École Polytechnique Fédérale de Lausanne

Modern methods for C++ Decompilation

by Liam Wachter

Master Project Report

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Dr. Flavio Toffalini
Thesis Supervisor

EPFL IC IINFCOM HEXHIVE
BC 160 (Bâtiment BC)
Station 14
CH-1015 Lausanne

January 6, 2023

Abstract

C++ reverse engineering of optimized binaries is notoriously hard, as higher level constructs are combined with surrounding code during optimization. One C++ language construct that regularly challenges reverse engineers are template classes and methods. Especially, vectors, sets and maps from the standard template library (STL) are prevalent in modern C++ code. One major problem for automatically identifying template code is inlining. This report investigates the challenges and potential solution approaches for identifying STL code, especially considering inlining and other optimizations with a focus on static approaches. In part this is done with the help of tools, that we implement, that can track how high level C++ constructs get transformed into low level binary code. We evaluate the our tools with respect to performance and correctness. The goal is to provide a useful framework for implementing a better decompilation of STL code in the future, for which we point out intersting directions in the light of challenges we discovered and describe.

Contents

Abstract **2**

1 Introduction **4**

2 Methodology **6**
2.1 LLVM IR Visual Diffs 6
2.2 codeexplorer++ 7
2.2.1 Dataset 9
2.3 Effect of Optimization 10

3 Evaluation **13**
3.1 Compiler Explorer Patch 13
3.2 codeexplorer++ 16

4 Exploring Solutions **19**

5 Conclusion **21**

Bibliography **22**

Chapter 1

Introduction

In 2022 C++ is the most popular language for writing programs that are compiled to binary code [13]. Compared to the also popular C, it allows programmers to use more high level abstractions, such as dynamic polymorphism and templates. While previous work on decompiling C++ focuses on recovery of classes and polymorphism [10], there is little help for reverse engineers when it comes to automatically identifying types and their usage from the STL. One main challenge for both issues are the compiler, platform and optimization dependent major differences in the binary code. E.g., the work of Schwartz et al. only targets MSVC[1] compiled 32 bit binaries. Their logic programming based approach for identifying class members requires hand writing a large amount of logic programming rules and manually optimizing them.

For the problem of identifying standard library functions, an approach that transfers to all optimization levels is desired. One promising approach is analysing the pointer structure on the heap collected from execution traces. This approach is implemented by the tools DDT [5] and MemPick [2]. In addition to the common problem, that all dynamic analysis approaches have of having code coverage, both of those fail, if the invariant is temporally violated by an opteration on the datastructure [9].

For the task of decompilation typically only static analysis is applied. This is due to a multitude of reasons such as execution time, architecture support, malicious code execution prevention, etc. To the best of our knowledge, there are no approaches for static identification of template code. Reading current reverse engineering blog posts and online discussions, it appears that the state of the art in industry is to have compiler specific helper scripts for decompilers such as HexRays and Ghidra [8, 11]. These help retyping once a reverse engineer knows the type of the template variables to make the code more readable, however these scripts provide no automatic detection or capability to deal with inlined code.

[1] Microsoft Visual C++, the C++ compiler of Microsoft

Studying C++ code decompilation and compilation, while only having source code and the compiled binary, is hard and tooling is lacking for showing the correspondance of low level instructions in disassembled code and high level code constructs in C++ source code. The state of the art is looking at compiler explorer[2] and hovering over code to see this correspondence. However, this feature fails for optimized code and compiler explorer lacks the ability to export or batch process binaries in order to create a labled dataset. Another state of the art technique is to use the decompiler Ghidra. Ghidra is able to show the correspondence between disassembled code and decompiled code. Also, Ghidra has a Java API providing programmatic access to many of the GUI features. However, this approach depends on correct decompilation in the first place, and as just described, no current decompiler is able to recognize STL types reliably.

To show this correspondence, fast and reliably from ground truth, we propose codeexplorer++ and an improvement for compiler explorer in Chapter 2. These tools track the flow from high level C++ code down to the resulting assembly code. Their performance and corectness is evaluated in Chapter 3. We hope these tools and the labeled dataset, that is manually created and labled with codeexplorer++, help future techniques, algorithmic or machine learning based, to improve decompilation of template code. Directions for further research, considering the challenges described in Section 2.3 and Section 3.2, are briefly discussed in Chapter 4. Also, as a side effect, they might help compiler developers, and anyone else interested, to study how optimizations affect low level code.

[2] godbolt.org

Chapter 2

Methodology

This chapter describes the tools we developed for studying the effects that C++ STL usage has on the low level binary code.

2.1 LLVM IR Visual Diffs

For the analysis of the binaries, that compilers produce at different optimization levels, and to quickly test many variants of input programs, it is necessary to have insights into the effects of the optimization stages of the compiler. While the specific implementation, order and parameters of optimizations differ from compiler to compiler, the main concepts and challenges are similar, as outlined in Chapter 2. clang++ uses the LLVM IR at every step of the compilation process, while g++ uses a multitude of typically less human readable intermediate representations during compilation. The consistency in syntax of LLVM IR allows to compute text diffs before and after every optimization pass. clang++ already has a way to get the IR before and every optimization pass with the hidden `print-after-all` and `print-before-all` flags, these print the IR before/after every pass to stderr. Since Oct 1 2020 there is also the `print-changed` flag that will omit printing IR, if there was no change done by a pass[1].

