Masarykova univerzita
Fakulta informatiky

Extracting Parts of Programs into Separate Binaries

Master's thesis

Tomáš Mészaroš

Brno, 2018

MASARYKOVA UNIVERZITA
Fakulta informatiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student:	Bc. Tomáš Mészaroš
Program:	Aplikovaná informatika
Obor:	Aplikovaná informatika
Specializace:	Bez specializace
Garant oboru:	prof. RNDr. Jiří Barnat, Ph.D. (AP)
Vedoucí práce:	Mgr. Marek Grác, Ph.D.
Katedra:	Katedra strojového učení a zpracování dat
Název práce:	Extracting Parts of Programs into Separate Binaries
Název práce anglicky:	Extracting Parts of Programs into Separate Binaries

Zadání:

- Get acquainted with means of the compilation of C programs using the LLVM compiler infrastructure - clang, LLVM Internal Representation, AST, LLVM optimizations.
- 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.
- Design and implement the proposed solution in a tool having an appropriate form (a standalone application or an LLVM plugin).
- Test the implemented tool on at least 2 real-world open-source C programs.

Zadání bylo schváleno prostřednictvím IS MU.

Prohlášení autora školního díla

Jméno, příjmení a UČO studenta: Tomáš Mészaroš 422336

Beru na vědomí, že *Masarykova univerzita* může na základě zákona (§ 35 odst. 3 a 4 autorského zákona č. 121/2000 Sb.) užít mou kvalifikační práci nebo jiné mé školní dílo, které jsem jako autor vytvořil ke splnění svých studijních povinností vůči této vysoké škole, a to k výuce nebo k její vlastní vnitřní potřebě nikoli za účelem přímého nebo nepřímého obchodního nebo jiného hospodářského prospěchu.

Vlastní vnitřní potřebou *Masarykovy univerzity* se rozumí užití díla nejen v původní podobě, ale též ve zpracované nebo jinak změněné podobě zahrnující též takové užití mého díla touto vysokou školou, které spočívá v zadání mého školního díla k dalšímu zpracování jinému studentovi této vysoké školy (členovi téže akademické obce) za účelem vytvoření další kvalifikační práce nebo jiného školního díla, které bude odvozené od mého díla při uvedení mého autorství, názvu mého díla a pramene; a to vše v souladu s rozvojem vzdělanosti na *Masarykově* univerzitě a zájmem této veřejné vysoké školy navazovat na výsledky mé práce a mé školní dílo dále rozpracovávat v téže akademické obci.

Okolnosti hodné zvláštního zřetele z mé strany, například zájem o vlastní další rozpracování své kvalifikační práce na *Masarykově univerzitě* nebo jinde, jsem povinen sdělit této vysoké škole prostřednictvím studijního oddělení nejpozději při odevzdání kvalifikační práce.

V Brně dne 10. 12. 2018

podpis studenta

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

Thesis Thanks TBA - Thesis Thanks TBA - Thesis Thanks TBA Thesis Thanks TBA - Thesis Thanks TBA - Thesis Thanks TBA Thesis Thanks TBA - Thesis Thanks TBA - Thesis Thanks TBA Thesis Thanks TBA - Thesis Thanks TBA - Thesis Thanks TBA Thesis Thanks TBA - Thesis Thanks TBA - Thesis Thanks TBA Thesis Thanks TBA - Thesis Thanks TBA - Thesis Thanks TBA

Abstract

Thesis Abstract TBA - Thesis Abstract TBA - Thesis Abstract TBA Thesis Abstract TBA - Thesis Abstract TBA - Thesis Abstract TBA Thesis Abstract TBA - Thesis Abstract TBA - Thesis Abstract TBA Thesis Abstract TBA - Thesis Abstract TBA - Thesis Abstract TBA Thesis Abstract TBA - Thesis Abstract TBA - Thesis Abstract TBA Thesis Abstract TBA - Thesis Abstract TBA - Thesis Abstract TBA

Keywords

keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword, keyword

Contents

1 Introduction

TODO: CITATION TEST, REMOVE THIS LATER:

