École Polytechnique Fédérale de Lausanne

Retrofitting defences to C++ code

by Hassan Habib

Bachelor Thesis

Prof. Dr. sc. ETH Mathias Payer
Thesis Advisor

Nicolas Badoux
Thesis Supervisor

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

June 12, 2022

Abstract

The Retrowrite project is a static binary rewriter that allows adding instrumentation into programs where the source code is not available.

C++ being a largely used programming language, we must ensure that programs written in this language are safe. By "safe" we mean that the application should behave normally when under malicious attack other other risks.

Our task is to test Retrowrite on various *Debian* packages written in *C++* to see if the binary could be modified for instrumentation.

From our results, we can see that Retrowrite worked on 38.1% of packages, but in 60.2% of cases, the assembly obtained could not be compiled back to a functioning executable. Unfortunately, we were unable to gain more information about the compilation errors because the compiler displayed the same error message each time. A more in-depth study of the assembly file would have been necessary to find the cause. Also, only a few programs encountered runtime errors. We have also found that file size plays a role in the success of Retrowrite. Large files seem not to be able to be transformed without encountering an error. One clue to the cause could be that large files use more shared libraries. We concluded that there are still possible improvements to make to have a reliable *C++* support.

Contents

Chapter 1

Introduction

Using only free software or even software whose source code is available is not that easy. For example, people are used to using a program suite like *Microsoft office* and don't have time to change their habits by adopting the use of an equivalent open-source application. This can be a security problem as it is difficult for the community to discover new bugs. Indeed, techniques such as fuzzing or memory sanitization can only be applied by applying instrumentation at compilation time. Fortunately, a technique named *binary rewriting* makes it possible to insert instrumentation without having access to the source code.

Binary rewriting is used to insert instrumentation into binary executables where the source code is not accessible. These tools allow the insertion of instrumentation by analyzing the executable file. They can be categorized into two approaches: static and dynamic. The first category generates an executable whose instrumentation could be added while the second adds these instrumentations during the execution of the program. The first approach has the advantage of having less impact on execution time.

Retrowrite is one most recent and successful tools. It has the advantage of producing executables that are sound and whose overhead is almost absent. Similarly, C/C++ are widely used languages because of their speed and the freedom they give in memory management. Retrowrite was initially built to support binary files written in C. Fortunately, work was then done to support C++.

We first wanted to integrate the work done in the *Hextype* project into Retrowrite, but we noticed that this work was too ambitious and could not be done in such a short time. And since the integration of C++ support has been done in the meantime, we decided to focus our work on testing Retrowrite on a large number of packages to get an overview of its performance and its reliability. To obtain a variety of programs to test, we chose to take the packages used in the *Debian* distribution. By this, we can use real programs used by users and see how Retrowrite works.

Chapter 2

Background

We will explain several aspects to allow the reader to be familiar with any binary rewriter and more precisely Retrowrite. The information given here is not exactly to understand our work but rather to be familiar with the tools we use and to understand how a binary rewriter works in general.

2.1 Compiler

Computer programs can be executed in two different ways: they can be interpreted by a software (an interpreter) or they can be compiled into machine code that can then be executed by the hardware.

In the first approach, the interpreter will read each line it encounters, and then it will translate and run them one at a time. Interpreting a program is much slower than running the machine code generated by a compiler as we must also run the interpreter alongside the program.

In the second approach, a compiler is needed to translate the source code into binary code. The compiler will analyze all the source code and generate the machine code. Note that it will not execute the source code but only generate an executable file.

Even if the compiled program runs faster than an interpreted one, one might prefer to write programs in an interpreted language because it is platform-independent in most cases. Furthermore, it takes less time for the application to be available if the source code is modified.

2.2 Binary executable

C++ being a compiled language, we will focus on the executable file generated by the compiler. We will now discuss the most common format of the binary executable: the ELF format.

2.2.1 ELF format

The dominant executable format for binary in Unix system is the ELF[1] format.

It is important to know the format of the binary file since we need to know where to find important information during the analysis done by a binary rewriter. Retrowrite focuses on ELF format files and uses *pyelftools*, a widely available ELF parser.

