



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 16, 2021

Nobody tosses a dwarf!
— Gimli, son of Glóin

TODO Dedication

Acknowledgments

TODO Acknowledgments

Lausanne, July 16, 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

Acknowledgments	1
Abstract	2
1 Introduction	4
2 Background	6
2.1 C++ and polymorphism	6
2.2 DWARF debugging standard	7
2.3 RetroWrite	8
3 Design	9
4 Implementation	11
4.1 Finding RTTIs	11
4.2 Creating DWARF data	11
5 Evaluation	13
6 Related Work	14
7 Conclusion	15
Bibliography	16

Chapter 1

Introduction

Work on C++ began in 1979, as a "C with classes" [19]. Since then, the language has grown in popularity, and even surpassed C itself [17]. Examples of well-known C++ projects include the zoom [4] conferencing software, the gold linker [20] 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 [12], IDA Pro [15] 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 naturally 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 [7] 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** [10] 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** [5] expression. Dynamic casting is a safe kind of type conversion. It only will successfully 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 run time.

This is where Run-Time Type Information (**RTTI**) comes into the picture. The system will use this RTTI to infer type inheritance for dynamic casting. We will now go into implementation details of RTTIs and VTables, which point to them.

To make RTTIs appear in your C++ binary, you will to define classes that inherit from each other, as well as at least one virtual method in one of these classes. Figure 2.1 shows an example of such classes.

```
class Animal {  
public:  
    virtual void speak() {}  
};  
  
class Dog {  
public:  
    virtual void speak() { cout << "Woof" << endl; }  
}
```

Figure 2.1: Polymorphic classes in C++

For RTTIs to appear in your C++ binary, you will also need to instantiate these classes, and have some run time logic to make the binary non-deterministic [this not the right word I think, I mean to say that the binary depends on some input for its class-instantiating logic]. For example, a conditional dynamic cast between the two classes we defined in Figure 2.1. If you do not do these things, the class logic will be abstracted away by the compiler for optimization reasons.

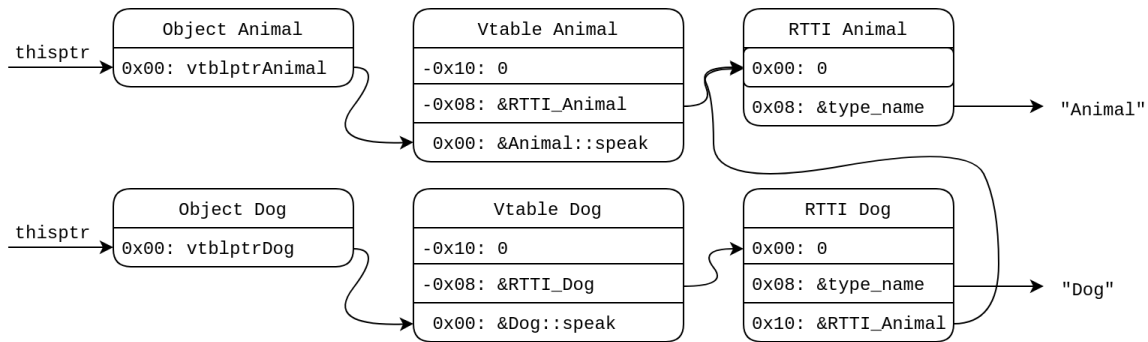


Figure 2.2: Overview of an example of VTables and RTTI in memory

The structure of VTables and RTTIs is detailed in Figure 2.2. All of this is defined in the Itanium C++ ABI [8]. An instance of a virtual class **Animal** will contain the **vtblptrAnimal**, a pointer to the virtual table (**VTable**) for the **Animal** class. This VTable will contain pointers to the virtual methods of the class, which are inherited and thus do not have to be duplicated for each class, they are simply pointed to. The first value preceding the VTable is a pointer to the RTTI of the class. The process finds out the class inheritance of a class instance by following this pointer.

