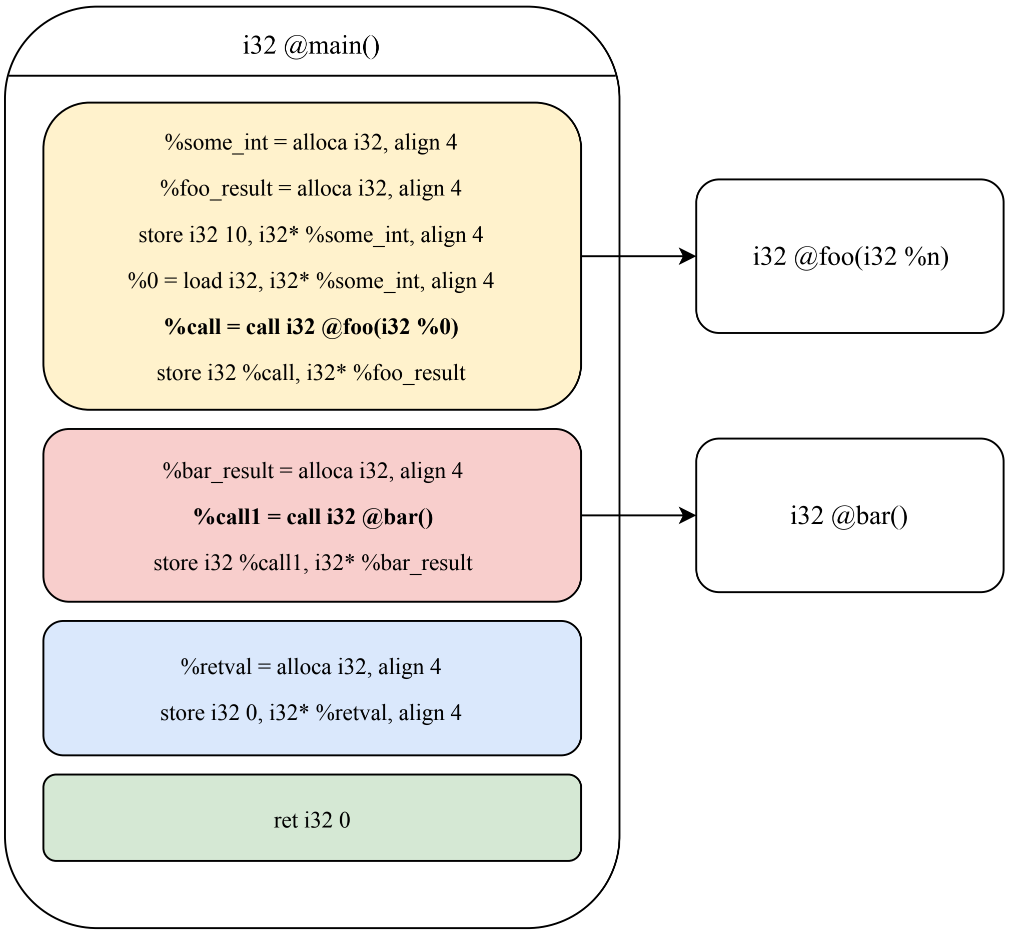

i32 @main()
%some_int = alloca i32, align 4
%foo_result = alloca i32, align 4
store i32 10, i32* %some_int, align 4
%0 = load i32, i32* %some_int, align 4
%call = call i32 @foo(i32 %0)
store i32 %call, i32* %foo_result
%bar_result = alloca i32, align 4
%call1 = call i32 @bar()
store i32 %call1, i32* %bar_result
%retval = alloca i32, align 4
store i32 0, i32* %retval, align 4
ret i32 0
i32 @foo(i32 %n)
i32 @bar()

Figure 3.3: Call graph computed from the connected components as shown in the Figure 3.2.

Having call graph represented in the structure shown in the Figure 3.3 is beneficial for finding program execution flow path between specific components within the program in relation to their dependencies. We can find path from source to target and know which components this path contains.

3.6 Finding Path from Source to Target

Now that we have computed the call graph based on the connected components, it is possible to find a **path** from *source* to *target*.

"A path is a simple graph whose vertices can be arranged in a linear sequence in such a way that two vertices are adjacent if they are consecutive in the sequence, and are nonadjacent otherwise."[BM08] In our case, linear sequence of vertices consists of connected components.

