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

August 27, 2021

Nobody tosses a dwarf!
— Gimli, son of Glóin

Dedicated to my parents, Sophie and Jean-Marc, my three little sisters Victoire, Colombe, and Céleste, as well as my dear Violaine. Thank you for supporting and helping me these last two years, and all of the years before that !

Acknowledgments

I want to thank my advisor, Prof. Mathias Payer, for his guidance and motivation, and for coming up with this project. Thank you also for writing this thesis' template [30].
I want to thank my supervisor Antony Vennard, for all of the hard work he accomplished with me during our pair coding sessions, and for the continuous feedback he was able to provide.
I want to thank my external supervisor Damian Pfammatter from armasuisse for his insightful comments and for accompanying me in this endeavor. This Research is supported by armasuisse Science and Technology.
I also want to thanks my family for their emotional support during this project. Special thanks go to my S.O. Violaine, who was ever present for me, and incredibly supportive throughout this project.
Finally, I would like to thank the polygl0ts and 0rganizers CTF teams. I was extremely lucky to participate in the DEF CON 29 CTF in Las Vegas during my thesis. This was an amazing experience.

Lausanne, August 27, 2021 Louis Merlin

Abstract

The analysis of closed-source C++ programs is an arduous task. C++ adds several powerful functionalities to the C language, which leads to complex binaries.

Knowledge of class types and hierarchies can prove valuable during the reverse engineering process. Fortunately, some of that information can be recovered from stripped binaries due to the way polymorphism is implemented in C++ compilers.

With *dis-cover*, the tool we developed, we extract *class information* and the *hierarchy tree* from a stripped C++ binary, and write DWARF debug information and symbols for the later use of that knowledge by any existing debugging tool. Through this method, we aim to demonstrate an open and collaborative way of writing static analysis tools.

By researching in detail the way C++ exceptions are implemented, we also augment the capabilities of the *RetroWrite* static rewriting tool to support position-independent C++ binaries. We also show how the class information extracted by dis-cover could be used to improve the instrumentation done by RetroWrite.

By utilizing *dis-cover*, we successfully extract useful information from closed-source applications like Zoom.

We also detail the improvements now needed for *RetroWrite* to support rewriting closed-source stripped position-independent C++ binaries.

Contents

Chapter 1

Introduction

Work on C++ began in 1979, as a "C with classes" [35]. Since then, the language has grown in popularity, and even surpassed C itself [33]. Examples of well-known C++ projects include modern browsers like Chrome [18] and Firefox [24], the Qt Framework [7], or the Zoom [6] conferencing software. Nevertheless, most reverse-engineering efforts have been focused towards C binaries, because analysis methods found this way are simpler and will often work on C++ binaries too.

This has meant reverse engineering tools like Ghidra [26], IDA Pro [31], radare2 [28] 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 complexity of C++ has also lead binary rewriters like RetroWrite [11] to focus on C binaries only, leaving a whole ecosystem of binaries unstudied.

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 assembly. This also leads to important information being lost from C++ source code to binary, but also certain information remaining, like run-time type information.

In this thesis we would like to present the *dis-cover* [23] static analysis tool. This tool extracts class hierarchy information from a C++ binary, and re-injects it into the binary as debug information using the DWARF format. This enables other debugging tools to see and display this information.

This class hierarchy information is included in the binary for functionality reasons, and is even kept after basic stripping of the program. Several large closed-source projects, like the Zoom conferencing software, have several thousands of classes linked to each other in complex class hierarchy trees. Having that information easily accessible would help reverse engineering efforts.

The recent RetroWrite [11] project by the HexHive lab is a static rewriting tool for unstripped x86_64 position-independent binaries. It enables the instrumentation of projects when we do not have access to the source code. Instrumentation is the process of adding instructions to

a binary, to add additional security checks for example. The target can be a legacy project, a closed-source product or even malware. In this thesis we will also detail our improvements to RetroWrite to support unstripped position-independent *C++* binaries, and the possibilities opened by this feature for future researchers.

Chapter 2

Background

2.1 Executable and Linkable Format (ELF)

The common standard *Executable and Linkable Format* (ELF) format is mostly used as executable files, object code and shared libraries. It is the current standard format for executable files on UNIX systems [22].

An ELF file contains a program header table, which describes memory segments (e.g. "this is read-only data", "this is executable instructions"). It also usually contains a section header table, describing sections (their name, size, offset and some flags). Memory segments contain information that is used at runtime, while sections contain relocation and linking information.

The rest of an ELF file is taken up by different sections. Famous sections include `.text` (where the instructions are), `.data` (where you will see global tables and other variables), `.rodata` (you will find strings in here) and `.eh_frame` (where exception unwinding information is stored for C++ binaries).

2.2 Position-Independent Code (PIC) and Executable (PIE)

Position-independence for code is a property that guarantees that code will execute correctly no matter the base address of the body of code. This is made possible by computing *relative positions* instead of using base addresses, as well as a feature called *relocations*.

PIC is most often found for shared libraries, so that they can be loaded and shared by any number of processes without breaking offsets, or even thinking about them. PIE binaries are seen as more secure because they allow for easy inserting of security measures like address space layout randomization [19]. We will talk more about PIE in section 2.5.

2.3 C++ features

2.3.1 Polymorphism

The C++ programming language implements polymorphism. This feature enables complex code logic that can comply with external business logic. Here we will introduce the example that will follow us throughout this thesis : in Figure 2.1 we define two abstract base classes *Animal* and *Feline*, and the two classes *Cat* (that inherits from Animal and Feline) and *Dog* (that inherits from Animal). In a large code base, with multiple teams of programmers working on different features, an Animal could be passed around without caring about whether it was a Cat, a Dog or any other species. This Animal is sure to overwrite the *speak* method and have whatever other properties and methods Animal has defined. This logic is verified by the compiler.

This polymorphism feature also enables re-use of functionality by inheriting parent classes. If you want to implement a Lemur and your Animal class already has an implementation of the eat method, you don't need a lemur-specific implementation if appropriate.

Polymorphic classes are defined by having at least one virtual method, which is inheritable and overridable. With this polymorphism comes type conversion, of which there are two kinds. The first, *static_cast* [9], will check that the conversion is upcasting at compile time, but does not do any runtime checks to verify the validity of the conversion. An example of upcasting is converting a pointer of a *Cat* to a pointer of an *Animal*. The fact that a class is polymorphic will have no impact on this type of casting. The second is the more interesting one for us. It is the *dynamic_cast* [8] expression. Dynamic casting is a safe kind of type conversion that can handle downcasting and sidecasting. An example of downcasting is converting a pointer of an *Animal* to a pointer of a *Cat*. This is not guaranteed to work. An example of sidecasting is converting a pointer of an *Animal* to a pointer of a *Feline*. This is also not guaranteed to work (if the Animal pointer was that of a Dog for example). In order to achieve a dynamic cast, the system must have some kind of information about the object's data type at runtime.

