École Polytechnique Fédérale de Lausanne

Recovering type information from compiled binaries to aid in instrumentation

by Louis Merlin

Master Thesis

Approved by the Examining Committee:

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Damian Pfammatter
External Expert

Antony Vennard
Thesis Supervisor

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

July 12, 2021

Nobody tosses a dwarf !
— Gimli, son of Glóin

TODO Dedication

Acknowledgments

TODO Acknowledments

Lausanne, July 12, 2021 Louis Merlin

Abstract

The abstract serves as an executive summary of your project. Your abstract should cover at least the following topics, 1-2 sentences for each: what area you are in, the problem you focus on, why existing work is insufficient, what the high-level intuition of your work is, maybe a neat design or implementation decision, and key results of your evaluation.

Contents

Chapter 1

Introduction

Work on C++ began in 1979, as a "C with classes" [12]. Since then, the language has grown in popularity, and even surpassed C itself [10]. Examples of well-known C++ projects include the zoom [2] conferencing software, the gold linker [13] or [TODO: find another "well-known" example]. Nevertheless, reverse-engineering efforts have been focused towards C binaries, because analysis methods found this way will often work on C++ binaries too.

This has meant reverse engineering tools like ghidra [7], IDA Pro [9] or Binary Ninja [1] have treated non-C binaries as second-class citizen. These tools will often show C++ specific features as passing comments, failing to show the real implications of a try/catch block or a polymorphic class.

The blame can mostly be put on the complexity of C++ when compared to C. Whereas C translates quite naturaly to assembly, abstractions specific to C++ require more work and complexity to be translated to asm. This also leads to important information being lost from C++ source code to binary, but also certain information remaining.

The recent RetroWrite [4] project by the HexHive lab is a static rewriting tool for x86_64 position-independent binaries. It enables the instrumentation of projects when we do not have access to the source code. This can include a legacy project, a closed-source product or even malware.

In this thesis we would like to present the **dis-cover** [5] static analysis tool, as well as improvements made to RetroWrite to support C++ binaries.

The dis-cover tool is able to extract information from a C++ binary, and re-inject it as debug information using the DWARF format into the binary. This enables other debugging tools to see and display this information.

[IF WE SUCCEEDED] In this thesis we would like to show how we brought C++ support to RetroWrite, and what research opportunities this will create.

[IF WE FAILED] In this thesis we will detail how we tried to bring C++ support to RetroWrite, and what remains to be done for the implementation to work.

Chapter 2

Background

2.1 C++ and polymorphism

The C++ programming language implements polymorphism. This feature enables complex code logic that can comply with external business logic for example. Polymorphic classes are defined by having at least one virtual method, which is inheritable and overridable. With this polymorphism comes type conversion, and more interestingly for us the **dynamic_cast** [3] expression. Dynamic casting is a safe kind of type conversion. It only will successfuly cast if the value is of the referenced type or a base type of that type. In order to achieve that, the system must have some kind of information about the object's data type at runtime. This information is called Run-Time Type Information (**RTTI**), and is stored in the binary if it is needed. In certain situations, if there are polymorphic classes but certain features are not used (casting, inheritance logic), the classes are abstracted away at compile time and the RTTIs do not exist. We will go into more details about the implementation of C++ RTTI in later chapters.

2.2 DWARF debugging standard

DWARF is the debugging standard used widly in conjunction with executable ELF files. It is often included in these ELF files in the **.debug_info** section (and other related sections). This debug information is made up of Debugging Information Entries (**DIEs**). These entries can contain information about variable names, method definitions, the compilation process, or more importantly for us, class names and class inheritance.

2.3 RetroWrite

Chapter 3

Design

Introduce and discuss the design decisions that you made during this project. Highlight why individual decisions are important and/or necessary. Discuss how the design fits together.

This section is usually 5-10 pages.