The reason why we constructed our special call graph from the connected components is now apparent. We want to find a path from source to the target and in doing so, know which connected components are part of this path or not.

This means that we are finding execution path from main function (source) to the target. Potentially, there may exist infinite number of such paths. From the optimization standpoint, it would be fitting to find all (or at least as many as we can) paths from source to target and pick some path according to selected optimization criteria (shortest path, path with smallest connected components, etc.).

However, for our purposes, it will be sufficient to find any path. We will use *Breadth-first search* to find such path and work with this path in the following stage.

As we have presented in the beginng of the chapter 3, user specified as the target line 7 in the `example.c` source code, which corresponds to the following IR instruction:

target IR instruction

```
store i32 42, i32* %y, align 4
```

Since we have source, target and also call graph, we can run *Breadth-first search* and obtain path from source to target in the call graph as shown in the Figure 3.4.

Given source and target, we can see from the Figure 3.4 that path only contains two components, one from `main` function and another from `bar`. These two components will be the core of the final, extracted program.

3.7 Eliminating Dead Components and Functions

Having successfully found path, we can proceed to the final phase. We need to eliminate all components and functions that do not impact the path.

If we take a look at the Figure 3.5, we can see that components marked as *red* do not impact the path at all and therefore, we can mark them for elimination.

The *yellow* component with the single instruction stands out. We will not remove this component because it contains terminator instruction `ret i32 0`.[7]

When we are done going over every component and collecting selected ones for elimination, we can proceed to actually eliminate them (remove them from the `example.s`).

After we are done with components, we can move to the functions. It may happen that there will be function not impacting path. For example, taking function `i32 @foo(i32 %n)` from the Figure 3.5, It is clearly not impacting path. It even has its one and only component marked for elimination. In this case, we can proceed and remove function `i32 @foo(i32 %n)` altogether.

7. `https://llvm.org/doxygen/classllvm_1_1TerminatorInst.html` More about terminators and why they need special treatment in the chapter 4.