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 *virtual tables* (vtables), which point to them.

To make vtables and RTTIs appear in your C++ binary, you will need 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. You will also need to instantiate these classes, and have some logic to make the class logic runtime dependent. This is extremely common in practice, as it is the core motivation for polymorphism in the first place. For an example, see the conditional creation of the my_animal variable, as well as dynamic cast between Animal and Feline we defined in Figure 2.1. If you do not do one of these things, the class logic is likely to be abstracted away by the compiler for optimization reasons.

```
#include <iostream>
#include <string.h>

using namespace std;


class Animal {
public:
  virtual void speak() {}
};

class Feline {
public:
  virtual void retract_claws() {}
};

class Cat : public Animal, public Feline {
public:
  virtual void speak() { cout << "Purr" << endl; }
};

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


int main (int argc, char *argv[]) {
  Animal *my_animal = nullptr;

  if (argc == 2) {

    // We can instantiate an instance of a subclass into a pointer of type
    // Animal*.
    if (strcmp("cat", argv[1]) == 0) {
      my_animal = new Cat();
    } else if (strcmp("dog", argv[1]) == 0) {
      my_animal = new Dog();
    } else {
      return 1;
    }

    my_animal->speak();

    // We can dynamic_cast an Animal* into a Feline*, even though one does not
    // inherit from the other (but Cat inherits from both).
    Feline *my_feline = dynamic_cast<Feline*>(my_animal);
    if (my_feline) {
      my_feline->retract_claws();
      cout << "This feline could retract its claws" << endl;
    }
  }

  return 0;
}
```

Figure 2.1: Polymorphic classes in C++

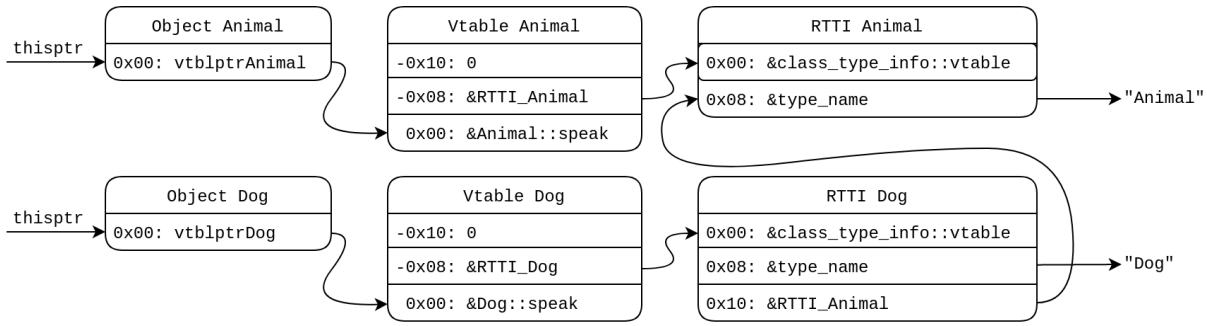

Figure 2.2: Overview of an example of vtables and RTTI in memory

The structure of vtables and RTTIs is detailed in Figure 2.2. All of this is defined in the Itanium C++ ABI [20]. 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 be shared by all instances of this class, and will contain pointers to the virtual methods of the class. The first value preceding the vtable is a pointer to the RTTI of the class. A program finds out the class inheritance of a class instance at runtime by following this pointer.

The RTTI itself is composed of a pointer to a vtable for the typeinfo class (defined by the compiler) as well as a pointer to the type name (this type name is not removed by simple stripping of the binary, like with *objdcopy –strip-all*). This name is mangled using C++ mangling, and can trivially be demangled. Here are some examples of C++ name mangling : *_ZTI6Feline* demangles to *typeinfo for Feline*; *_ZTV6Feline* demangles to *vtable for Feline*. Mangling is especially useful when defining several methods in the same scope with different arguments (this is called overloading), in order to differentiate the different methods. Mangling is also useful because of special characters in class names, like the *::* characters who separate namespaces. The next values of the RTTI are pointers to the RTTIs of the parent classes. These are used at runtime to check the relationship between classes. See Figure 2.2 for an example, with the Dog RTTI containing a pointer to the Animal RTTI.

2.3.2 Exceptions

Exceptions are one of the defining features that make C++ stand out from C. In contrast to C errors, which terminate the whole program, C++ exceptions can be caught and handled.

In order for the program to continue execution after an exception is caught, binaries need some way of safely doing *stack unwinding*. This means that the systems needs to restore the stack variables to the state they were before entering the try block.

The information on how to safely unwind the stack is stored in the `.eh_frame` section. It is encoded in a format similar to DWARF data (see section 2.4).

```
#include <iostream>

using namespace std;


void this_method_might_throw(int x) {
  // This buffer is instantiated.
  char* this_object_could_leak = new char[1024];

  // If the throw is called, the buffer will stay instantiated if there is
  // no stack unwinding.
  if (x) throw runtime_error(":explosion:");

  // If the throw is not called, we free the buffer ourselves.
  delete [] this_object_could_leak;
}

int main() {
  try {
    this_method_might_throw(10);
  } catch (const exception& e) {
    return 1;
  }
  // Thanks to stack unwinding, we can be sure that the buffer from
  // this_method_might_throw was freed and will not leak.
  return 0;
}
```

Figure 2.3: Example of C++ exception throwing and catching to illustrate stack unwinding

The system also needs to safely release memory that is outside of the scope reached after an exception is caught. See Figure 2.3 for an example of code that will only guarantee safety if there is this mechanism in place.

This memory safety logic is defined in a table called the *Language Specific Data Area*, in a section of the binary called the `.gcc_except_table` [36]. In theory this section should be language specific, but in practice the implementation is the same for every language when using the *GNU Compiler Collection* [16] (GCC), that supports languages like Objective-C, Fortran or Go, and when using *LLVM* [13], that supports most well-known compiled languages.

In Figure 2.4 you can see a `.gcc_except_table` that was generated from the code in Figure 2.3. We will now go through line by line to explain the inner workings of this table. This will help us explain why they are so important, and later why we had to rewrite them (see section 3.2).

The first three lines define that "this is an except table starting".