Class recovery from compiled C++ binaries has been done multiple ways before: there are plugins for popular debugging tools, like ida_gcc_rtti [6] for IDA Pro, or Ghidra-Cpp-Class-Analyzer [11] for Ghidra; there are academic projects such as MARX [8] that rely on heuristics around VTables to find classes and inheritance information. The MARX project can recover classes when no RTTI is present (in the Chrome project for example, where the **fno-rrti** compilation flag is used), but not with 100% accuracy.

For this project we decided to focus on RTTIs which, as mentioned in the Background chapter, are available for a class hierarchy tree when there is at least one virtual method implemented by one of the classes. We decided to test if RTTIs occured often in the wild : we proceed to download every Debian package that listed C++ as a dependency (get the list with **apt-cache rdepends libgcc1** on a debian machine). Out of around 80'000 packages, 5827 of them list C++ as a dependency. Out of those, we were able to extract classes from 3194 (54%). We will share more details about this experiment in the Evaluation chapter.

We also considered extracting information about exceptions, which is another big feature that differenciates C++ from C. We decided not to focus on this feature, as it did not seem to bring as much to the project considering the evaluated development time.

Chapter 4

Implementation

The implementation covers some of the implementation details of your project. This is not intended to be a low level description of every line of code that you wrote but covers the implementation aspects of the projects.

This section is usually 3-5 pages.

Chapter 5

Evaluation

In the evaluation you convince the reader that your design works as intended. Describe the evaluation setup, the designed experiments, and how the experiments showcase the individual points you want to prove.

This section is usually 5-10 pages.

As we have shown before, out of the 5827 debian packages that list C++ as a dependency, we were able to extract classes from 3194 (54%). The total number of classes we found is 960'188, with 39% of them unique across all packages (unique name in unique tree).

Chapter 6

Related Work

The related work section covers closely related work. Here you can highlight the related work, how it solved the problem, and why it solved a different problem. Do not play down the importance of related work, all of these systems have been published and evaluated! Say what is different and how you overcome some of the weaknesses of related work by discussing the trade-offs. Stay positive!

This section is usually 3-5 pages.

Chapter 7

Conclusion

In the conclusion you repeat the main result and finalize the discussion of your project. Mention the core results and why as well as how your system advances the status quo.

Bibliography

[1] Vector 35. *Binary Ninja*. https://binary.ninja/. Accessed: 2021-06-06.

[2] Zoom Video Communications. *Zoom Cloud Meetings*. https://zoom.us/. Accessed: 2021-06-09.

[3] cppreference.com. *dynamic_cast conversion*. https://en.cppreference.com/w/cpp/language/dynamic_cast. Accessed: 2021-06-10.

[4] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. "RetroWrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization". In: *IEEE International Symposium on Security and Privacy*. 2020.

[5] Louis Merlin. *dis-cover*. https://github.com/HexHive/dis-cover. Accessed: 2021-06-06.

[6] mwl4. *ida_gcc_rtti*. https://github.com/mwl4/ida_gcc_rtti. Accessed: 2021-06-12.

[7] NSA. *ghidra*. https://ghidra-sre.org/. Accessed: 2021-06-06.

[8] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. "MARX: Uncovering Class Hierarchies in C++ Programs". In: *NDSS Symposium*. 2017.

[9] Hex Rays. *IDA Pro*. https://hex-rays.com/IDA-pro/. Accessed: 2021-06-06.

[10] stackoverflow. *Most Popular Technologies*. https://insights.stackoverflow.com/survey/2020#most-popular-technologies. Accessed: 2021-06-05.

[11] Andrew Strelsky. *Ghidra C++ Class and Run Time Type Information Analyzer*. https://github.com/astrelsky/Ghidra-Cpp-Class-Analyzer. Accessed: 2021-06-12.

[12] Bjarne Stroustrup. *When was C++ invented?* https://www.stroustrup.com/bs_faq.html#invention. Accessed: 2021-06-05.

[13] Ian Lance Taylor. *Gold Linker*. https://en.wikipedia.org/wiki/Gold_(linker). Accessed: 2021-06-09.