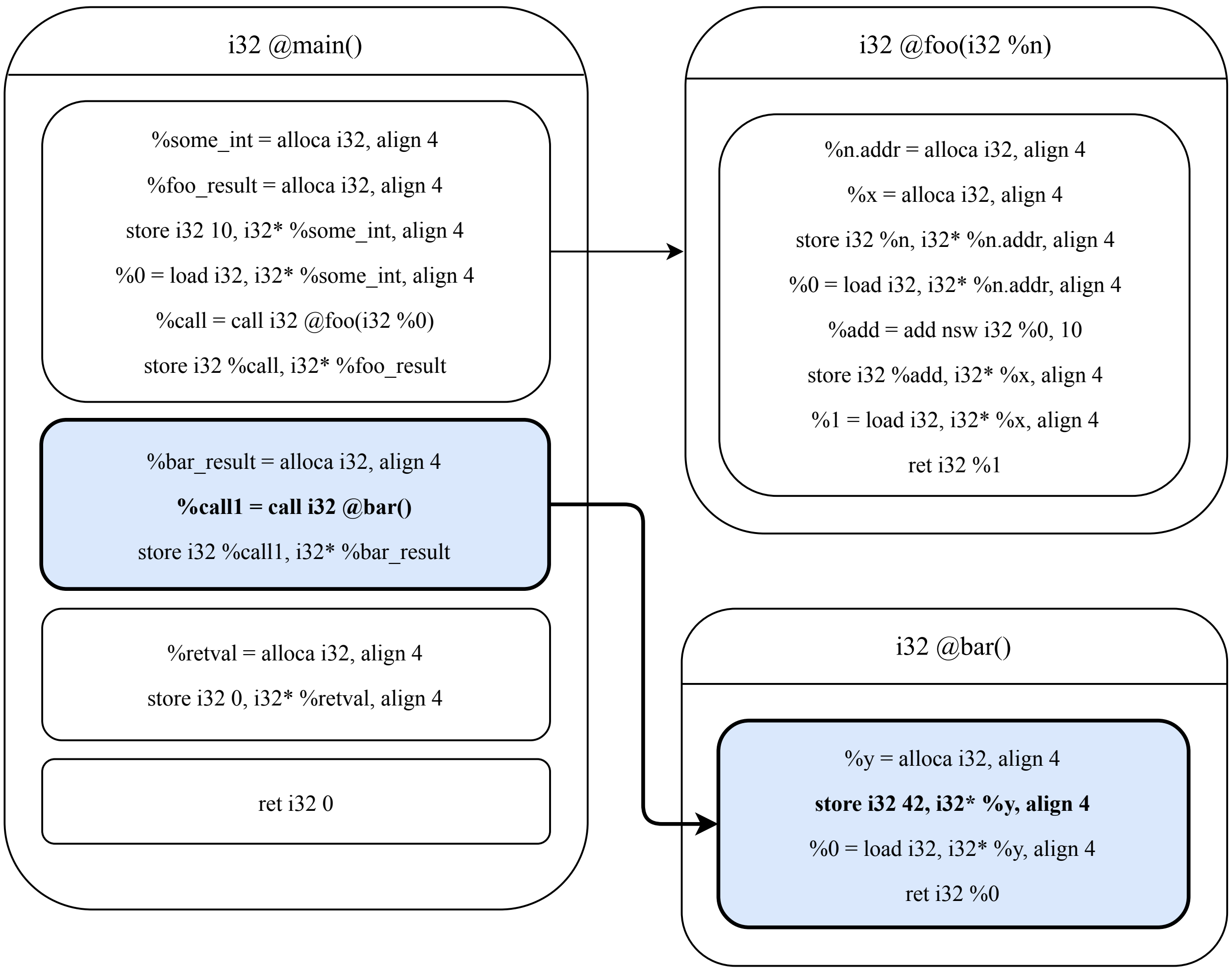

i32 @main()
%some_int = alloca i32, align 4
%foo_result = alloca i32, align 4
store i32 10, i32* %some_int, align 4
%0 = load i32, i32* %some_int, align 4
%call = call i32 @foo(i32 %0)
store i32 %call, i32* %foo_result
%bar_result = alloca i32, align 4
%call1 = call i32 @bar()
store i32 %call1, i32* %bar_result
%retval = alloca i32, align 4
store i32 0, i32* %retval, align 4
ret i32 0
i32 @foo(i32 %n)
%n.addr = alloca i32, align 4
%x = alloca i32, align 4
store i32 %n, i32* %n.addr, align 4
%0 = load i32, i32* %n.addr, align 4
%add = add nsw i32 %0, 10
store i32 %add, i32* %x, align 4
%1 = load i32, i32* %x, align 4
ret i32 %1
i32 @bar()
%y = alloca i32, align 4
store i32 42, i32* %y, align 4
%0 = load i32, i32* %y, align 4
ret i32 %0

Figure 3.4: Path from **source** to **target** computed on the Figure 3.3. Components in the path are *blue*.

Finally, finished with elimination process, we are left with the result in the form of the Figure 3.6.

TODO: this below should be move to the section 3.7

Lets take `%call = call i32 @foo(i32 %0)` as an example. Having computed connected components, we can see that this instruction belongs to the *yellow* connected component.

Now, if we were not interested in this particular instruction and wanted to remove it, we now know that we need to take care a closer look at the whole component and not only on one particular instruction.

TODO: Explicitly define when we are removing functions and what exactly are we doing

It would not be sufficient to only remove `%call = call i32 @foo(i32 %0)` because there are other instructions within the same component, which means that other members of the *yellow* component are in dependency relationship with each other.