The following seven lines define the parameters of this except table. These are, for example, the encoding of the values present in the table (the `.byte 3` and `.byte 1` lines), or the length of the table (the `.uleb128 .Lcst_end1-.Lcst_begin1` line).

```
        .section        .gcc_except_table,"a",@progbits
        .p2align        2
GCC_except_table2:

.Lexception1:
        .byte   255                     # @LPStart Encoding = omit
        .byte   3                       # @TType Encoding = udata4
        .uleb128 .Lttbase0-.Lttbaseref0

.Lttbaseref0:
        .byte   1                       # Call site Encoding = uleb128
        .uleb128 .Lcst_end1-.Lcst_begin1

.Lcst_begin1:
        .uleb128 .Ltmp3-.Lfunc_begin1   # >> Call Site 1 <<
        .uleb128 .Ltmp4-.Ltmp3          #   Call between "try" and "catch"
        .uleb128 .Ltmp5-.Lfunc_begin1   #     jumps to .Ltmp5 (cleanup function)
        .byte   1                       #   On action: 1

        .uleb128 .Ltmp4-.Lfunc_begin1   # >> Call Site 2 <<
        .uleb128 .Lfunc_end2-.Ltmp4     #   Call after "catch"
        .byte   0                       #     has no landing pad (do nothing)
        .byte   0                       #   On action: cleanup

.Lcst_end1:
        .byte   1                       # >> Action Record 1 <<
                                        #   Catch TypeInfo 1
        .byte   0                       #   No further actions
        .p2align        2
                                        # >> Catch TypeInfos <<
        .long   _ZTISt9exception        # TypeInfo 1 (what we are catching)
```

Figure 2.4: Assembly representation of an except_table for a simple program

Next, we have the beginning of the call site table at `Lcst_begin1`. The first call site defines that if an exception is thrown while executing instructions between the "try" and "catch blocks (here defined as the `Ltmp3` and `Ltmp4` labels), then you see if it matches *action 1* before jumping to the landing pad (here `Ltmp5`).

The actions are defined under the call site table, in the action table. You can see them defined at the `Lcst_end1` label. The first action is defined as "catch TypeInfo 1". The TypeInfos are defined bellow the actions, as a list of pointers to the different exceptions we might be catching. Here, typeinfo 1 is a simple "exception", which matches with our C++ code.

Coming back to the first call site : the landing pad of this call site is, in this case, a pointer to code that will free and clean up the buffers and variables that might have been instantiated.

Quite interestingly, the "TypeInfo" types are checked against the *Run-Time Type Information* (RTTI) we talked about in the previous subsection. We will have more to say about RTTIs in the following chapters.

2.3.3 Global constructors and frame dummy

During the execution of an ELF file, some methods will be run before the main program starts. Pointers to these methods are stored in the `init_array`. A simple C++ binary that uses exceptions will have an initialization array containing a pointer to the `frame_dummy`. This method is in charge of setting up the stack frames to unwind the frame during exception handling later on. In the initialization array, you will also find pointers to a global constructor for every compilation unit (usually, a compilation unit is a C++ source file).

2.4 The 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. You can also map source code to binary code. All of this information is usually used in debuggers.

The `.eh_frame` section mentioned in subsection 2.3.2 is also used by debuggers to unwind the stack for debugging purposes. It gives information to identify where to catch exceptions, as well as what destructors to call while unwinding the stack.

DWARF data takes the form of a tree of values. The root of the tree is called the *Compile Unit* (CU). It contains information about the source code file, the programming language used as well

as the compiler.

2.5 RetroWrite

RetroWrite[11] is a project from the HexHive lab that aims to statically rewrite binaries and instrument them, to improve fuzzing capabilities, or retrofit security measures. Static binary rewriting is the process of taking a binary file, and modifying it such that there is a patch applied to it but the other functionalities remain the same.

RetroWrite's core idea is to symbolize all of the addresses in the machine code, in order to generate reassembleable assembly. This means transforming an address offset (+8) into a symbol (.LD1234) and adding the appropriate label at that location. These labels will be used by the assembler to compute the addresses of functions and data offsets for example. This way, we can add any instruction after and before existing instructions without breaking relative offsets.

RetroWrite only works on *x86_64* unstripped position-independent executables (see section 2.2), compiled from C. This enables RetroWrite to stay heuristics-free (meaning that it is designed to work for every example without exceptions).

If you try to rewrite a binary that is stripped, the code blocks that correspond to functions will not be identified as such. RetroWrite currently needs to know the function boundaries to rewrite a binary. Identifying the function boundaries would require heuristics.

If you try to rewrite a binary that is not position-independent, the positions (pointers) in your binary are indistinguishable from data, and it becomes an unsolvable problem to differentiate between data and a pointer, which is why you would also need heuristics in that scenario.

Chapter 3

Design

3.1 dis-cover

The primary goal of this project was to extract some debug information from a stripped C++ binary, make it available through DWARF in order to use that information in other projects, as a proof of concept. The hope is that this process will be re-used in other projects in the future. For this project, we decided to focus on recovering class information from *Run-Time Type Information* (RTTI, see subsection 2.3.1).

Class recovery from compiled C++ binaries has been tried multiple ways before: there are plugins for popular debugging tools, like ida_gcc_rtti [25] for IDA Pro (which requires a license), or Ghidra-Cpp-Class-Analyzer [34] for Ghidra (but this information is only usable inside of the Ghidra debugger); there are academic projects such as MARX [29] 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), with around 90% accuracy.

3.1.1 Finding Run-Time Type Information

As we mentioned in the Background chapter, the RTTIs 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%). This does not mean that the other 46% of packages do not use classes, because in many cases these classes are optimized away by the compiler (in the more straightforward cases). We will share more details about this experiment in section 5.3.