An executable file respecting the ELF format must follow a certain layout. At first, it must have an ELF header followed by a program header table or a section header table, or both. The ELF header contains the offset to the section header table and the program header table. The program header contains information about the segments used at runtime. The section header lets locate all the single sections of the binary. Figure 2.1 illustrates the structure of an ELF file. The *readelf* [11] command in *GNU/Linux* allows us to see the different section's information. The sections can be divided into two categories: the predefined non-user sections and the predefined user sections [10].

[1] Executable and Linkable Format

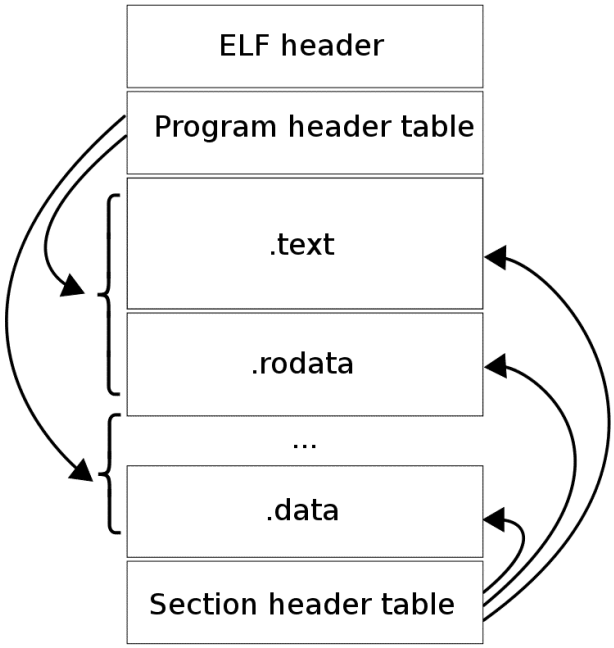

Figure 2.1: Structure of ELF file

2.2.2 Predefined User Sections

This categorie contains all sections that can be manipulated by the section control directives.

- .bss: contains uninitialized data.
- .data: contains initialised read-write data.
- .debug: contains debugging information.
- .rodata: contains initialised read-only data.
- .text: contains the code.

2.2.3 Predefined Non-User Sections

These are all the predefined sections by the assembler.

- .dynamic: Holds all needed information for dynamic linking.
- .dynstr: contains string table of .dynsym section.

- .dynsym: contains symbol tables dedicated to dynamically linked symbols.
- .got: contains the global offset table.
- .interp: contains the path name of a program interpreter.
- .plt: contains the procedure linkage table.
- .strtab: contains string table of .symtab section.
- .symtab: contains global symbol table.

2.2.4 Usage

The binary executable produced by the compiler can now be used to run the program. Having the program in this new file does not remove the utility of the source code. Indeed, a compiler can be executed with multiple flags to have a more specific executable. For example, a program might need to be tested before its release. To do so, memory errors can be discovered using a feedback-guided fuzzer combined with sanitization.

To check the memory, *Address Sanitizer* (ASan) [12] can be used by compiling the source code with flags that will tell the compiler to add instrumentation. *ASan* is a memory error detector. It can help find issues like use after free, heap buffer overflow, or stack buffer overflow. The typical slowdown introduced is 2x.

In the case of an end-user wanting to test a closed source software, it is nearly impossible to get the program compiled with the instrumentation needed. Tools such as binary rewriters can be used to face this issue.

2.3 Binary rewriting

Binary rewriting consists of modifying an executable in the absence of source code while maintaining functionality. It can be used for multiple reasons for example for program optimization, program obfuscation, or as in the case of Retrowrite for security policy enforcement via instrumentation.

Binary rewriters can be divided into two main approaches: dynamic instrumentation and static instrumentation.

2.3.1 Dynamic instrumentation

In this scheme, the rewriting is done while the program is executed. Usually, the binary is executed side by side with the rewriter engine. To analyze and interact with the program during the execution. The rewriter might use the OS's primitives as the PTRACE API in Linux for example. An advantage of dynamic instrumentation is that analyzing the whole binary might be avoided. This is practical for large binaries. However, the disadvantage of using this technique is the high-performance cost. Translating the program at runtime does not come with no cost and the overhead is way higher than running a modified binary executable [3].