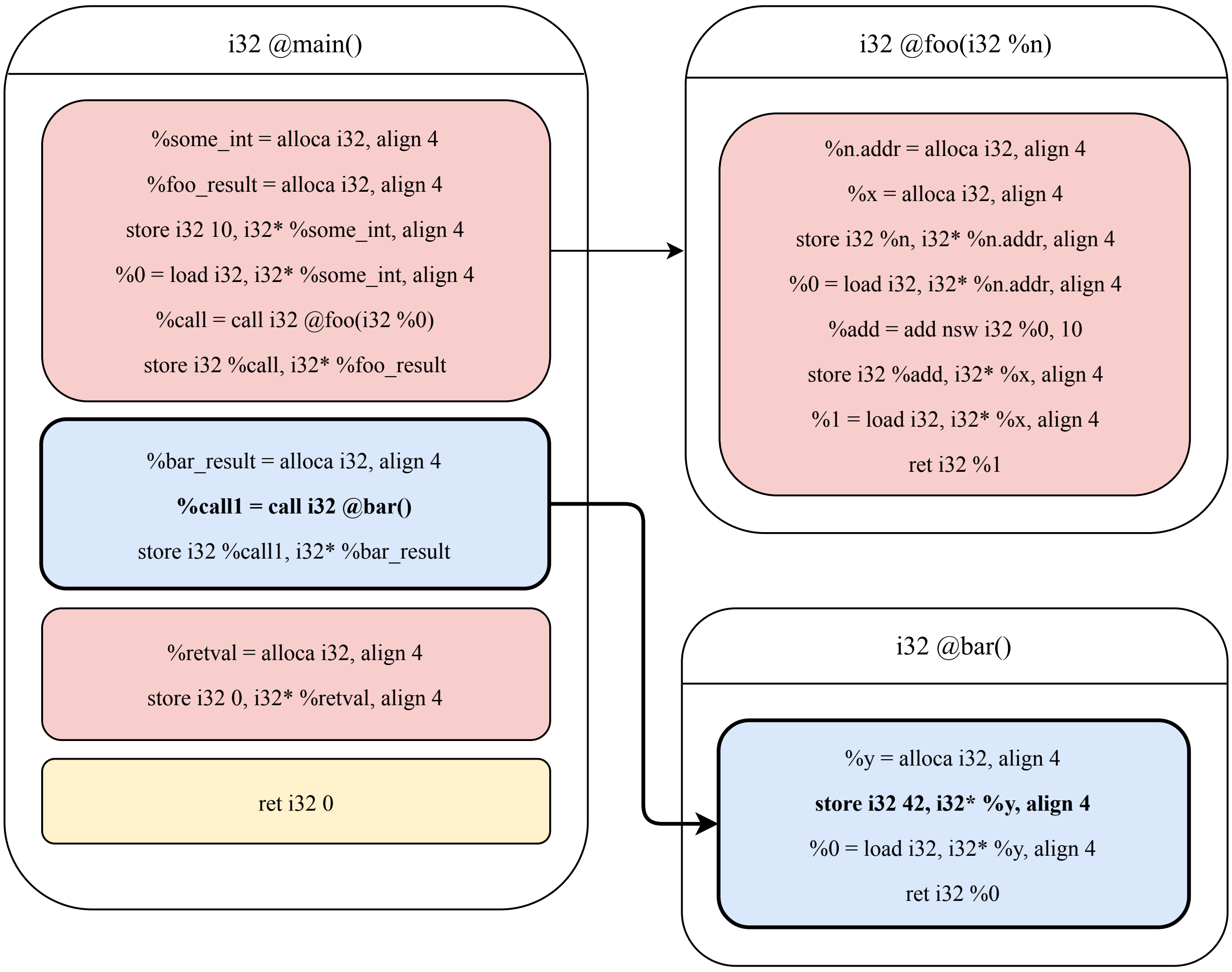

i32 @main()
%some_int = alloca i32, align 4
%foo_result = alloca i32, align 4
store i32 10, i32* %some_int, align 4
%0 = load i32, i32* %some_int, align 4
%call = call i32 @foo(i32 %0)
store i32 %call, i32* %foo_result
%bar_result = alloca i32, align 4
%call1 = call i32 @bar()
store i32 %call1, i32* %bar_result
%retval = alloca i32, align 4
store i32 0, i32* %retval, align 4
ret i32 0
i32 @foo(i32 %n)
%n.addr = alloca i32, align 4
%x = alloca i32, align 4
store i32 %n, i32* %n.addr, align 4
%0 = load i32, i32* %n.addr, align 4
%add = add nsw i32 %0, 10
store i32 %add, i32* %x, align 4
%1 = load i32, i32* %x, align 4
ret i32 %1
i32 @bar()
%y = alloca i32, align 4
store i32 42, i32* %y, align 4
%0 = load i32, i32* %y, align 4
ret i32 %0

Figure 3.5: Components marked for removal are marked *red*. *Yellow* component is not marked for removal, because it contains terminator.

The good thing is that if we were to remove some unwanted instruction, for instance `%retval = alloca i32, align 4` from the *blue* component we now have the information and know that we should remove also `store i32 0, i32* %retval, align 4`. This way, we will not leave any instructions stranded and with missing dependencies. As we mentioned earlier, this will ensure that the IR is in the consistent state and we will be able to recompile the modified IR back into the functioning executable.