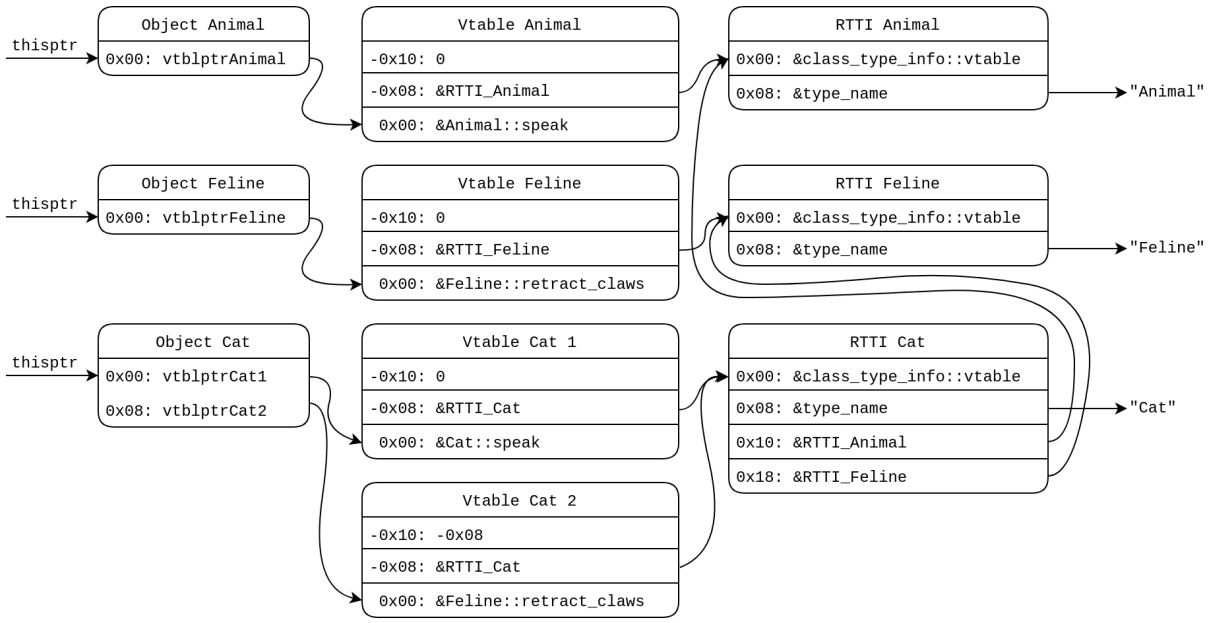

Figure 3.1: Overview of an example of vtables and RTTI in memory with multiple inheritance

When looking at a binary's vtables and RTTIs, we only need to look for a specific subset of them : the primary base virtual tables and their RTTIs. There is one for each class defined in the binary. Secondary virtual tables occur when there is multiple inheritance and complex method overwrites. They define only a subset of a class' methods, and may not contain RTTIs. We can differentiate the two kinds of virtual tables by the presence of an offset at the beginning of the vtable : a primary base virtual table will have a value of 0 as an offset, whereas a secondary virtual table will have an offset to the primary base virtual table. See Figure 3.1 for a visual example of RTTIs and vtables in a multiple inheritance scenario.

3.1.2 DWARF

Once we have extracted class information from RTTI, next comes the question : what should we do with all of this class information ? What would be the most useful format to output the data in ? This is where DWARF [5] comes in the picture. DWARF is the debugging standard for UNIX 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 compiled binaries, written in C, C++ or Rust for example. 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 [10] from DARPA. He has published a paper in 2011 where he

```
< 1><0x00001d94>    DW_TAG_class_type
                      DW_AT_containing_type        <0x00001d94>
                      DW_AT_calling_convention     DW_CC_pass_by_reference
                      DW_AT_name                   Animal
                      DW_AT_byte_size              0x00000008
                      DW_AT_decl_file              0x00000022  /tmp/animals.cpp
                      DW_AT_decl_line              0x00000006
< 2><0x00001da1>      DW_TAG_member
                        DW_AT_name                   _vptr$Animal
                        DW_AT_type                   <0x00001dd2>
                        DW_AT_data_member_location   0
                        DW_AT_artificial             yes(1)
< 2><0x00001dab>      DW_TAG_subprogram
                        DW_AT_linkage_name           _ZN6Animal5speakEv
                        DW_AT_name                   speak
                        DW_AT_decl_file              0x00000022  /tmp/animals.cpp
                        DW_AT_decl_line              0x00000008
                        DW_AT_virtuality             DW_VIRTUALITY_virtual
                        DW_AT_declaration            yes(1)
                        DW_AT_external               yes(1)
                        DW_AT_accessibility          DW_ACCESS_public
                        DW_AT_containing_type        <0x00001d94>
```

Figure 3.2: Extract of a dwarfdump output showing pointers and methods

exploits certain features of DWARF to control program execution : Exploiting the hard-working DWARF [27]. In it, they prove that the DWARF information used during exception handling is Turing-complete and can be used to embed exploits in executables.

DWARF data is structured as a tree of *Debug Information Entries* (DIEs). These entries contain a *tag* and a list of *attributes*. These entries can describe how the original code is written, as well as information about classes, variables, types, and other structures in the binary. Figure 3.2 shows an example of such entries, describing pointers and methods in a simple program. Lines beginning with *DW_TAG* describe the type of the debug information entry, and the following lines beginning with *DW_AT* describe the attributes of this entry.

The DWARF format also uses abbreviations to compress the debug information. Instead of re-describing a DIE every time, their structure is defined in an abbreviation table (stored in the *debug_abbrev* section of the ELF file). Each DIE will have an index in that table, which describes its tag and list if attributes names and types. The rest of the DIE is the list of its attributes, in the order they were defined in the abbreviation.

In this project, we use very minimalist DIE abbreviations, containing only what we could infer from the analysis.

With dis-cover, we 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 section 4.4.

3.1.3 Exceptions

Exceptions are often implemented as classes, and dis-cover will naturally recover information about programmer-defined exceptions. Future extensions of this work might consider recovering more information about exceptions.

We had to study and work with exceptions and exception handling frames for the augmentation of RetroWrite with C++ capabilities (see section 4.6, section 3.2 and subsection 2.3.2 for more details).

3.2 RetroWrite

Today, RetroWrite supports the reversing of *x86_64* position-independent binaries. There was also work to augment RetroWrite to support kernel code [32] and *arm_64* [3]. We aimed to add C++ capabilities for *x86_64* binaries only, as it is the most widely used platform (though the same logic will most probably apply with a little tweaking to *arm_64* and other architectures).

The first bug we encountered while trying to use RetroWrite on C++ binaries was due to the fact that the *initialization array* (see subsection 2.3.3) contained a pointer to the *frame dummy* and this was not handled well during RetroWrite's symbolization process. By adding special treatment for that scenario, we resolved this failure (see more details in subsection 4.6.1).

Once that was fixed, we saw that using RetroWrite on binaries that had C++ exception handling would cause crashes in the new binary, caused by the fact that the exceptions thrown were never caught.

To fix that, we had to fix the symbolization of two sections. First, we focused on the `eh_frame` section that contains stack unwinding information. See subsection 4.6.2 for the implementation details of this process. Second, we worked on the `.gcc_except_table` section that contains the *Language Specific Data Area* (LSDA), which handles the correct handling of exception catching as well as the proper calling of variable destructors when leaving certain code blocks during a throw.

This LSDA describes, row by row, actions to take for specific instruction in the code we are hitting while dealing with a thrown exception. These actions could be : destructing an out-of-scope variable, or examining a catch clause to see if they should catch the current exception. See subsection 2.3.2 for an example of this process.

We had to rewrite the LSDA in our new assembly representation, with the appropriate labels and values, in order to make the exceptions work as they should (see how we achieved that in subsection 4.6.3)

Once this was done, we could rewrite basic C++ binaries that used exceptions. Bigger projects still caused some issues, that will be addressed before the code becomes public.