While visually showing diffs of neighbouring steps does not involve any unsolved computer science problems, it still requires time to implement in a already short semester project, with many other things to implement – the semi-structured output must be parsed and grouped by functions, loops and modules, the names must be demangled and the diffs must be shown in a graphical user interface preferably with syntax highlighting. We discovered that compiler explorer already has a functionality for this implemented. However compiler explorer will fail for even very simple C++ programs with the error: “An error occurred while generating the optimization pipeline output: Clang invocation timed out”. Considering the time it would take

[1] reviews.llvm.org/D86360

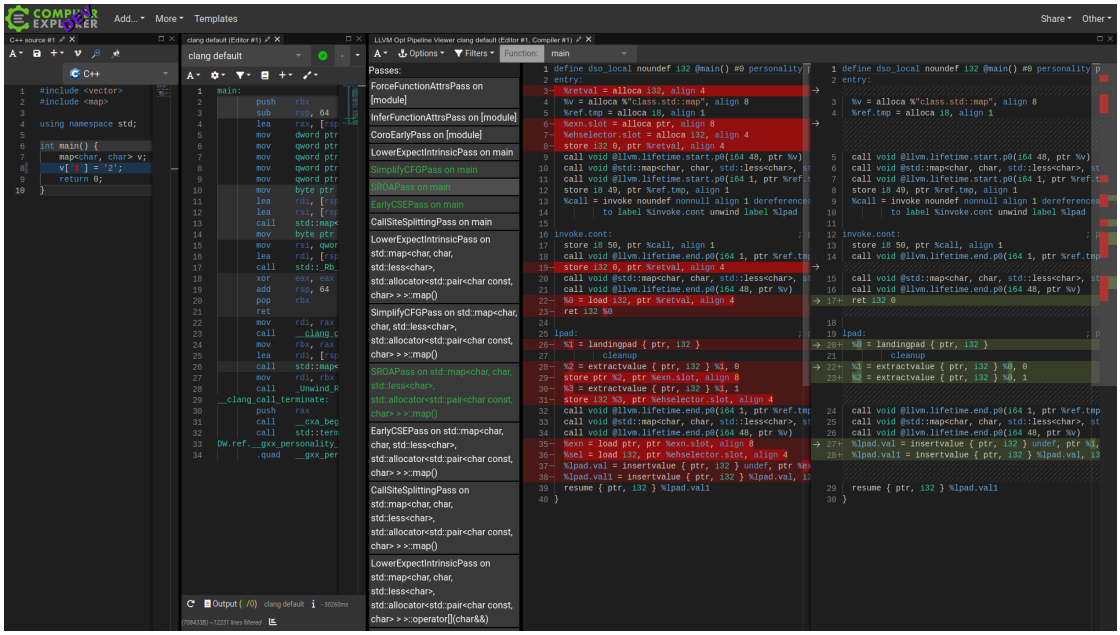

Figure 2.1: Screenshot of the LLVM IR diff view in patched compiler explorer

to develop a diff tool from scratch we decided to investigate the behaviour in compiler explorer, that is open source[2], in the hope to contribute the fix back and help more people facing this error message. The compiler explorer codebase is written in TypeScript and contains a lot of code for communicating to a large number of compilers and parse their respective outputs. After locating the code for the invocation of clang and finding the parsing logic of the IR dumps, we locate the origin of the error message. Further analysis reveals that the bottleneck is the amount of data communicated over stderr. Compiler explorer uses `print-after-all` and `print-before-all`, and therefore sends a lot of redundant data. Changing this reduces the amount of data send and improves the runtime as shown in Section 3.1. The original code is then reconstructed during parsing, leading to the exactly same output as the original compiler explorer, but the ability to handle larger programs, i.e. any kind of none tirival C++ program at a higher optimization level.

For this to be contributed back, the patch needs to be cleaned up and respect the fact, that not all versions of clang support the `print-changed` flag.

2.2 codeexplorer++

To get insights in what assembly instructions correspond to a line of C++ code or, most importantly, multiple lines of code in the case of an inlined call, we develop the tool *codeexplorer++*. codeexplorer++ can display and save this information for any binary, that supports dwarf symbols and is compiled from C or C++ code. It further has the capability to search large datasets for