The RTTI itself is composed of an offset (used for complex class inheritance structures) as well as a pointer to the type name (this type name is not removed by simple stripping of the binary, like with **objcopy -strip-all**). This name is mangled using C++ mangling, and can trivially be de-mangled. This name is what we use to uniquely identify a class in a project (and make sure we have no duplicates). The next values of the RTTI are pointers to the RTTIs of the parent classes. See Figure 2.2 for an example, with the **Dog** RTTI containing a pointer to the **Animal** RTTI.

2.2 DWARF debugging standard

DWARF is the debugging standard used widely 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

Class recovery from compiled C++ binaries has been done multiple ways before: there are plugins for popular debugging tools, like `ida_gcc_rtti` [11] for IDA Pro, or Ghidra-Cpp-Class-Analyzer [18] for Ghidra; there are academic projects such as MARX [14] 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-rtti** 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 occurred 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 extracted 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 differentiates 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. We did have to study and work with exceptions and exception handling frames for the augmentation of RetroWrite (see the end of the Implementation chapter).

Next comes the question : what should we do with all of this information ? What would be the most useful format to output the data in ? This is where DWARF [3] comes in the picture. DWARF is the debugging standard for programs. It is mostly used by developers trying to understand where their implementation fails, and by reverse engineers to get a better understanding of how a program was conceived (although DWARF information is usually stripped from proprietary software). This kind of debug data is mostly found in C and C++ projects. By having DWARF data as an output, the information would become readable by most modern reversing tools.

There is a current push for DWARF to become the lingua franca for reverse engineering tools, lead by researchers like Dr. Sergey Bratus [6] from DARPA. He has published a paper in 2011 where he was able to exploit certain features of DWARF to control program execution : Exploiting the hard-working DWARF [13].

With dis-cover, we are able to inject information in the debug and symbol sections of the binary, creating a new ELF file with all of this useful info included. We will go more into the implementation details in the next chapter.

Another focus of this project was the augmentation of RetroWrite for C++ capabilities. Today, RetroWrite supports the reversing of x86_64 position-independent binaries. There was also work to augment RetroWrite to support kernel code [16] and arm_64 [2]. We aimed to add C++ capabilities for x86_64 binaries only, as it seems like the biggest target.

Another potential use of the class information is to implement defenses around type confusion, working with the findings from the HexType [9] paper. We decided not to focus on this for our project, as it would have been out of the scope of a 6-months master thesis.

One of the defining properties of RetroWrite is that it does not use heuristics (and thus does not have probabilistic features and failures). Dis-cover has the same no-heuristics property, and thus could be used by the RetroWrite project without breaking the property.

Chapter 4

Implementation

We decided to write a python module for this project, as the python ecosystem has great reverse engineering packages, and because the other projects from the HexHive lab are usually python modules.

4.1 Finding RTTIs

The first thing dis-cover does is find ELF sections where the VTables and RTTIs could be hiding. This is usually **.rodata** (read-only data), but could sometimes also be related sections like **.data.rel.ro**, **.data.rel.ro.local**, or **.rdata**. [TODO see if we can just look in .rodata] As per the Itanium C++ ABI, the base VTable of a class will contain an "offset-to-top", which will be 0 in the primary base virtual table's case, followed by a pointer to an RTTI. We simply have to pattern-match for zeroes directly followed by a pointer to another part of the read-only data sections, and we have a potential RTTI pointer.

To check whether we have found an RTTI, we assert that the next value is a pointer to a string located in a read-only data section. If it is, we can extract it, demangle it, and we have a class and a name. The next values in the RTTI are pointers to the RTTIs the class inherits from. We can go through them and parse them if we have not already, to add this inheritance information to the original class.

This algorithm is $O(n)$, as adding a class only adds one more value to parse.

4.2 Creating DWARF data

Next, we want to add that information to the debug sections of a new ELF file.

In order to write DWARF data, the first step is defining the types we will be using, and their fields. This is done by writing bytes in the **.debug_abbrev** section. For example, we create an abbrev of type **class_type**, which has a **name**, and can have sub-field (children). Then, we create the abbrev of type **inheritance**, which has a **type** (a reference to the parent type).