3.3 Design for future integrations

As described in subsection 3.1.2, we use the DWARF format as an output for the extracted information. This makes the knowledge easily reusable in other reverse engineering projects.

For example, dis-cover output could be used in RetroWrite in order to add instrumentation around *Object Type Integrity*, as defined in the CFIXX paper [4], or reinforcing type checks as defined in the HexType paper [21] to avoid type confusion errors and vulnerabilities. See more details about these projects in subsection 6.3.1 and subsection 6.3.2.

We also designed dis-cover in a heuristics-free way, which fits perfectly with RetroWrite's current goal to avoid heuristics.

This heuristics-free design should not be an end goal though. If later additions to RetroWrite were to improve its capabilities with heuristics like those defined in the MARX paper for example (see section 6.1), it would only make the tool stronger and more reliable. The fact that we used the DWARF format as an output is what we consider the most interesting part of the dis-cover tool.

Chapter 4

Implementation

We decided to write a python module for this project, as the python ecosystem has great reverse engineering packages, and for easy integration in RetroWrite, which is also written in python.

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) and `.data.rel.ro`, but could sometimes also be related sections like `.data.rel.ro.local`, or `.rdata`. As per the Itanium C++ ABI, the base vtable of a class will contain an "offset-to-top" at offset *-0x10*, which will be 0 in the primary base virtual table's case, followed by a pointer to an RTTI at offset *-0x08*. 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. See Figure 4.1 for a hexdump of vtables and RTTIs in an ELF file.

Next, we analyze the data at the pointer's value. We check that we have not already extracted a class at that address, and if not, we assert the following address is a pointer to a string located in a read-only data section. If it is, we can extract that string, demangle it, and we have a class name, a virtual table address and an RTTI address that we have extracted. 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 extracted 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.

```
Relevant symbols:
0000000000003d40  _ZTV6Animal  VTable for Animal
0000000000003d68  _ZTI6Animal  RTTI for Animal
0000000000003d78  _ZTV3Dog     VTable for Dog
0000000000003da0  _ZTI3Dog     RTTI for Dog

Hexdump:
[...]
00002000  01 00 02 00 36 41 6e 69  6d 61 6c 00 33 44 6f 67  |....6Animal.3Dog|
00002010  00 00 00 01 1b 03 3b 7c  00 00 00 00 00 00 00 00  |......;|........|
[...]
00003d30  50 11 00 00 00 00 00 00  10 11 00 00 00 00 00 00  |P...............|
00003d40  00 00 00 00 00 00 00 00  68 3d 00 00 00 00 00 00  |........h=......|
00003d50  80 12 00 00 00 00 00 00  90 12 00 00 00 00 00 00  |................|
00003d60  c0 12 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00003d70  04 20 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |. ..............|
00003d80  a0 3d 00 00 00 00 00 00  d0 12 00 00 00 00 00 00  |.=..............|
00003d90  f0 12 00 00 00 00 00 00  20 13 00 00 00 00 00 00  |........ .......|
00003da0  00 00 00 00 00 00 00 00  0b 20 00 00 00 00 00 00  |......... ......|
00003db0  68 3d 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |h=..............|
00003dc0  01 00 00 00 00 00 00 00  01 00 00 00 00 00 00 00  |................|
[...]
```

Figure 4.1: Hexdump of the location of vtables and RTTIs in an 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_type*s. These *class_type*s will themselves have as children *inheritance* values if the class inherits from another class. Figure 4.2 shows a very simple example of this, with two class types and one inheriting from the other.

The strings themselves are stored in another section, `.debug_str`, and are referred to with their offset in that section.

4.3 Creating symbols

Symbols are mainly used for shared object loading. To use the *printf* method from *stdio.h* for example, the binary must be aware of a method named *printf*. The same is true for *system_clock* in C++ for example. Symbols are also used to have access to variable names, class names, function names or any other kind of text information when debugging a binary. In dis-cover, we want to create two symbols for every class we have found during the analysis : one pointing to the vtable

```
< 1><0x0000001a>    DW_TAG_class_type
                      DW_AT_containing_type        <0x0000001a>
                      DW_AT_calling_convention     DW_CC_pass_by_reference
                      DW_AT_name                   Animal
                      DW_AT_byte_size              0x00000008
< 1><0x00000026>    DW_TAG_class_type
                      DW_AT_containing_type        <0x00000026>
                      DW_AT_calling_convention     DW_CC_pass_by_reference
                      DW_AT_name                   Dog
                      DW_AT_byte_size              0x00000008
< 2><0x00000031>      DW_TAG_inheritance
                        DW_AT_type                   <0x0000001a>
```

Figure 4.2: Extract of a dwarfdump output showing simple inheritance

and one pointing to the RTTI, and labeling them as such.

In order to create new symbol sections, we take the symbol table from the original binary (if there was one) and append the aforementioned symbols.

The two symbol sections are `.symtab`, which contains the information (offset, size, type, ...) for each symbol, and `.strtab`, which contains the strings related to these symbols.

4.4 Wrapping things together

Once we have the three debug sections ready (`.debug_abbrev` with the debug types, `.debug_info` with the debug information, and *debug_str* with the strings) as well as the two symbol sections, we want to make them available to the user. They contain all of the information we extracted from the binary.

We now have to go through a few hoops to make the debug and symbol information available in one binary because these sections do not come alone : they have to be accompanied by machine code and data. We did not want to recreate a whole compiler to assemble all of this data. Luckily, the *eu-unstrip* [12] tool was created as an utility application to "combine stripped files with separate symbols and debug information". Figure 4.3 shows a diagram of the process we will explain in this section.

4.4.1 Creating a fake ELF file

We start by constructing a *program header table*. This table contains information about the offset and size of each segment of the binary (which segment is used for what, and their read/write permissions). The ELF file we're creating will not be run, but only used temporarily. Thus, we noticed that we did not have to create a valid program header table for the process to work. We