[2] github.com/compiler-explorer/compiler-explorer

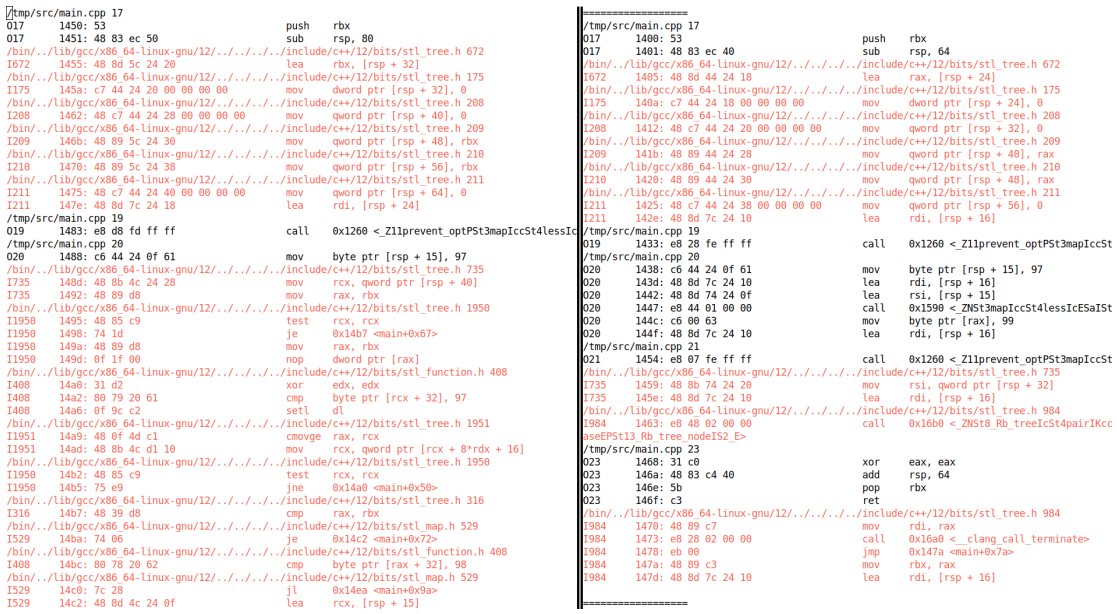

Figure 2.2: codeexplorer++ screenshot comparing a map creation/destruction and insert with clang++ with `O1` (left) and `O3` (right)

interesting calls, specifically for example calls to the STL. codeexplorer++ is not only useful for manually exploring compiler results, but can also play an important role for evaluating a solution or provide training and testing data for a learning based approach. We publish codeexplorer++ at `github.com/sevenmaster/declinefunction-annotator.git`.

codeexplorer++ can be run on a single file or on a whole dataset of C++ programs. The output shows instructions that are inlined for a specific compiler setting and specific functions of interest. Functions of interest can either be specified directly e.g. with a regular expression on the name or can be searched for with a CodeQL query. One predefined query searches for all calls in the STL. Results can be shown visually, as in Figure 2.2, saved as a json file, that maps function names to virtual addresses. Additionally, codeexplorer++ can compile a binary without optimization and only inline classes of functions that are specified beforehand, by annotating it with `always_inline` before passing the source file to the compiler.

CodeQL[3] is a query language, by GitHub that allows to write sophisticated queries on source code. Among others, CodeQL supports taint tracking, flow tracking and type parsing. It is primly used for variant analysis of vulnerabilities either for continuous scanning of repositories, or for large scale vulnerability discovery. CodeQL works by hocking the compiler and therefore needs to compile a codebase to create the search database. Therefore it takes at least as long as compilation to construct the search database. To offer a faster solution, in cases where only basic C++ parsing is required, codeexplorer++ also supports lizard[4] as a search backend. A brief comparison between both backends can be found in Section 3.2.

[3] codeql.github.com
[4] github.com/terryyin/lizard

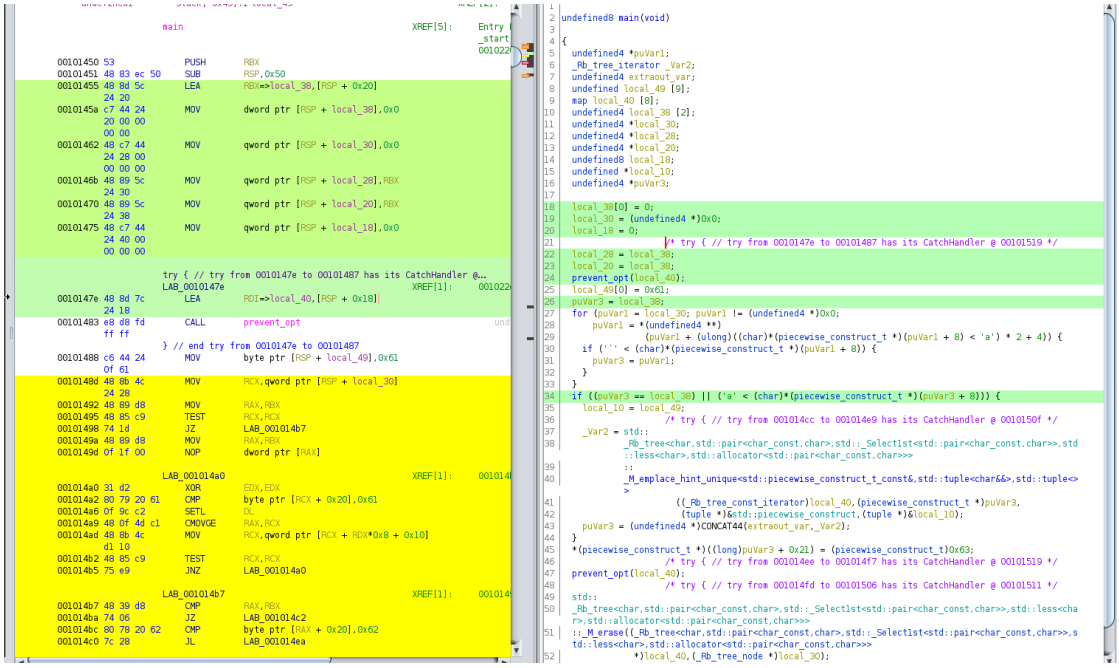

Figure 2.3: codexplorer++ Ghidra plugin showing the results of Figure 2.2 (left). The map constructor code is marked (green) in disassembly and decompilation.