We can then populate the **.debug_info** with classes and their inheritance data. DWARF data takes the form of a tree of values. We have to create a **compile_unit** value at the root, and then the branches will be **class_types**. These **class_types** will themselves have as children **inheritance** values if the class inherits from another class. Figure 4.1 shows a very simple example of this, with two class types and one inheriting from the other.

< 1><0x0000001a>	DW_TAG_class_type	
	DW_AT_containing_type	<0x0000001a>
	DW_AT_calling_convention	DW_CC_pass_by_reference
	DW_AT_name	Shape
< 1><0x00000026>	DW_AT_byte_size	0x00000008
	DW_TAG_class_type	
	DW_AT_containing_type	<0x00000026>
	DW_AT_calling_convention	DW_CC_pass_by_reference
< 2><0x00000031>	DW_AT_name	Rectangle
	DW_AT_byte_size	0x00000008
	DW_TAG_inheritance	
	DW_AT_type	<0x0000001a>

Figure 4.1: Extract of a dwarfdump output showing simple inheritance

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] Luca Di Bartolomeo. *ArmWrestling: efficient binaryrewriting for ARM*. <http://hexhive.epfl.ch/theses/20-dibartolomeo-thesis.pdf>. Accessed: 2021-06-15.
- [3] DWARF Standards Committee. *The DWARF Debugging Standard*. <http://www.dwarfstd.org/>. Accessed: 2021-06-14.
- [4] Zoom Video Communications. *Zoom Cloud Meetings*. <https://zoom.us/>. Accessed: 2021-06-09.
- [5] cppreference.com. *dynamic_cast conversion*. https://en.cppreference.com/w/cpp/language/dynamic_cast. Accessed: 2021-06-10.
- [6] DARPA. *Dr. Sergey Bratus*. <https://www.darpa.mil/staff/dr-sergey-bratus>. Accessed: 2021-06-15.
- [7] 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.
- [8] *Itanium C++ ABI*. <https://itanium-cxx-abi.github.io/cxx-abi/abi.html>. Accessed: 2021-06-15.
- [9] Yuseok Jeon, Priyam Biswas, Scott Carr, Byoungyoung Lee, and Mathias Payer. “HexType: Efficient Detection of Type Confusion Errors for C++”. In: *ACM Conference on Computer and Communication Security*. 2017.
- [10] Louis Merlin. *dis-cover*. <https://github.com/HexHive/dis-cover>. Accessed: 2021-06-06.
- [11] mw14. *ida_gcc_rtti*. https://github.com/mw14/ida_gcc_rtti. Accessed: 2021-06-12.
- [12] NSA. *ghidra*. <https://ghidra-sre.org/>. Accessed: 2021-06-06.
- [13] James Oakley and Sergey Bratus. “Exploiting the hard-working DWARF: Trojan and Exploit Techniques With No Native Executable Code”. In: *Usenix Workshop on Offensive Technologies*. 2011.
- [14] 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: *Network and Distributed System Security Symposium*. 2017.

- [15] Hex Rays. *IDA Pro*. <https://hex-rays.com/IDA-pro/>. Accessed: 2021-06-06.
- [16] Matteo Rizzo. *Hardening and Testing Privileged Code through Binary Rewriting*. <http://hexhive.epfl.ch/theses/19-rizzo-thesis.pdf>. Accessed: 2021-06-15.
- [17] stackoverflow. *Most Popular Technologies*. <https://insights.stackoverflow.com/survey/2020#most-popular-technologies>. Accessed: 2021-06-05.
- [18] Andrew Strelsky. *Ghidra C++ Class and Run Time Type Information Analyzer*. <https://github.com/astrelsky/Ghidra-Cpp-Class-Analyzer>. Accessed: 2021-06-12.
- [19] Bjarne Stroustrup. *When was C++ invented?* https://www.stroustrup.com/bs_faq.html#invention. Accessed: 2021-06-05.
- [20] Ian Lance Taylor. *Gold Linker*. [https://en.wikipedia.org/wiki/Gold_\(linker\)](https://en.wikipedia.org/wiki/Gold_(linker)). Accessed: 2021-06-09.