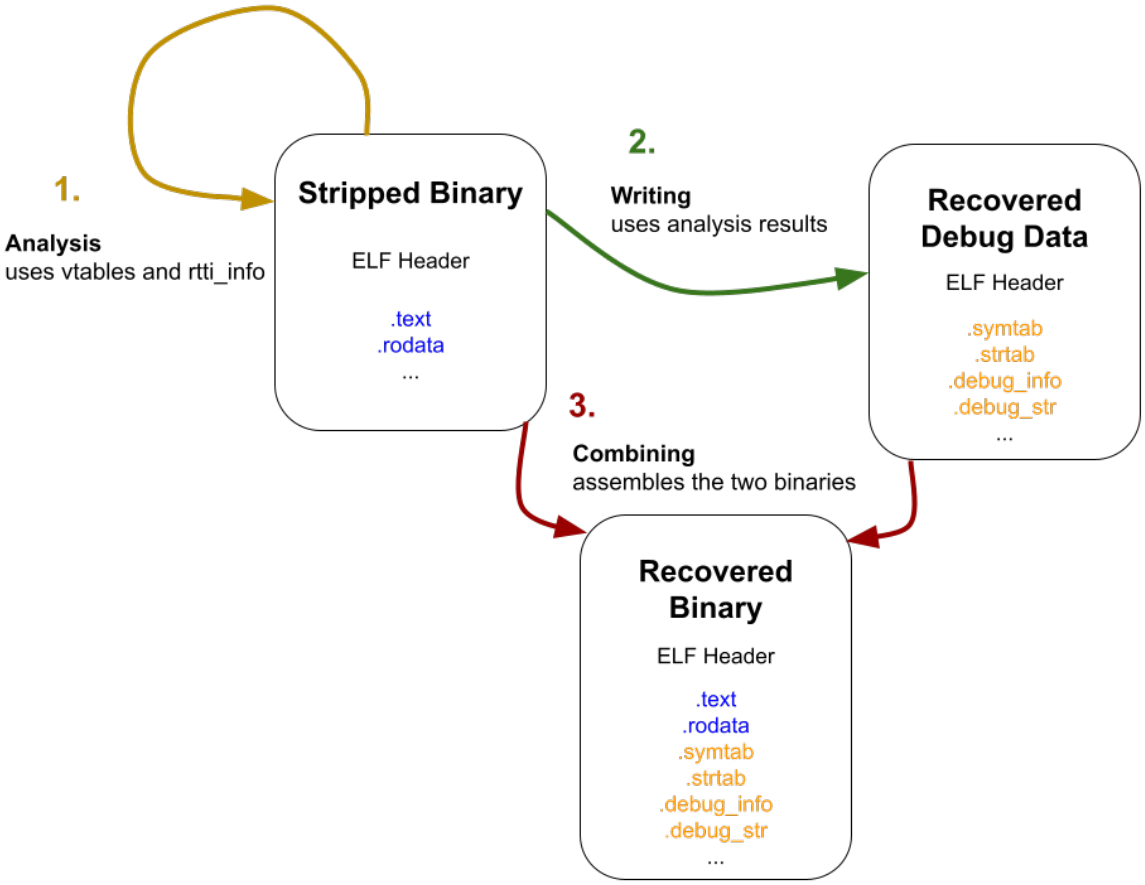

Figure 4.3: Combining the ELF files

simply copy this program header table almost as-is from the original binary.

Next, we use the individual sections we built earlier and construct the *section header table*. For every section present in the original binary, we create an entry in the section header table, reusing most values from the original section header table. The only values we modify is the offset. For every section that we have created, we add the appropriate row in the section header table. Every section name gets added to the `.shstrtab` section, as per the convention.

Finally, we construct the *elf header*, taking some of the values from the original binaries, and calculating some others from the size of the tables and sections we have built. We can now create a fake ELF file by appending the elf header, the program header table, the sections we built and the section header table.

In order to use eu-unstrip, we had to make the fake binary match the original binary as closely as possible regarding headers and segments.

4.4.2 Stripping the original ELF file

Next, we will create a stripped version of the original ELF file. We use the *objcopy –strip-all* command. This is to avoid section conflicts in the next step.

4.4.3 Combining the two ELF files

Now, we can use the ELF utility program *eu-unstrip* to combine the two ELF files we have created into one. The newly created combined ELF file will contain all of the code and data from the original file, as well as the debug and symbol sections we have created.

4.5 Visualizing and analyzing the output

The *dis-cover* tool also comes with two options for the visualization and further analysis of the class hierarchy we have found.

The first of these options is writing the data to a `.dot` file, that can be later used with the well-known Graphviz [17] tool to create a useful visual representation of the classes and their relationships to each other. An example of such a graph can be found in Figure 5.1. This is only a partial view of a much larger graph.

The second option is serializing the data to a `.pickle` file [14], that can later be opened by another python script for further analysis. This is how we conducted the research detailed in section 5.3.

4.6 C++ capabilities in RetroWrite

4.6.1 Fixing the .init_array

The first modification we did to RetroWrite to add support for C++ binaries was to address the `init_array` errors described in section 3.2. This meant treating the `frame_dummy` method differently, as it was ignored by the symbolizer, but included in the `init_array`.

As described in subsection 2.3.3, the `frame_dummy` method is used to set up the stack frames for exception handling by the program. The compiler will automatically generate one during the compilation process.

The fix to this bug was simply to remove the `frame_dummy` pointer from the `init_array` when it was present in the intermediate representation of the assembly code. We then let the assembler add a pointer to the `frame_dummy` it just created to the `init_array` at assembly time.

4.6.2 Adding CFI directives

We then had to insert the relevant Call Frame Information (CFI) directives into the code blocks. These directives specify which code blocks should have an entry in the `.eh_frame` section, and what these entries should contain, for stack unwinding during exception handling (see subsection 2.3.2).

We implemented this by recovering the original binary's `.eh_frame` section, parsing each value, and adding the relevant directive at the right location for each one.

4.6.3 Creating a Language Specific Data Area

The next modification was a larger one : the whole *Language Specific Data Areas* (LSDAs) had to be symbolized and rewritten. See section 3.2 and subsection 2.3.2 for more details. These tables, contained in the `.gcc_except_table` section, have offset to cleanup functions, as well as lengths of instructions (which we transform into label differences). There is one LSDA for each code block in a program.

The LSDA starts with some fields that describe some properties of itself, such as the encoding of its values, an offset for its pointers and its length.

There are multiple tables in the LSDA : First the *call-site table*. It contains information about what method to call if an exception is thrown in a certain location, and what *actions* it should catch (what type of error it should catch).

The second table is the *action table*. It contains offsets to next action (similarly to a linked list), and an index to the types table.

The third and final table is the *types table*, which contains pointers to type information structures. For example, it might contain a pointer to type information for `std::invalid_argument` and another pointer to type information for `MyCustomException`.

As a side note : these pointers could also be found by an improved version of *dis-cover* to find even more classes during the analysis.

In order to successfully rewrite the binaries with exceptions, we had to symbolize every table, which meant extracting the information we needed from the original binary's LSDAs, and rewriting our own for each code block, using the appropriate labels for each value (for example, an offset to another table is computed with a difference between two labels, to maintain the position-independence of the code).

Chapter 5

Evaluation

5.1 Small case studies

In addition to the first version of dis-cover, we created three small programs highlighting different features of C++. One was using simple inheritance, one had a namespace (which we can and should recover as part of the analysis), and the last one was a use case of multiple inheritance (using the diamond problem). These examples can be found in the code repository [23] for the dis-cover project.