The mapping from C++ source line is obtained with dwarf symbols. Adding dwarf symbols has very good compiler support by gcc and clang, however this design choice limits the applicability of this tool beyond ELF executables. The disassembling and dwarf parsing is done by llvm-objdump. To exactly see how inlined constructs get decompiled we provide a Ghidra plugin, that reads the saved json files from codeexplorer++ and highlights the inlined disassembly lines in yellow. As described in Chapter 1, Ghidra can then show the correspondence between disassembly and decompilation. Figure 2.3 shows a screenshot of the plugin showing the output of Figure 2.2 (left) in Ghidra. However the mapping Ghidra provides is not always totally correct. In the screenshot the call to `prevent_opt` is marked, even though it is not marked in the disassembly. Furthermore, there are cases where an assembly instruction is not mapped to any part of the decompilation.

2.2.1 Dataset

As mentioned we created an own dataset with STL calls in different settings. It is published at `github.com/sevenmaster/semester-project-dataset`. The goal of this dataset is to have isolated calls, in a controlled environment. This includes making sure that the call is not optimized out. To do that, there is a `prevent_opt` function before and after the relevant call, that function prints random elements from the STL type and also inserts random elements. It is impossible to consider every STL datatype and every method in every context, therefore we limit us on the STL datatypes and methods shown in Table 2.1. Table 2.1 further shows the types used for

StlType::Method	Template Type	Compiler	Optimization Level
map::at	char	g++	O0
map::pop_erase	int	clang++	O1
map::insert	double		O2
set::erase	string		O3
set::insert	struct		
vector::at			
vector::pop_back			
vector::push_back			

Table 2.1: Aspects of combinations in our dataset. Note that map takes two template arguments.

template initialization, compilers and optimization levels used. All combinations of STL type, template type, compiler and optimization level are present in the dataset. Note that map take two template arguments. The struct consits of two integers. For each STL type and template type there are ten C++ files, except for `vector::pop_back` only has six, because it takes no arguments. This amounts to 979 C++ source files and 7926 executable files, each with labels in json format. The source files include a direct usage of the method in main, the STL type in an array, the STL type in an array with the method called in a loop, the STL method called in an other function from the constructor/destructor, once passed by value, once passed by reference, the same is done with a longer call chain to the method usage and some variations with loops. For details see the code samples on GitHub.

Additionally to using this controlled dataset, in many scenarios it is useful to have a larger dataset with real world code. For this we use the Description2Code dataset [1]. It consist of competitive programming tasks and solutions in C++ and Python and is used primary for training machine learning code models [7]. Because of its size and the fact that uses a multitude of C++ versions on different platforms without providing labels for that, we do not provide labels for the full dataset.

2.3 Effect of Optimization

In this section we will investigate what modern compilers do when compiling C++ code that contains data structures from the STL. This problem can be investigated in a number of different scenarios such as availability of debugging symbols and run time type information (RTTI). Furthermore, different compilers use different optimizations methods invoked by a multitude of compiler flags. It should be noted, that while clang++ and g++ have mostly the same goals for the optimization compiler flags, the meaning in MSVC is different. In clang++ and, the `-O1` flag enables optimization options that are fast to execute and improve the overall performance, mostly with respect to runtime, while in MSVC, the `/O1` flag enables optimization options that are intended to reduce the size of the generated code. We will focus on clang++ and g++ in this

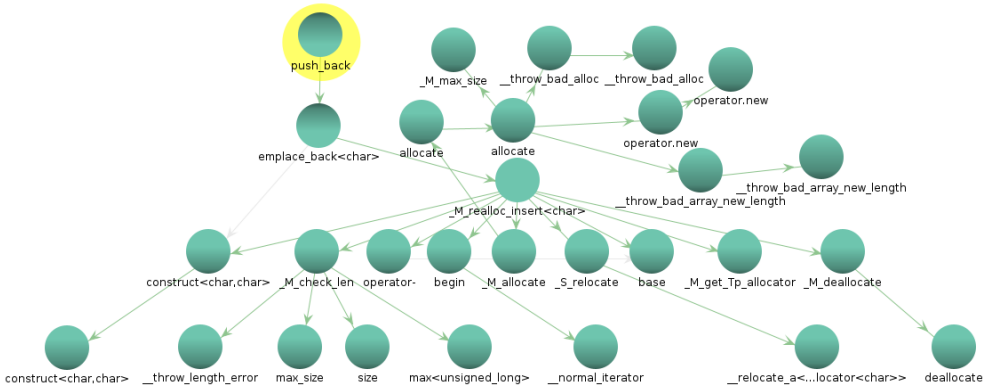

Figure 2.4: Call graph of the push_back operation on a vector of chars

report and always specify to which compiler the optimization flag is supplied, when there are notable differences. The cross product of compilers, compiler versions, compiler flags, STL implementations and STL usage in code is so large, that it is impossible to analyse all combinations. Instead we want to point out the most notable difference in the resulting binaries drawn from experience and reading compiler documentations.