2.3.2 Static instrumentation

Contrary to dynamic instrumentation, static rewriters operate on binaries at static time. It outputs a new binary executable that corresponds to the initial executable but is rewritten. This generated executable has the advantage of being almost as fast as the original executable that would have been compiled for instrumentation as the overhead introduced is usually low.

2.3.3 Steps in binary rewriting

Both dynamic and static instrumentation follows four main steps:

1. Parsing: This step consists of obtaining the instruction stream from the binary and passing it to a disassembler.
2. Analysis: This part consists of gaining as much information as possible. The objective is to recover the structure of the source code.
3. Transformation: Here, we must find all the locations where instrumentation must be added or removed.
4. Code generation: We have to make sure that adding the instrumentation does not modify the behavior of the executable. The modified executable must be sound. And then the original executable must be patched or a new executable must be generated. There are multiple techniques for this part.

Both dynamic and static instrumentation follows four main steps:

1. Parsing: This step consists of obtaining the instruction stream from the binary and passing it to a disassembler.

2. Analysis: This part consists of gaining as much information as possible. The objective is to recover the structure of the source code.

3. Transformation: Here, we must find all the locations where instrumentation must be added or removed.

4. Code generation: We have to make sure that adding the instrumentation does not modify the behavior of the executable. The modified executable must be sound. And then the original executable must be patched or a new executable must be generated. There are multiple techniques for this part.

2.3.4 Transformation and Code generation techniques

There are multiple techniques for the transformation and code generation parts. The simplest technique is called *Trampoline*. All new codes are added in new sections in the binary. Branches are then added to redirect to the right location after an instrumentation point. This technique introduces too much overhead as we need to jump from and to the trampoline at each instrumentation.

Another technique is named *Direct*. This is one of the oldest techniques. Here the code is simply overwritten. We have to make sure to readjust every branch and reference. But this technique does not scale well as adding and removing instructions implies an update of all branch targets.

The technique used by Retrowrite is called *Symbolization*. It consists of transforming the binary into an assembly text file. It transforms all reference constants into assembly labels. Then, many tools can insert instrumentation into the reassembled file. Unlike the trampoline technique, this method is defined as being *zero-overhead* as the only overhead added is the time to execute the instrumentation.

2.4 Retrowrite

Retrowrite [3] is a binary rewriter developed by HexHive. It uses the symbolization technique by generating re-assemblable assembly. Retrowrite has as objective to be sound by the way they determine reference constants and symbolize them. It uses *Capstone* as a disassembler and *pyelftools* as a binary parser.

2.5 Other security techniques

C/C++ are two of the most used programming languages. They are mainly used for their runtime performance and low-level memory access capabilities, but this comes with a cost. In these languages, memory and type safety are managed by the programmers. Tools can be used to enforce the safety of software. For example, one might want to use *HexType* to enforce type safety.

2.5.1 HexType

HexType is a tool written by the *HexHive* lab to detect type confusion in software written in *C++*. Typecast is when a pointer of a certain type is converted into another. Typecast are normally checked statically. The vulnerability lies in the fact that down-casting is allowed. Down-casting means transforming a pointer representing a class into one of its descendants as seen in the code example. This can lead to type confusion as no one prevents the program to cast an object of an incompatible base type. *HexType* remedies this issue by transforming static checks into runtime checks.

```
1  class Animal {
2      public:
3      int age;
4  };
5
6  class Dog: public Animal {
7      public:
8      int id ;
9      virtual void bark();
10 };
11
12 int main(void){
13     Animal *animal_ptr = new Animal;
14     Dog *weird_dog = static_cast<Dog*>(animal_ptr); // Type confusion.
15     weird_dog->id = 0x43; // Memory safety violation!
16     weird_dog->bark(); // Control-flow hijacking
17 }
```

Listing 2.1: Example of type confusion

As you can in the code above, the Dog class derives from the Animal class. Although a dog is an animal, we see that an animal cannot specifically be seen as a dog. An animal does not have an "id" attribute and therefore accessing it can be considered a memory safety violation.

Chapter 3

Goal

As stated in their paper [3], *Retrowrite* was firstly designed to support Linux x86-64 PIC binaries that were written in the *C Programming Language*. The main obstacle to adding support for *C++* is that the symbolization of C++ exception handlers is not yet supported. Multiple works were done to add support for C++ as it can be seen in the *Retrowrite* repository [6].