We also created a script that would compile these three programs using different levels of compiler optimization. We also compiled the programs with DWARF information included, and we could extract from these binaries the number of classes that were present in the program, to use as a base comparison number. We could then use dis-cover to recover every class and the correct hierarchy tree from the other binaries. This served as a useful benchmark to check whether dis-cover was functioning correctly if we tried to apply changes to it.

These examples alone were not capable of letting us evaluate the full capabilities of dis-cover. We found a few big applications that could serve that purpose.

5.2 Real-world case studies

The first and smallest of these case studies was the *gold linker* [37]. We found 571 classes in version 1.16 of the program. This provides a good benchmark, but the classes themselves do not make use of multiple inheritance (only a "simple" inheritance tree).

LibreOffice [15] on the other hand provided us with a great test case : the program is fragmented into many small libraries, containing some interesting uses of multiple inheritance. See Figure 5.1

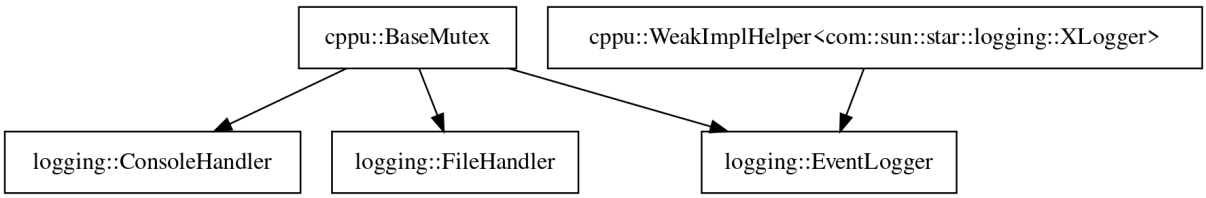

Figure 5.1: Partial class tree of libloglo.so from LibreOffice

for an example of multiple inheritance in the *libloglo.so* library from LibreOffice. This particular example was very important for verifying a big bug that was present in an early version of dis-cover. After fixing the bug, we were noticing around 10% more inheritance links in some projects (but not more classes). Having access to the open-source LibreOffice code to check that we had found the right inheritance links was extremely helpful.

Finally, we also studied the closed-source *zoom* [6] binary. This test case is very interesting in two ways. First, we found *6039 classes* in the binary, with *5601 edges* in the class hierarchy graph. Second, as a large (76M) and complex ELF file, it served as a perfect benchmark for the performance of the algorithm. By adding a simple check earlier in the RTTI-spotting pipeline, we sped up the analysis of the zoom binary 13.37 times, from over an hour to around five minutes.

Table 5.1 details the benchmarks we conducted on many well-known projects.

ELF File and version	Size	Size of .rodata	Computation time	Amount of classes recovered
gold 1.16	2.3M	0.1M	0m0.605s	571
ceph-dencoder 15.2.13	29M	0.9M	0m12.493s	2959
zoom 5.5.7938.0228	76M	16M	5m50.626s	6039

Table 5.1: Benchmarks of dis-cover on a machine running Ubuntu 20.04 with a 2.6 GHz dedicated vCPU

5.3 Analysis of Debian packages

As we have shown before, out of the 5827 Debian packages that list C++ as a dependency, we extracted 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). The mean number of classes in packages where there are classes is 300.

In Figure 5.2, we can see the graph of the number of classes in individual packages. We excluded the 20 projects with the most classes, as they made the graph much less readable (some packages had more then 10,000 classes).

Using data from the Debian Popularity Contest [2] (popcon), we analyzed the most popular packages to see if the numbers evolved. In Table 5.2 you can examine our results. We can observe

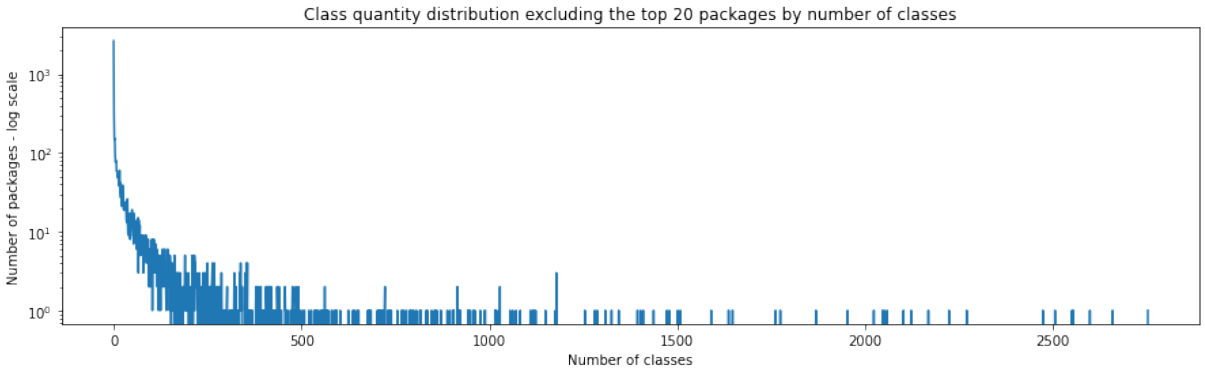

Figure 5.2: Class quantity distribution in Debian packages with C++ as a dependency (y log scale)

there are more classes (more frequently and in greater numbers) in the most popular Debian packages then in the overall population of Debian packages.

Number of packages	Sorting criteria	Percentage of packages with classes recovered	Mean number of classes per package with classes
100	usage	71%	330
100	installations	61%	356
1000	usage	64%	171
1000	installations	61%	169
5827	-	54%	300

Table 5.2: Class statistics on the most popular packages of Debian

5.4 Testing RetroWrite

We tested the new C++ capabilities of RetroWrite on examples we had coded ourselves. These included complex examples with user-defined errors types, catching of multiple error types, and memory allocation in the code blocks that threw the exception. The tool did not yet manage to rewrite larger projects. This will be fixed before we make the code public.

Also, testing on well-known closed-source projects proved an issue, because the binaries were stripped, and thus the function boundaries were not known. RetroWrite does not yet support unstripped binaries because of this. In order to make RetroWrite work on projects such as Chrome [18] or Zoom [6], one could use a function identification tool such as the ones found in IDA Pro [31] or radare2 [28]. These tools use heuristics to find the function boundaries, so they might not work for every project.

Chapter 6

Related Work

6.1 MARX

MARX [29] is a static analysis tool which is related to what dis-cover is trying to achieve. Using analysis of the virtual tables in a C++ binaries, they accurately extract 90% of the classes. This is done without the need for RTTIs, and thus can target a larger number of projects (such as the chrome browser and the Node.js runtime) at the cost of having a probabilistic approach.