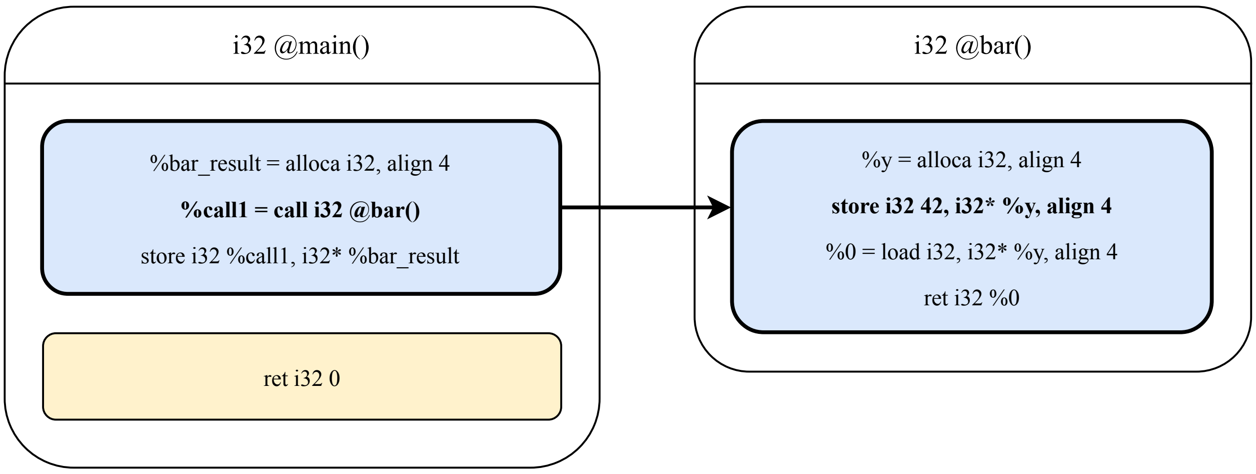

i32 @main()
%bar_result = alloca i32, align 4
%call1 = call i32 @bar()
store i32 %call1, i32* %bar_result
ret i32 0
i32 @bar()
%y = alloca i32, align 4
store i32 42, i32* %y, align 4
%0 = load i32, i32* %y, align 4
ret i32 %0

Figure 3.6: Final state after removing dead components and functions.

4 Implementation: TODO

4.1 APEX

4.1.1 what is APEX

4.1.2 two components of apex

1. launcher
2. APEXPass

4.2 Launcher

4.2.1 compiling input to bytecode with dbg symbols

4.2.2 running basic opts

4.2.3 linking input with apexlib

4.2.4 running apexpass

4.2.5 exporting call graph, dependency graph, disassembly bc

4.2.6 running final binaries

4.3 APEXPass

4.3.1 User Inpur Parsing, Mapping C code to IR

4.3.2 Compute data dependencies between instructions.

- running dg, init apexdg

4.3.3 Find connected components in the apexdg.

4.3.4 C

onstruct call graph, mapping between connected components and functions that are being called from these components.

4.3.5 Find path from source to target in the call graph.

4.3.6 Eliminate dead components and functions that do not depend on the path.

1. Handling of terminators

4.3.7 Injecting extraction and exit

4.3.8 Recompilation, stripping debug symbols

The simplest and seemingly correct way would be to remove every connected component that is not part of the path that we calculated in the earlier chapter.

This approach would unfortunately produce inconsistent IR. It not enough to remove only components in the path. We need to include every other component that is dependent on any other component that is already part of the path.

Checking if we have any branching dependent on the @path

1. investigating block, collecting basic blocks
2. block has no instruction in "if.*" basic block
3. block has some instruction in "if.*" basic block
4. Find branch instruction that services this BB and add block associated with this branch instruction to the @path.

Computing what dependency blocks we want to keep

1. marking every block from @path as to keep
2. Mark as visited to make sure we do not process this block in BFS.
3. setting up initial queue for BFS search
4. Go over @path and figure out if there are any calls outside the @path. If there are, put those called blocks for investigation into the @queue.
5. running BFS
6. Run BFS from queue and add everything for keeping that is not visited.
7. Collecting everything that we do not want to keep
8. We store blocks and functions that we want to remove into sets.

Removing unwanted blocks

1. Remove instructions that we stored earier. Watch out for terminators (do not remove them).
2. Do not erase instruction that is inside target instructions (We need those instructions intact.)

4.4 Recompilation back to the executable

5 Experiments: TODO

6 Conclusions: TODO

6.1 Summary of the Results

6.2 Further Research and Development

Bibliography

[BM08] J. Bondy and U. Murty, *Graph theory*, 1st. Springer Publishing Company, Incorporated, 2008, ISBN: 1846289696.