Let us start the analysis by looking at the unoptimized case in clang++ at the example of the push_back operation on a vector of chars. The minimal example of first creating a vector and then pushing back, will generate a sizable call graph, that is shown in Figure 2.4. For the sake of comprehensibility, the function names are preserved here, however in many cases this debug information will be stripped from the binary. Furthermore, many of the functions contain little code and are mostly just delegating the call to the next function, which makes exploring the decompiler of current decompilers tedious. An insert into a map produces an even bigger call graph, making it difficult to show in this report. In general clang++ in `O0` will only inline functions if they are marked with `__attribute__((always_inline))`, by the default AlwaysInlinerPass. The C++ `inline` keyword is only a hint, that will not be followed by the compiler in this case. For g++ this is essentially the same, however while clang++ does not have any optimizations relevant to inlining with `-O0`, g++ will inline functions, with a body smaller than the function call overhead, due to the `fearly-inlining` flag being enabled by default.

With the `O1` optimization, g++ will do inlining of functions, that are called more than once, due to the `finline-functions-called-once` flag. This leads to the call graphs of the same call having different shapes in different `O1`-binaries. clang++ with `O1` will include the InlinerPass that does a sophisticated analysis to calculate the time/space trade-off for inlining. In practice this leads to more inlining by clang++, compared to g++. It should also be noted, that clang++ recently used completely different inlier pass, now called LegacyInliner, this leads to very different binaries for the same source code, but different clang++ versions. Furthermore, passes like InstCombinePass will combine instruction that stem from the inlined STL usage with surrounding unrelated code. This makes it impossible to label inlined code exactly in decompilation. Also, the presence of certain libc calls that one might consider as characteristic at first sight can be changed while

optimizing. For example the SROAPass pass of clang++ will replace allocations (i.e. memset or memcpy) with scalar SSA values. This might remove the operation or even split the operation into multiple.

For clang++ with `O2` there are no additional passes added or removed that are directly related to inlining. However g++ will apply the equivalent of the mentioned `O1` optimizations of clang++. In addition, there are some additional notable optimizations. `findirect-inlining` will also inline indirect calls where the target can be deduced by the pass at compile time. `fpartial-inlining` is particularly interesting, because it will inline parts of a function [6]. clang++ will do something similar with `O3` in the CallSiteSplittingPass. How the decompilation of programs using STL code looks at different optimization levels is described as a byproduct of the correctness evaluation of codeexplorer++ in Section 3.2.

Chapter 3

Evaluation

In this chapter we evaluate the compiler explorer patch and codeexplorer++. While the main focus of the compiler explorer evaluation is showing that it is faster than the current version and therefore can handle more programs, the focus of the codeexplorer++ evaluation is the correctness of the labels.

3.1 Compiler Explorer Patch

To evaluate the improvement to code explorer, we compare the current version on the main branch[1] with our patch based upon this version. Both versions run on the same machine. We randomly select 10 samples from our dataset, described in and test it with both versions. The patch is compared to the unpatched version, both run without a timeout, that the public version of compiler explorer has, to limit computational resources. Execution time, are compared by serially sending identical requests to both versions. It is important to disable the compiler explorer cash, to ensure independent execution. The results are shown in Table 3.1. Because we identified the amount of data transferred from the compiler process over stderr to the web server as a bottleneck, we also compare the amount of data transferred in Table 3.2. At `O3` the transferred data is reduced by a factor of 7.8 on average. For programs growing in the number of functions (assuming only a bound fraction of all optimization apply, which is a reasonable assumption for real world programs) the reduction in size is in $\mathcal{O}(n^2)$.

[1] commit hash: e7be3313

source file	O0 orig/patched (X)	O1 orig/patched (X)	O2 orig/patched (X)	O3 orig/patched (X)
main.cpp	6/4 (1.5)	9/5 (1.8)	9/6 (1.8)	9/6 (1.8)
map/insert/string_double/ by_reference.cpp	70/61 (1.1)	x/108 (∞)	x/129 (∞)	x/70 (∞)
map/at/struct_string/ by_reference.cpp	142/63 (2.3)	x/127 (∞)	x/153 (∞)	x/142 (∞)
map/insert/string_string/ array_single.cpp	78/48 (1.6)	x/103 (∞)	x/127 (∞)	x/78 (∞)
map/insert/string_struct/ in_loop.cpp	98/63 (1.6)	x/145 (∞)	x/170 (∞)	x/98 (∞)
map/insert/string_struct/ array.cpp	71/47 (1.5)	x/103 (∞)	x/119 (∞)	x/71 (∞)
vector/pop_back/char/ in_loop.cpp	69/46 (1.5)	x/89 (∞)	x/112 (∞)	x/69 (∞)
vector/push_back/string/ in_loop.cpp	23/26 (0.9)	97/58 (1.7)	97/72 (1.3)	x/23 (∞)
map/insert/struct_int/ function_call_chain_value.cpp	112/76 (1.5)	103/111(0.9)	103/129(0.79)	x/112 (∞)
map/erase/struct_struct/ function_call_chain_reference.cpp	74/48 (1.5)	x/76 (∞)	x/91 (∞)	x/74 (∞)
set/insert/string/ function_call_chain_by_reference.cpp	70/41 (1.7)	x/104 (∞)	x/119(∞)	x/70 (∞)