[CHA25] [LLV18d] [LLV18a] [LLV18b] [LLV18c]

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

"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." [LLV18d]

"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

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

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

```llvm
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 Design of the Method

- calculate data dependencies between instructions - compute connected components out of calculated data dependencies inside every function - compute callgraph, mapping between connected components and functions they are calling - find path from source to target in call graph - remove functions and other connected components that do not depend on the path

3.1 Calculating Data Dependencies

In order to identify what parts of the IR we need to remove, we need to compute dependencies between instructions.

We recognize two types of dependencies between IR instructions: control and data. Terminology comes from the Mareks work.

- control dependence: "Control dependence explicitly states what nodes are controlled by which predicate."

- data dependence: "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 our transformation. We have to know which instructions depend which. We cannot afford to remove blindly any instruction, because it may happen that this removed instruction is needed by some other instruction.

The above rationale can be demonstrated on the example below:

```
%call3 = call i32 @flag()
store i32 %call3, i32* %f, align 4
%0 = load i32, i32* %f, align 4
```

We have tree instructions. We call function flag, capture return value into variable call3, store value from call3 into variable f and finally load value from f into 0.

We can see that if we would remove instruction

```
%call3 = call i32 @flag()
```

it would mean that call3 is going to be undefined and instruction

```
store i32 %call3, i32* %f, align 4
```

would work with undefined variable. This would break integrity of the IR as the later computation dependent on the

```
store i32 %call3, i32* %f, align 4
```

would fail.

We need to make sure we do not corrupt IR with undefined variables.

3.2 Finding Connected Components Between Data Dependencies

Since we know that there are data dependencies between instructions, 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).

Let us demonstrate on the following example. Lets take this simple C function:

```c
int z(void) {
  printf("hello from z\n");
  int tmp = 1;
  return tmp;
}
```

Above C code corresponds to the following IR:

```
define i32 @z() #0 {
entry:
  %tmp = alloca i32, align 4
  %call = call i32 (i8*, ...) @printf(i8* ...)
  store i32 1, i32* %tmp, align 4
  %0 = load i32, i32* %tmp, align 4
  ret i32 %0
}
```

Graph G would look like (in the adjacency list form):

```
[%tmp = alloca i32, align 4] -> [store i32 1, i32* %tmp, align 4]
[%call = call i32 (i8*, ...) @printf(i8* ...)] -> []
[store i32 1, i32* %tmp, align 4] -> [%0 = load i32, i32* %tmp, align 4]
[%0 = load i32, i32* %tmp, align 4] -> [ret i32 %0]
[ret i32 %0] -> []
```

Now, we can find connected components in G easily by traversing graph using BFS.

```
FUNCTION: z
- COMPONENT:
    %tmp = alloca i32, align 4
    store i32 1, i32* %tmp, align 4
    %0 = load i32, i32* %tmp, align 4
    ret i32 %0
- COMPONENT:
    %call = call i32 (i8*, ...) @printf(i8* ...)
```

Having instructions within function separated into connected components comes useful. We can see that since call to printf is in the different component than other instructions. If we were to remove this printf call, integrity of the instructions in the other component would not be compromised. We can take advantage of this when we know for sure that we are interested about instructions in certain components and not in the others. We can proceed and remove whole components and not compromise integrity.

The procedure describing how are we going to pick components for removal and remove them will come later.

3.3 Constructing Call Graph

Call graph in general

Call graph is a control flow graph that represents relationship between program procedures in respect to control flow. Lets have call graph G = V, E, where set of vertices V typically represents functions and set of edges E represents calls from one function in V to `another`.

TODO: put here image of stock CFG

Call graph in our case

In our case, call graph represents relationship between individual function connected components and functions that these components call, more specifically instruction from component calls.

To demonstrate more clearly what exactly is callgraph in our context, lets take the following C program:

```
int y(void) {
  return 2;
}

int x(void) {
  return 1;
}

int main(void) {
  int x_ret = x();
  int y_ret = y();
  return 0;
}
```

Classic callgraph would look like:

