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 26, 2021

Nobody tosses a dwarf !
— Gimli, son of Glóin

TODO Dedication

Acknowledgments

TODO Acknowledments

Lausanne, July 26, 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" [25]. Since then, the language has grown in popularity, and even surpassed C itself [23]. Examples of well-known C++ projects include modern browsers like Chrome [12] and Firefox [16], the Qt Framework [6], or the zoom [5] conferencing software. 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 [18], IDA Pro [21] 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 [10] 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** [15] 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. 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 have the **speak** method and whatever other properties and methods Animal could define. This logic is verified by the compiler.

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

class Feline {
  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; }
};
```

Figure 2.1: Polymorphic classes in C++

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** [8], will check that the casting is upcasting at compile time, but does not do any run-time checks to verify the validity of the conversion. 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** [7] expression. Dynamic casting is a safe kind of type conversion that can handle downcasting and sidecasting. 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.

You will also need to instantiate these classes, and have some run time logic to make the binary non-deterministic.

Louis: deterministic is not the right word, 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 children 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.

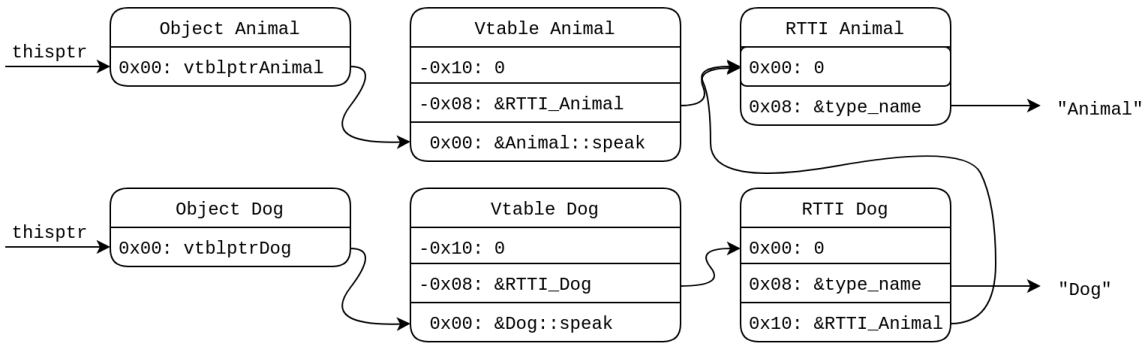

Figure 2.2: Overview of an example of VTables and RTTI in memory

The structure of VTables and RTTIs is detailed in Figure 2.2. All of this is defined in the Itanium C++ ABI [13]. 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 a pointer to a VTable for the typeinfo class (defined by the compiler) and 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. 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.

2.2 Executable and Linkable Format (ELF)

Now we will study how all of the previously mentioned data is stored and organized. The common standard **Executable and Linkable Format** (ELF) format is mostly used as executable files, object code and shared libraries. It usually 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). The rest of an ELF file is taken up by 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).

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

Position-independence for code is a property that guarantees that code will execute correctly no matter the absolute address of the body of code. This is mostly possible by computing relative positions instead of using absolute addresses.

PIC is most often found for shared libraries, so that they can be inserted into an ELF file without worry. PIE binaries are seen as more secure because they allow for easy inserting of security measures like address space layout randomization [27].

Louis: @antony, what would be a more appropriate citation for ASLR ?

We will talk more about PIE in section 2.6.

2.4 Exceptions in C++

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 be able to continue execution after an exception is caught, binaries need some way of safely doing **stack unwinding**. 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 2.5).

Louis: Should I talk about relocation tables ? If yes how ?

2.5 DWARF debugging standard

Antony: MOAR!! :) There is quite a bit you can say here. DWARF powers debuggers, and lets you do things like map source to binary code, recover variable names, function names that might not be external symbols, as well as class information. Another key point to mention for later will be the structure of how it is used in EH_Frame. DWARF uses an abstract representation of the processor, so its register 1 for example must be mapped to the processor.

Louis: Regarding Antony's comment => more info about the justification for using DWARF.

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 section 2.4 is also used by debuggers to unwind the stack for debugging purposes.

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.

2.6 RetroWrite

The **RetroWrite**[10] project is a project from the HexHive lab that aims to statically rewrite binaries (transform from binary to assembly) to instrument them later to improve fuzzing capabilities.

RetroWrite's core idea is to symbolize all of the addresses in the machine code. This means transforming an an address offset (+8) into a symbol (.LD1234) and adding the appropriate label at that location. This way, we can add any instruction after and before existing instructions without breaking relative offsets.

RetroWrite only works on **x86_64** position-independent executables (see section 2.3), compiled from C. This enables the program to stay heuristics-free (meaning that it is designed to work for every example without exceptions).

Chapter 3

Design

3.1 Goals

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 be able 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 section 2.1).

Class recovery from compiled C++ binaries has been tried multiple ways before: there are plugins for popular debugging tools, like ida_gcc_rtti [17] for IDA Pro (which requires a license), or Ghidra-Cpp-Class-Analyzer [24] for Ghidra (but this information is only usable inside of the ghidra debugger); there are academic projects such as MARX [20] 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 the Evaluation chapter.

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 of each 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 do 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.

3.1.2 Exceptions

Exceptions are often implemented as classes, and our tool will naturally recover information about programmer-defined exceptions. 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). Future extension of this work might consider recovering more information about exceptions.

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 [4] 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.

Louis: Should I add another language ?

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 [9] 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 [19]. In it, they prove that the DWARF information used during exception handling is Turing-complete and can be used to embed exploits in executables.

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 [22] and arm_64 [2]. 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 and other architectures).

Dis-cover could be used in RetroWrite in order to add instrumentation around **Object Type Integrity**, as defined in the CFIXX paper [3], or reinforcing type checks as defined in the HexType paper [14] to avoid type confusion errors and vulnerabilities.

One of the goals of the project was to augment RetroWrite's process using **dis-cover**'s output. As RetroWrite does not use heuristics, we decided to design dis-cover in a heuristics-free way if possible (which it is).

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", 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.

Louis: This could be diagrammed (RTTIs) => multiple inheritance, diamond problem...

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_type**s. These **class_type**s 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
                      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                  Rectangle
                      DW_AT_byte_size             0x00000008
< 2><0x00000031>      DW_TAG_inheritance
                        DW_AT_type                  <0x0000001a>
```