Table 3.1: Comparing execution time and calculating speedup between the original version of code explorer and our patch. Execution time in seconds. x means, that the nodejs server crashed, because the heap limit was reached.

source file	O0 orig/patched (X)	O1 orig/patched (X)	O2 orig/patched (X)	O3 orig/patched (X)
main.cpp	0.8 /0.1 (8.0)	5.7/0.6 (9.5)	5.9/0.6 (9.8)	5.9 /0.6 (9.8)
map/insert/string_double/ by_reference.cpp	50.4/6.4 (7.9)	283.8/38.2 (7.4)	292.1/38.0(7.7)	295.2/38.1 (7.7)
map/at/struct_string/ by_reference.cpp	39.9/4.5 (8.9)	190.9/26.5 (7.2)	202.8/27.7(7.3)	205.4/27.8 (7.4)
map/insert/string_string/ array_single.cpp	51.9/6.6 (7.9)	337.0/45.5 (7.4)	347.7/45.6(7.6)	351.5/45.7 (7.7)
map/insert/string_struct/ in_loop.cpp	39.2/4.4 (8.9)	227.9/31.3 (7.3)	235.3/31.4(7.5)	237.8/31.5 (7.5)
map/insert/string_struct/ array.cpp	38.4/4.4 (8.7)	220.6/30.2 (7.3)	227.2/30.3(7.5)	229.6/30.4 (7.6)
vector/pop_back/char/ in_loop.cpp	16.0/1.7 (9.4)	38.1/4.6 (8.3)	40.8/4.8 (8.5)	41.4 /4.9 (8.4)
vector/push_back/string/ in_loop.cpp	24.1/2.6 (9.3)	154.4/21.6 (7.1)	163.8/22.7(7.2)	165.4/22.7 (7.3)
map/insert/struct_int/ function_call_chain_value.cpp	55.8/6.1 (9.1)	208.1/26.4 (7.9)	220.1/27.5(8.0)	222.7/27.6 (8.1)
map/erase/struct_struct/ function_call_chain_reference.cpp	40.0/4.3 (9.3)	145.0/18.9 (7.7)	153.4/19.1(8.0)	156.7/19.7 (8.0)
set/insert/string/ function_call_chain_by_reference.cpp	35.9/4.1 (8.8)	246.8/35.7 (6.9)	257.1/36.0(7.1)	259.5/36.0 (7.2)

Table 3.2: Comparison between the original version of code explorer and our patch. Amount of data transferred over stderr in MiB and reduction of size.

3.2 codeexplorer++

As discussed in Chapter 1, there are no comparable tools to codeexplorer++, to compare against. Also note, that in codeexplorer++ currently not all aspects of code explorer are optimized, therefore the execution times are more of an upper bound for what is achievable. Furthermore, codeexplorer++ is designed to process large datasets. It runs in parallel without synchronization on different source code inputs, therefore execution times on a single source input are not to be added, but need to first be divided by the number of cores. When CodeQL as a code search backend is used, it benefits greatly from building a database containing all source code samples ahead of time. Overall there are a lot of feature combinations, with different performance behaviour. In the case of a single file from our dataset and using CodeQL to find STL calls, `O3` compilation, and visually printing inline information, which can be considered as the worst case, takes 8 seconds to find interesting calls and on average 2 seconds per function to find the virtual addresses that are associated with a STL call. Processing a single file with lizard on the other hand takes only 1.4ms on average on our dataset, because it only does parsing and does not require compilation.

More relevant than the exact runtime, is the correctness. Quantitatively evaluating correctness in this scenario is hard, because there exist no ground truth dataset. Additionally to the normal testing done when developing software, we qualitatively look at representative samples and manually check the output of codeexplorer++. We take a three samples, a map insert, a vector access in an array, and a set erase, and compile them with `O1` and `O3` flags in clang++. `O2` is not considered, as it is not very different in clang++. The analysis is done with the Ghidra plugin described in Section 2.2 The remainder of this section describes the methodology of the qualitative evaluation and list the findings of the evaluation. Manually reverse engineering includes creating structures for STL types and retying the relevant variables. One of the more complex memory layout is the one of a map, in this example one from int to struct of two ints. The used memory layout of this map is shown in Listing 3.1.