3.1 Challenges

When Retrowrite was presented to the public, C++ was not yet fully supported. C++ is a more versatile language than C, it seemed obvious that the support of the latter was essential to make Retrowrite a complete tool. Our first intention was to integrate Hextype into Retrowrite. It turned out then that the task was more difficult than expected. Since not having the source code, it was complicated to obtain the information on the classes, as this information often resides in the debug section. It would be possible to do something if debugging information was available, but this is not the case for many executables. Similarly, pyelftools cannot analyze executables with a version of gdwarf < 4.0, whereas the standard is version 5 today. We can compile our programs with gdwarf 4 to test the programs but that wouldn't make sense in the real world where the standard is different.

Then, *Hexhive* released a version of Retrowrite on their repository where C++ support was added. So we decided to focus on testing this new feature. We focused on testing Retrowrite on as many packages as possible but didn't pay attention to whether the modified file behaved like the original file.

3.2 Task

The task was to test many packages from the *Debian* [9] distribution. Debian being a widely used distribution, we thought it would be a good idea to use it as a source. We tested Retrowrite on a bunch of programs that can be found on Debian. The tests were not extensive as we were only testing if the command *–help* worked. The purpose of the testing was to see if the compiler was able to compile the assembly file generated by Retrowrite.

Chapter 4

Implementation

The main script is *clone.sh*. It will start with several containers which will each take care of one package. This script also takes care of running the command that will prepare the results.

4.1 Docker

The first decision was to test *Retrowrite* in *docker* [8] containers. This allows us to emulate *Debian* on any machine. Each container will execute *entrypoint.sh* on a specific package. The binaries available from the command *apt install* [13] are stripped, so we had to download the source code and compile the program ourselves. Then we ran the program with the argument "*–help*" and compared the exit code with the recompiled program from the assembly file re-assembled by Retrowrite. The programs are then sorted according to their output code.

4.2 Linking

One problem encountered was recompiling the assembly files by linking them to the correct shared libraries. To do so, a script was written called *flib.sh* which takes as an argument the original binary file. It will then return, using the command "ldd"[7] the list of libraries to link.

4.3 List of packages

Having obtained a list of packages from the Debian distribution [2], we needed a way of filtering to only obtain a list of packages written in *C++*. To do this, we use the script *filter_c++_packages.sh*.

It will simply look in the information given in *apt show* [13] if the package was written in *C++* or not.

4.4 Filtering the result

When the docker containers have finished executing their commands, the *result_count.sh* script will be executed. This script takes care of searching and sorting the results by arranging them and putting them in a new folder.

4.5 Use of exceptions

We also needed a way to find which binaries were using exceptions. Using pyelftools, we noticed that all binary files using exceptions also had a call instruction to `__cxa_begin_catch@plt` function or `__cxa_allocate_exception@plt`, both in the plt section. So we just looked in the *.rela.plt* section and search for the `__cxa_allocate_exception` symbol. The script is named *elfexceptions.py*

All the scripts used can be found on the GitHub repository [5].

Chapter 5

Evaluation

In Table 5.1, you can see the result of our scripts on the most popular packages in *Debian*. The Table 5.2 corresponds to the remaining packages found in the other list. The *discard* line corresponds to the packages where the program was not testable, e.g., running them with –*help* leads to a crash or an error. The category "*unsuccessful compilation*" means that we were not able to compile the re-assembled assembly back to an executable for various reasons as missing shared libraries or the assembly file contains some error. It was not possible to categorize these errors as *clang* always outputs the same error in each situation: "`<unknown>:0: error: Undefined temporary symbol .L26312 `". The "`runtime error`" category corresponds to when we succeed to run the modified executable but a segmentation fault was encountered.

category	#packages	percentage
discard	302	-
no errors	243	40.0%
unsuccessful compilation	349	57.4%
runtime error	16	2.6%
total	910	

Table 5.1: Popular packages

category	#packages	percentage
discard	109	-
no crash	172	35.8%
unsuccessful compilation	306	63.7%
runtime error	2	0.41%
total	589	

Table 5.2: Remaining packages

As you can see in these figures, the most dominant category is the "*unsuccessful compilation*" one. Only a few programs crashed at runtime. We conclude that an improvement can be made in the assembly file generation but the good thing is that very few programs crash once the executable is compiled.