```
[main] -> [x, y]
[x] -> []
[y] -> []
```

main calls x and y while both x and y do not call anything.

The IR representation of the above program:

```
define i32 @y() #0 {
entry:
  ret i32 2
}

define i32 @x() #0 {
entry:
  ret i32 1
}

define i32 @main() #0 {
entry:
  %retval = alloca i32, align 4
  %x_ret = alloca i32, align 4
  %y_ret = alloca i32, align 4
  store i32 0, i32* %retval, align 4
  %call = call i32 @x()
  store i32 %call, i32* %x_ret, align 4
  %call1 = call i32 @y()
  store i32 %call1, i32* %y_ret, align 4
  ret i32 0
}
```

After applying approach from the previous chapter, we get these connected components:

```
FUNCTION: y
- COMPONENT: y:1
    ret i32 2

FUNCTION: x
- COMPONENT: x:1
    ret i32 1

FUNCTION: main
- COMPONENT: main:1
    %retval = alloca i32, align 4
    store i32 0, i32* %retval, align 4
- COMPONENT: main:2
    %x_ret = alloca i32, align 4
    %call = call i32 @x()
    store i32 %call, i32* %x_ret, align 4
- COMPONENT: main:3
    %y_ret = alloca i32, align 4
    %call1 = call i32 @y()
    store i32 %call1, i32* %y_ret, align 4
- COMPONENT: main:4
    ret i32 0
```

As was written above, call graph in our situation is mapping between individual components and functions that are being called within these components. That gives us the following structure:

```
[y:1] -> []
[x:1] -> []
[main:1] -> []
[main:2] -> [x]
```

```
[main:3] -> [y]
[main:4] -> []
```

Having call graph represented in the structure above is beneficial for finding program execution flow path between specific instructions within the program in relation to their dependencies.

3.4 Finding Path from Source to Target

from wiki, citation needed dude "In graph theory, a path in a graph is a finite or infinite sequence of edges which connect a sequence of vertices which, by most definitions, are all distinct from one another."

In our case, sequence of vertices is sequence of connected components and sequence of edges which connect these connected components are function calls (from one component to another component within called function).

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

Source in our case is entry point to the program, main functions. Target is supplied by the user and consists of the pair filename, line number. This pair represents specific instruction (or set of instructions when compiled into IR).

This means that we are finding execution path from main function to the target fine of code. 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 any path and work with this path in the later stages (see chapter "removing unneeded stuff").

3.5 Removing Unneeded Stuff

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

- investigating block, collecting basic blocks - TBA - block has no instruction in "if.*" basic block

Then do nothing

- block has some instruction in "if.*" basic block - TBA

- Find branch instruction that services this BB and add block associated with this branch instruction to the @path. - TBA

Computing what dependency blocks we want to keep

- marking every block from @path as to keep Mark as visited to make sure we do not process this block in BFS.

- setting up initial queue for BFS search 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.

- running BFS Run BFS from queue and add everything for keeping that is not visited.

- Collecting everything that we do not want to keep We store blocks and functions that we want to remove into sets.

Removing unwanted blocks

Remove instructions that we stored earier. Watch out for terminators (do not remove them).

Do not erase instruction that is inside target instructions (We need those instructions intact.)

4 Implementation

[LLV18b] how do we extract remaining program adding to the path removing non-path functions at the end along with dependencies before and after code samples before and after call graphs

5 Experiments

6 Conclusions

6.1 Summary of the Results

6.2 Further Research and Development

Bibliography

[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.

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

[LLV18b] ——, (2018). LLVM's Analysis and Transform Passes, [Online]. Available: https://llvm.org/docs/Passes.html (visited on 11/09/2018).

[LLV18c] ——, (2018). opt - LLVM optimizer, [Online]. Available: https://llvm.org/docs/CommandGuide/opt.html (visited on 11/07/2018).

[LLV18d] ——, (2018). The LLVM Compiler Infrastructure, [Online]. Available: https://llvm.org/ (visited on 11/03/2018).

A Archive structure

Content of the attached archive:

TBA TBA TBA