Listing 3.1: Structs required to type the map insert correctly

```
1  struct xy {
2      int x;
3      int y;
4  };
5  struct _Rb_tree_node_base {
6      enum _Rb_tree_color _M_color;
7      struct _Rb_tree_node* _M_parent;
8      struct _Rb_tree_node* _M_left;
9      struct _Rb_tree_node* _M_right;
10 };
11 struct _Rb_tree_node {
12     struct _Rb_tree_node_base _base;
13     struct kv_pair {
```

```
            int key;
            struct xy* value;
        } pair;
};
struct map {
    void* allocator;
    struct _Rb_tree_node_base _M_header;
    int _M_node_count;
};
```

Applying the structure shown in Listing 3.1 to a map variable makes the semantic of the inlined map constructor readable, the difference is shown in Listing 3.2

Listing 3.2: Difference between the map constrcutor decompilation before and after retyping with the map struct

```
// before
local_20 = local_30;
local_30[0] = 0;
local_28 = (undefined8 *)0x0;
local_10 = 0;
local_18 = local_20;
// after
local_38._M_header._M_left = (_Rb_tree_node *)&local_38._M_header;
local_38._M_header._M_color = _S_red;
local_38._M_header._M_parent = (_Rb_tree_node *)0x0;
local_38._40_8_ = 0;
local_38._M_header._M_right = local_38._M_header._M_left;
```

Associating the decompilation in Ghidra back to the disassembly shows that codeexplorer++ labeled the instructions correctly. Evaluating the rest of the samples at `O1` and `O3` does not reveal any wrong labels. If the same register assignment is used by both non-inlined code and inlined code, we observe the tendency of codeexplorer++ to give it the `MOV/LEA` the same label as the previous instruction. An notable compiler behaviour, to be aware of when reproducing this evaluation is the optimized size calculation for vectors. The divisions common when writing code with `sizeof`, get optimized to multiples and shifts, using the memory representations of `size_t` integers. For a `vector<int>` this becomes `(vec.end - vec.start) » 2`, a common way to devide by 4, the size of an int. For a `vector<char[5]>` the inlined size calculation becomes `((vec.end - vec.start) * 0xCCCCCCCCCCCCCCCD) » 2`, a maybe less obvious way to devide by 5. Another notable applied in this evaluation is to retype C++ strings with a custom structure in Ghidra, and understand their behaviour, laid out in Listing 3.3. A string is then only a type alias of a basic_string.

Listing 3.3: String structure to import into Ghidra

```
struct basic_string {
    char *begin_;
    int size_;
    union {
        int capacity_; // is >=16B
        char sso_buffer[16]; // if < 16B
    };
};
```

Chapter 4

Exploring Solutions

This chapter describes possible research directions to move further towards better STL code decompilation. One approach that appears to be very promising in the future is end to end machine translation using transformers. End to end machine translation for decompilation does only require pairs of source code and low level code, without handcrafting lifting rules. To low level code does not necessarily be disassembly but could also work with the intermediate language of a decompiler such as, Binary Ninja LLIL, Ghidra pcode or Hex-Rays microcode. These intermediate languages provide an architecture independent representation without doing decompilation already, such as Binary Ninja MLIL. Current end to end decompilation research does not consider C++ and is still prone to produce incorrect code in many situations [4]. Beyond the standard transformer model used in [4], we conjecture that hierarchical summarization, similar to the work of Zhang et al. [15] for natural languages, for used definition can significantly improve decompilation results.

Apart from machine translation, improvements to STL decompilation will likely come from improvements to the high level analysis of current decompilers. Beyond the mere recognition and retyping STL a next step is outlining parts of functions. In 2022 Hex-Rays decompiler and Binary Ninja gained support for inlining, [12, 14] but there is no decompiler support outlining. Likely this would be implemented by assembly rewriting: nopping out code and replacing it with a call in a new virtual address space where the instructions are moved and function prologue and epilogue are added. Given a continuous section of code, analysis of used and changed variables is required to infer parameters and return type. However to fully solve the problem the solution must also consider inlined code interleaved with original code and partial inlining, as done by `fpartial-inlining` in gcc.

At first sight identifying STL code might seem equivalent to the code semantic equality problem that, despite to be known to be undecidable, it is subject of research of extensive research for finding good approximations and solution for special cases [3]. However, there are a couple of crucial differences. Optimization can remove parts of inline code that is known to be

not executed, e.g. it is guaranteed that the error case is never thrown. Again, partial inlining is a problem. Also, there must be search for semantically equivalent code on a instruction level and sometimes even on a sub instruction level, cf. InstCombinePass in Section 2.3. There is neither an indication for the beginning inlined function nor for end and functions can be inlined multiple times. A naive search with current approaches for semantic equality would be inefficient, and the false positives would add up. All these effects break the assumptions of code equality approaches. Fuzzy match approaches will have a hard time distinguishing user code that uses STL code from the STL code itself, as a user might implement a function that consists of a small check plus the STL call.

We believe that in the `O0` analysing the call graph is a very promising approach. As shown in Figure 2.4, these call graphs have little to no inlining and can be assumed to be characteristic for an STL function, at least if the template types are either primitive types or fixed. An implementation might create these call graphs while decompiling with the template of specific types when a more simple check, such as checking for exception strings triggers. Comparing these Graphs can either be done similar to generating signatures for molecules with a Graph Neural Network [0], with handcrafted graph query rules in a language Joern (similar to CodeQL but does not need compilation), or by full inlining of the call graph. We assume full inlining is possible in almost all cases, as all call graphs of STL methods, we investigated, are acyclic. After inlining, methods for text comparison can be used. An interesting further improvement for this approach is automatically generating the structures. This is not a simple code search problem in the STL code, because there are a lot of environment specific marco code, especially fields surrounded with `ifdef`. Furthermore, the current correct datatypes need to be inserted into the templates. Template processing is not done by the macro preprocessor, but is a step during compilation. Also, C classes and notation for struts and enums must be converted to C structs and enums, to be importable by current decompilers.