By reading the Retrowrite paper [3], we understand that the biggest issue with adding *C++* support was handling symbolization of exception handlers. It seems that is not yet fully supported as testing a simple program using exception leads to a runtime error:

```
#include <iostream>
using namespace std;

int main (int argc, char *argv[]) {
    try
    {
        if (argc < 2){
            throw "No argument given !";
        }else {
            cout << argv[1] << '\n';
        }
    }
    catch (const char* e)
    {
        cout <<"Error: "<< e << '\n';
    }
    return 0;
}
```

Listing 5.1: Usage of exception handlers

Retrowrite succeeded to rewrite the executable but running it leads to a segmentation fault. We wanted to see for all the packages, which one fails and was using exceptions. The result can be seen in Table 5.3.

As you can see, most of the segmentation faults were encountered with programs using exceptions. The difference is not that big, but we see that the compilation was more successful when the program was using exceptions. We understand from the test done that the program crashes only if he gets in a catch clause.

category	#packages	percentage
crash during compilation	199	64.6%
no crash	92	29.9%
crash during runtime	17	5.5 %

Table 5.3: Packages using exception

category	#packages	percentage
crash during compilation	427	57.2%
no crash	318	42.6%
crash during runtime	1	0.1%

Table 5.4: Packages not using exceptions

Another observation that can be made is that the size of the executable matters. As you can see in Figure 5.1, the number of executables that could be compiled and run drops drastically to 0 as the file size increases. In the other direction, the number of errors at the compilation level rises in parallel to the size of the file. Similarly, the number of runtime errors also increases with file size, but we believe this is more related to exception usage.

Take into consideration that the file sizes shown here correspond to stripped executables. But we used Retrowrite on their non-striped versions. Unfortunately, we no longer had the unstriped versions, and recompiling each package would have taken us too long. These figures given are therefore only representative.