[CHA25] M. CHALUPA, "Slicing of llvm bitcode [online]", Master's thesis, Masaryk University, Faculty of Informatics, Brno, 2016 [cit. 2018-11-25]. [Online]. Available: `Available%20from%20WWW%20%3Chttps://is.muni.cz/th/vik1f/%3E`.

[Cor+09] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to algorithms, third edition*, 3rd. The MIT Press, 2009, ISBN: 0262033844, 9780262033848.

[LLV18a] LLVM. (2018). LLVM's Analysis and Transform Passes, [Online]. Available: `https://llvm.org/docs/Passes.html` (visited on 11/09/2018).

[LLV18b] ——, (2018). The LLVM Compiler Infrastructure, [Online]. Available: `https://llvm.org/` (visited on 11/03/2018).

[SSK09] A. Sanyal, B. Sathe, and U. Khedker, *Data flow analysis: Theory and practice*. CRC Press, 2009.

A Archive structure

Content of the attached archive:

TBA TBA TBA

B Outline

Extracting Parts of Programs into Separate Binaries

1. Get acquainted with means of the compilation of C programs using the LLVM compiler infrastructure - clang, LLVM Internal Representation, AST, LLVM optimizations.
2. Propose a solution to statically transplant a subset of a C program. This subset should be extracted from the original program and synthesized as an independent binary.
3. Design and implement the proposed solution in a tool having an appropriate form (a standalone application or an LLVM plugin).
4. Test the implemented tool on at least 2 real-world open-source C programs.

- ▪ Introdution
  - Give introduction to wider context
  - Clearly explain aim of the thesis
  - Give outline of the following chapters
- ▪ The LLVM Compiler Infrastructure
  - IR
  - Optimizations
  - clang
- ▪ Extracting Program Subsets
  - Intro
  -
  - Computing Data Dependencies
  - Finding Connected Components
  - Constructing Call Graph
  - Finding Path
  - Removing Unnecessary Parts
- ▪ Implementation
  - APEX
  - APEXPass
  - Input Source Code
  - Parsing User Input (Locating Target Instructions)
  - Computing Dependencies using dg
  - Extracting Target Data (Injecting Exit and Extraction)
    - ∗ Stripping debug symbols
- ▪ Experiments
  - Experiment 1
  - Experiment 2

  - Experiment 3

- Conclusion
  - Show our contribution to the problem
  - Show wider image in context to this thesis

C example.s

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

; Function Attrs: noinline nounwind optnone uwtable
define i32 @foo(i32 %n) #0 {
entry:
  %n.addr = alloca i32, align 4
  %x = alloca i32, align 4
  store i32 %n, i32* %n.addr, align 4
  %0 = load i32, i32* %n.addr, align 4
  %add = add nsw i32 %0, 10
  store i32 %add, i32* %x, align 4
  %1 = load i32, i32* %x, align 4
  ret i32 %1
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @bar() #0 {
entry:
  %y = alloca i32, align 4
  store i32 42, i32* %y, align 4
  %0 = load i32, i32* %y, align 4
  ret i32 %0
}

; Function Attrs: noinline nounwind optnone uwtable
define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %some_int = alloca i32, align 4
  %foo_result = alloca i32, align 4
  %bar_result = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  store i32 10, i32* %some_int, align 4
  %0 = load i32, i32* %some_int, align 4
  %call = call i32 @foo(i32 %0)
  store i32 %call, i32* %foo_result, align 4
  %call1 = call i32 @bar()
  store i32 %call1, i32* %bar_result, align 4
  ret i32 0
}

attributes #0 = { noinline nounwind optnone uwtable
 ↪  "correctly-rounded-divide-sqrt-fp-math"="false"
 ↪  "disable-tail-calls"="false" "less-precise-fpmad"="false"
 ↪  "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
 ↪  "no-infs-fp-math"="false" "no-jump-tables"="false"
 ↪  "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false"
 ↪  "no-trapping-math"="false" "stack-protector-buffer-size"="8"
 ↪  "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2,+x87"
 ↪  "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.module.flags = !{!0}
!llvm.ident = !{!1}

!0 = !{i32 1, !"wchar_size", i32 4}
!1 = !{!"clang version 5.0.1 (tags/RELEASE_500/final)"}
```