In comparison, dis-cover can recover 100% of the classes present in a binary, but only if the RTTIs were kept at compilation time (so no class can be recovered from the chrome browser and the Node.js runtime).

MARX uses six heuristics to locate the virtual tables, related to the table layout and content.

This project could be a great candidate for future integration into RetroWrite, if we extend the tool to add defenses around type integrity for example.

6.2 Plugins

There exists multiple plugins for reverse engineering frameworks that serve the same purpose as dis-cover. For IDA Pro, there exists *IDA GCC RTTI* [25], and for Ghidra there is the *Class and Run-Time Type Information Analyzer* [34].

These are very useful, and integrate very well into the reverse engineering frameworks, but they cannot make that information available to other tools.

The IDA Pro plugin does offer the option to create a *.dot* graph file that can be used to create a

visual representation of the hierarchy tree. Our *dis-cover* tool offers the same functionality (see section 4.5)

6.3 Type Integrity

6.3.1 CFIXX

CFIXX [4] introduces *Object Type Integrity*, which is a dynamic (runtime) mechanism that keeps track of object types and prevents adversaries from overwriting type information to hijack the control flow.

Indeed, some attacks rely on overwriting the pointer to an object's virtual table with another one, to modify the methods called by the code and tamper with the control flow integrity of the program. Object Type Integrity is achieved by making sure that the right virtual table is accessed at runtime for each specific object.

A future project could use the classes extracted with dis-cover and the rewriting capabilities of RetroWrite to implement object type integrity on closed-source binaries.

6.3.2 HexType

HexType [21] aims to remove *type confusion errors* from programs, by adding explicit runtime checks during compilation. This is done even for objects that are not polymorphic, so as to remove all of the attack surface.

Type confusion errors occur when a type cast is done on incompatible types. This can then lead to errors that can be abused to attack programs and gain control of systems.

Adding runtime checks during compilation allows for great coverage and low overhead.

6.4 Exploiting the hard-working DWARF

Exploiting the Hard-Working DWARF [27] demonstrates that it is possible to incorporate exploits in the `.eh_frame` and `.gcc_except_table` sections of an ELF file.

They show that the exception handling mechanism of ELF files is a Turing-complete language, and that they can incorporate arbitrary programs into DWARF sections that get executed whenever an exception is thrown by the code. These arbitrary programs can be used to take control over the system.

This is quite interesting, as malware-finding software usually does not scan these debug sections for malicious code, and existing reverse engineering tools do not know how to interpret these malformed DWARF sections.

This paper rekindled a global interest in DWARF as an academic research topic, which lead to projects like this one.

This project also shows both the power of the DWARF debugging format and the need for more research to be done around this topic.

Chapter 7

Conclusion

In conclusion we develop *dis-cover*, a static analysis tool for C++ binaries that can find polymorphic class hierarchies without using heuristics, if the compiler was told to keep the *Run-Time Type Information*. The tool injects its findings in the analyzed binary as DWARF debug information and ELF symbols, so it can be used in any debugging tool. We hope this serves as an example for future binary analysis tools.

We also extend the static position-independent binary rewriter RetroWrite with C++ support. To our knowledge, RetroWrite is the first static rewriter that can add instrumentation like AddressSanitizer to closed-source C++ binaries that make use of exceptions.

Bibliography

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

[2] Bill Allombert. *Debian Popularity Contest*. https://popcon.debian.org/. Accessed: 2021-08-20.

[3] Luca Di Bartolomeo. *ArmWrestling: efficient binaryrewriting for ARM*. http://hexhive.epfl.ch/theses/20-dibartolomeo-thesis.pdf. Accessed: 2021-06-15.

[4] Nathan Burow, Derrick McKee, Scott A. Carr, and Mathias Payer. "CFIXX: Object Type Integrity for C++ Virtual Dispatch". In: *Network and Distributed System Security Symposium*. 2018.

[5] DWARF Standards Committee. *The DWARF Debugging Standard*. http://www.dwarfstd.org/. Accessed: 2021-06-14.

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

[7] The Qt Company. *Qt Framework*. https://www.qt.io/product/framework. Accessed: 2021-06-22.

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

[9] cppreference.com. *static_cast conversion*. https://en.cppreference.com/w/cpp/language/static_cast. Accessed: 2021-06-22.

[10] DARPA. *Dr. Sergey Bratus*. https://www.darpa.mil/staff/dr-sergey-bratus. Accessed: 2021-06-15.

[11] 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.

[12] Free Software Foundation. *The elfutils project*. https://sourceware.org/elfutils/. Accessed: 2021-08-26.

[13] LLVM Foundation. *The LLVM Compiler Infrastructure*. https://llvm.org/. Accessed: 2021-08-25.

[14] Python Software Foundation. *pickle - Python object serialization*. https://docs.python.org/3/library/pickle.html. Accessed: 2021-08-20.

[15] The Document Foundation. *LibreOffice*. https://www.libreoffice.org/. Accessed: 2021-06-19.

[16] Inc Free Software Foundation. *GCC, the GNU Compiler Collection*. https://gcc.gnu.org/. Accessed: 2021-08-25.

[17] Emden R. Gansner, Eleftherios Koutsofios, and Stephen North. *Drawing graphs with dot*. https://www.graphviz.org/pdf/dotguide.pdf. Accessed: 2021-08-20.

[18] Google. *Google Chrome*. https://www.google.com/chrome/. Accessed: 2021-06-22.

[19] grsecurity. *Address space layout randomization*. https://grsecurity.net/PaX-presentation.pdf. Accessed: 2021-06-23.

[20] *Itanium C++ ABI*. https://itanium-cxx-abi.github.io/cxx-abi/abi.html. Accessed: 2021-06-15.

[21] 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.

[22] Evan Leibovitch. *Unix-on-Intel players agree on a common binary (It's the Linux ELF format)*. https://web.archive.org/web/20070227214032/http://www.telly.org/86open/. Accessed: 2021-08-17.

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

[24] Mozilla. *Firefox Browser*. https://www.mozilla.org/en-US/firefox/new/. Accessed: 2021-06-22.

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

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

[27] 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.

[28] radare org. *radare2*. https://rada.re/n/radare2.html. Accessed: 2021-08-25.

[29] 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.

[30] Mathias Payer. *Thesis Template*. https://github.com/HexHive/thesis_template. Accessed: 2021-08-19.

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

[32] Matteo Rizzo. *Hardening and Testing Privileged Code through Binary Rewriting*. http://hexhive.epfl.ch/theses/19-rizzo-thesis.pdf. Accessed: 2021-06-15.

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

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

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

[36] Ian Lance Taylor. *.gcc_except_table*. https://www.airs.com/blog/archives/464. Accessed: 2021-08-19.

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