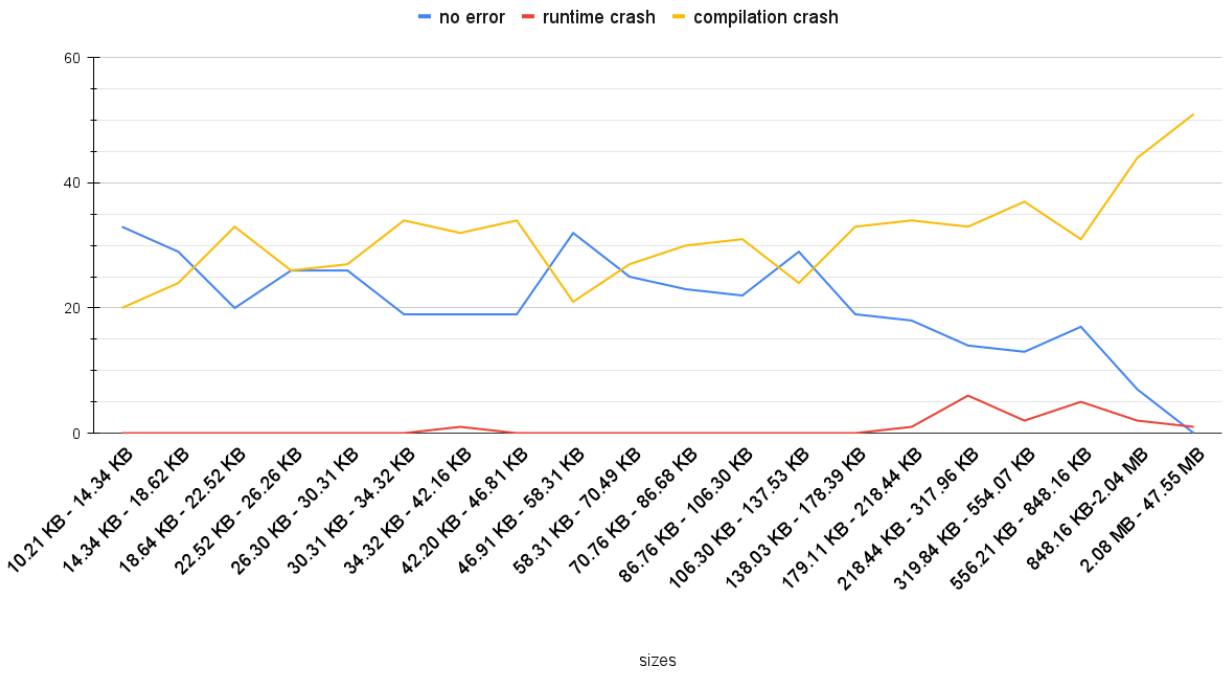

Figure 5.1: classification of packages according to their size

sizes	no error	runtime crash	compilation crash
10.21 KB-14.34 KB	33	0	20
14.34 KB-18.62 KB	29	0	24
18.64 KB-22.52 KB	20	0	33
22.52 KB-26.26 KB	26	0	26
26.30 KB-30.31 KB	26	0	27
30.31 KB-34.32 KB	19	0	34
34.32 KB-42.16 KB	19	1	32
42.20 KB-46.81 KB	19	0	34
46.91 KB-58.31 KB	32	0	21
58.31 KB-70.49 KB	25	0	27
70.76 KB-86.68 KB	23	0	30
86.76 KB-106.30 KB	22	0	31
106.30 KB-137.53 KB	29	0	24
138.03 KB-178.39 KB	19	0	33
179.11 KB-218.44 KB	18	1	34
218.44 KB-317.96 KB	14	6	33
319.84 KB-554.07 KB	13	2	37
556.21 KB-848.16 KB	17	5	31
848.16 KB-2.04 MB	7	2	44
2.08 MB-47.55 MB	0	1	51

Table 5.5: classification of packages according to their size

We also chose to test a program not using exceptions to see exactly how the newly generated executable behaves. We have chosen as program fish-shell [4]. Unfortunately, we were unable to recompile the program. The errors obtained lead us to understand that there was an error concerning the shared library libstdc++.so.6. Despite our research, it was quite difficult for us to find the exact cause of the problem. We leave this as future work.

```
/usr/bin/ld: _ZSt15__once_callable: TLS definition in /usr/lib/libstdc++.so.6
    section .tbss mismatches non-TLS reference in /tmp/ccAGaHdi.o
```

During our tests, we were able to discover some problems and reported them to the developers. The first is that some assembly files can only be transformed into an executable format using clang and not GCC. Another issue was that the compiler seemed to not recognize certain sections in the assembly file. A solution was then offered in the comments when the issue was written where a user simply proposed to remove all references to these sections.

Chapter 6

Conclusion

In general, Retrowrite worked on a lot of packages, and we encountered few runtime errors. The main problem was that it was not possible to compile the re-assembled assembly back on most of the packages. The results also showed that Retrowrite can not handle exceptions yet. Similarly, Retrowrite seems to encounter errors with large file sizes. We also had to find the shared-libraries ourselves. It would be an enhancement if Retrowrite could do it for x86 CPU as it seems that this option is only available for *ARM* cpu [1].

Bibliography

[1] Luca Di Bartolomeo. “ArmWrestling: efficient binary rewriting for ARM”. In: 2021.

[2] *Debian Popularity Contest*. https://popcon.debian.org. Accessed: 2022-06-01.

[3] 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.

[4] fish-shell. *fish-shell*. https://github.com/fish-shell/fish-shell. 2022.

[5] Hassan Habib. *debian docker*. https://github.com/ha2san/debian_docker. 2022.

[6] HexHive. *Retrowrite*. https://github.com/HexHive/retrowrite. 2022.

[7] *ldd(1) Linux User's Manual*. Aug. 2021.

[8] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.

[9] Debian News. *Updated Debian 11: 11.3 released*. https://www.debian.org/News/2022/20220326. Accessed: 2022-06-01. Mar. 2022.

[10] Oracle. *SPARC Assembly Language Reference Manual*. https://docs.oracle.com/cd/E53394_01/html/E54833/elf-23207.html. Accessed: 2022-06-01. Apr. 2020.

[11] *readelf(1) Linux User's Manual*. Jan. 2021.

[12] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. “AddressSanitizer: A Fast Address Sanity Checker”. In: *2012 USENIX Annual Technical Conference (USENIX ATC 12)*. Boston, MA: USENIX Association, June 2012, pp. 309–318. ISBN: 978-931971-93-5. URL: https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[13] APT team. *apt(8) Linux User's Manual*. Jan. 2019.