Figure 4.1: Extract of a dwarfdump output showing simple inheritance

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

Louis: The three dwarf sections should be explained in more detail in the Background or Design sections.

4.3 Creating symbols

Symbols are mainly used for shared object loading, to import the **cout** object from **iostream** for example, the binary must be aware of an object named **cout**. 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 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

4.4.1 Creating a fake ELF file

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 have to make them available to the user. They contain all of the information we were able to extract from the binary. The first step is building an empty ELF file with only these five sections in them. We will discuss why in subsection 4.4.3.

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 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.

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.

The reason why we had to go through these hoops to make the debug and symbol information available to the user is 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** tool was created as an utility application to "combine stripped files with separate symbols and debug information". In order to use this tool, we had to make the fake binary match the original binary as closely as possible regarding headers and segments.

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).

We also created a script that would compile these 3 programs using different levels of compiler optimization. We could then use dis-cover to see if we could recover every class and the correct tree from the 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 Big case studies

The first and smallest of these case studies was the **gold linker** [26]. We were able to find 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 [11] 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 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). Being able to go check with the open-source LibreOffice code that we had found the right inheritance links was extremely helpful.

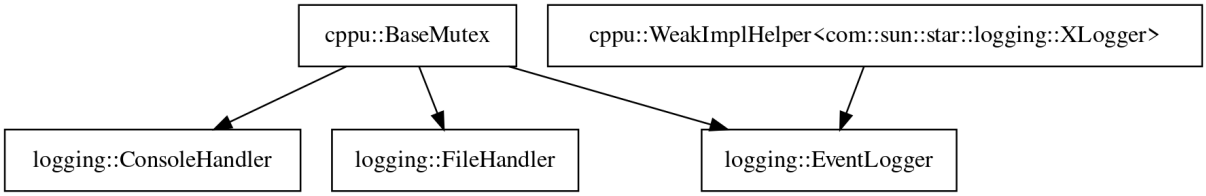

Figure 5.1: Partial class tree of libloglo.so from LibreOffice

Finally, we also were able to study the closed-source **zoom** [5] binary. This test case is very interesting in two ways. First, we are able to find **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 were able to speed up the analysis of the zoom binary 13.37 times, from over an hour to 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

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] 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.

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

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

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

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

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

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

[10] 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.

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

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

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

[14] 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.

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

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

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

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

[19] 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.

[20] 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.

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

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

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

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

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

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

[27] Wikipedia. *Address space layout randomization*. https://en.wikipedia.org/wiki/Address_space_layout_randomization. Accessed: 2021-06-23.