Chapter 5

Conclusion

In this report, we introduced new tools for tracking how high level language features affect low level code. Additionally, we provide a labeled dataset with common STL types and methods, that can be used for evaluating STL decompilation improvements in the future. We also release the tool for creating the dataset. It can be repurposed for creating other datasets with labeled disassembly, particularly as it relates to inlining. Our patch to compiler explorer advances the state of the art. It can now decompile non trivial C++ programs and reduces the amount of data that needs to be send from the compiler process to the application by a factor of 7.8 on average at `O3` on the randomly selected test data.

Furthermore, we describe the challenges of STL decompilation in detail for g++ and clang++, pointing out what optimization passes at what optimization level are responsible for the resulting low level code. In the light of these challenges, we describe directions of future research.

Bibliography

[1] Ethan Caballero, . OpenAI, and Ilya Sutskever. *Description2Code Dataset*. Aug. 2016. DOI: 10.5281/zenodo.5665051. URL: https://github.com/ethancaballero/description2code.

[2] Istvan Haller, Asia Slowinska, and Herbert Bos. "MemPick: High-level data structure detection in C/C++ binaries". In: *2013 20th Working Conference on Reverse Engineering (WCRE)*. 2013 20th Working Conference on Reverse Engineering (WCRE). ISSN: 2375-5369. Oct. 2013, pp. 32–41. DOI: 10.1109/WCRE.2013.6671278.

[3] Irfan Ul Haq and Juan Caballero. "A survey of binary code similarity". In: *ACM Computing Surveys (CSUR)* 54.3 (2021), pp. 1–38.

[4] Iman Hosseini and Brendan Dolan-Gavitt. "Beyond the C: Retargetable Decompilation using Neural Machine Translation". In: *arXiv preprint arXiv:2212.08950* (2022).

[5] Changhee Jung and Nathan Clark. "DDT: design and evaluation of a dynamic program analysis for optimizing data structure usage". In: *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture - Micro-42*. the 42nd Annual IEEE/ACM International Symposium. New York, New York: ACM Press, 2009, p. 56. ISBN: 978-1-60558-798-1. DOI: 10.1145/1669112.1669122. URL: http://portal.acm.org/citation.cfm?doid=1669112.1669122 (visited on 12/25/2022).

[6] Jun-Pyo Lee, Jae-Jin Kim, Soo-Mook Moon, and Suhyun Kim. "Aggressive function splitting for partial inlining". In: *2011 15th Workshop on Interaction between Compilers and Computer Architectures*. IEEE. 2011, pp. 80–86.

[7] Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, et al. "Competition-level code generation with alphacode". In: *Science* 378.6624 (2022), pp. 1092–1097.

[8] Rolf Rolles. *Automation in Reverse Engineering C++ STL/Template Code*. Möbius Strip Reverse Engineering. Jan. 25, 2022. URL: https://www.msreverseengineering.com/blog/2021/9/21/automation-in-reverse-engineering-c-template-code (visited on 11/07/2022).

[9] Thomas Rupprecht, Xi Chen, David H. White, Jan H. Boockmann, Gerald Lüttgen, and Herbert Bos. "DSIbin: Identifying dynamic data structures in C/C++ binaries". In: *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2017

32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). Oct. 2017, pp. 331–341. DOI: 10.1109/ASE.2017.8115646.

[10] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. “Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables”. In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’18: 2018 ACM SIGSAC Conference on Computer and Communications Security. Toronto Canada: ACM, Oct. 15, 2018, pp. 426–441. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243793. URL: https://dl.acm.org/doi/10.1145/3243734.3243793 (visited on 11/05/2022).

[11] Igor Skochinsky. *Answer to "Find the C++ STL functions in a binary"*. Reverse Engineering Stack Exchange. Mar. 17, 2014. URL: https://reverseengineering.stackexchange.com/a/3890/42404 (visited on 11/23/2022).

[12] Igor Skochinsky. *Igor’s tip of the week #106: Outlined functions – Hex Rays*. Jan. 2022. URL: https://hex-rays.com/blog/igors-tip-of-the-week-106-outlined-functions/ (visited on 01/01/2023).

[13] *Stack Overflow Developer Survey 2022*. Stack Overflow. URL: https://survey.stackoverflow.co/2022/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2022 (visited on 12/24/2022).

[0] Ruoxi Sun, Hanjun Dai, and Adams Wei Yu, eds. *Rethinking of graph pretraining on molecular representation*. 2022. URL: https://arxiv.org/abs/2207.06010.

[14] Jordan Wiens. *Binary Ninja - 3.0 The Next Chapter*. Binary Ninja. Sept. 2022. URL: https://binary.ninja/2022/01/27/3.0-the-next-chapter.html (visited on 01/01/2023).

[15] Xingxing Zhang, Furu Wei, and Ming Zhou. “HIBERT: Document Level Pre-training of Hierarchical Bidirectional Transformers for Document Summarization”. In: *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*. Florence, Italy: Association for Computational Linguistics, July 2019, pp. 5059–5069. DOI: 10.18653/v1/P19-1499. URL: https://aclanthology.org/P19-1499.