Thesis Dissertation

Evaluating Randomized Binaries

Antonis Yenagritis

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

December 2018

UNIVERSITY OF CYPRUS

COMPUTER SCIENCE DEPARTMENT

Evaluating Randomized Binaries

Όνομα Φοιτητή

Supervisor

Dr. Elias Athanasopoulos

Thesis submitted in partial fulfilment of the requirements for the award of degree of Bachelor in Computer Science at University of Cyprus

December 2018

Acknowledgements

I would like to express my sincere gratitude to my thesis advisor Dr. Elias Athanasopoulos for guiding me throughout my research.

I am also grateful for my family and friends who provided me with support and continuous encouragement throughout the process of the thesis.

Abstract

One promising technique for countering code-reuse attacks is Execute-only Memory. This makes the code layout unreadable and thus attackers cannot locate any more malicious gadgets. For enforcing Execute-only Memory, randomizing the code layout of the process is essential. To this end, several frameworks have been proposed for fine-grained randomization of code. In this thesis we explore one of them, namely CCR, and measure the parts of code that are not randomized at all. Furthermore, we develop an LLVM pass, perform static analysis of C/C++ source code and then we measure which parts remain untouched by CCR. In conclusion, we find that CCR cannot perform any in-function randomization in 1-BB functions and cannot randomize virtual methods in C++ programs.

Contents

Chapter 1

Introduction

Control-flow attacks are one of the many problems a software developer has to face. A single bug in a code can be an open door for software exploitation. That is why many resources have been devoted to the research of a defence mechanism against this kind of threat. That being said, Return Oriented Programming (ROP) is one of the most-used control-flow attacks. Furthermore, a widely known mechanism for defending against ROP is code randomization. By rearranging the code of the programme it becomes much harder for a bug to become useful for exploitation. This led to the use of different approaches of code randomization such as the Compiler-assisted code randomization (CCR) method.

Like any other code randomization technique, the CCR has to be assessed so that it's level of defence that is responsible for "giving" can be evaluated. This kind of evaluation is very important because it reveals the different weaknesses each technique possesses and gives the pros and cons of its use. Subsequently, this can be especially crucial for CCR because of its different approach.

This evaluation however proves extremely challenging because it demands a heavily reverse engineering of C++ code as well as extensive knowledge on types of interpreters like Bash. In the case of this paper, the code was used to test the CCR in order to assess how it reacts. Further, there was a focus on how to change the code in order to track any changes in the CCR. Another example of reverse engineering in research might revolve around a piece of software that has been running for years and is responsible through its code for a number of functions in a business [26]. By using reverse engineering one can be able to detect and deal with vulnerabilities in order to protect the software. Lastly, when the need arises to maintain parts of a software or even enhance or reuse them, proficiency in reverse engineering is needed. Nevertheless, in this paper the main idea was to test how the CCR reacts in different changes to instances in the code. This way the different weaknesses of this defence mechanism become visible.

Through the experiments and trials presented in this thesis the weakness that was found in CCR's approach revolves around the virtual tables. The fact that the vtables always have the same fixed addresses regardless of the randomization in the rest of the code clearly creates a situation that can be exploited.

As an answer to new defences against control-flow attacks, vtable hijacking has been adopted by many attackers. With this approach, the attacker takes advantage of bugs in C++ programs to overwrite pointers in the vtables of objects. This has led to the need for methods that pay attention in the ordering and interleaving of vtables in the memory. This way suggests that by assessing the validity of a vtable at runtime is more efficient than a set membership test [25].

The following thesis starts by presenting some background work on LLVM Compiler Infrastructure and then describes what is an LLVM pass and where it can be used. Further, the Clang compiler is introduced and its relationship with the LLVM compiler infrastructure is explained. Following this, there are brief explanations on the Docker and the use of SQLite3 in the creation and management of databases. After the background work, the research problem of interest is presented and briefly discussed along with 3 examples. Moreover, a small explanation on how to defend against certain attacks is presented and the non-executable storage space is introduced with the ultimate purpose to evaluate the approach of randomization tools against ROP attacks. The first step to this approach is then discussed in chapter 4.1 “LLVM Pass and Basic Block Count” and it is accompanied by an explanation on function randomization using SQLite3 and the “objdump” command. Further, an insight is provided on the meaning and use of Virtual Functions and Virtual Tables and more examples are presented to further support this thesis’ idea. Naturally, the implementation of the research is presented with details on the LLVM Pass and a line-by-line explanation of the Bash script used. Following this, the Evaluation section, briefly explains the output of our approach and points out the vulnerability in the method. Finally, related work is presented that focuses on research around code randomization functions, code diversification and use of execute-only memory.

Chapter 2

Background

2.1 LLVM Compiler Infrastructure

The LLVM compiler infrastructure project is a "collection of modular and reusable compiler and toolchain technologies" [18] used to develop compiler front ends and back ends. LLVM is written in C++ and is designed for compile-time, link-time, run-time, and "idle-time" optimization of programs written in arbitrary programming languages. LLVM can provide the middle layers of a complete compiler system, taking Intermediate Representation (IR) code from a compiler and emitting an optimized IR. This new IR can then be converted and linked into machine-dependent assembly language code for a target platform. LLVM can accept the IR from the GNU Compiler Collection (GCC) toolchain, allowing it to be used with a wide array of extant compilers written for that project.

2.2 LLVM Pass

The LLVM Pass is an operation (procedure invocation) on a unit of LLVM IR code. The granularity of code operated on can vary from a Function to an entire program (Module in LLVM parlance). Passes may be run in sequence, allowing a successive pass to reuse information from (or work on a transformation carried out by) preceding passes. The LLVM pass framework provides APIs to tap into source-level meta-data in LLVM IR.

2.3 Clang

Clang is a compiler front end for the C, C++, Objective-C and Objective-C++ programming languages, as well as the OpenMP, OpenCL, RenderScript and CUDA frameworks. It uses the LLVM compiler infrastructure as its back end. It is designed to act as a drop-in replacement for the GNU Compiler Collection (GCC), supporting most of its compilation flags and unofficial language extensions. Clang is intended to work on top of LLVM. The combination of Clang and LLVM provides most of the toolchain, to allow replacing the full GCC stack. Because it is built with a library-based design, like the rest of LLVM, Clang is easy to embed into other applications.

2.4 Docker

The Docker is a computer program that performs operating-system-level virtualization, also known as "containerization". The Docker is developed primarily for Linux, where it uses the resource isolation features of the Linux kernel to allow independent "containers" to run within a single Linux instance, avoiding the overhead of starting and maintaining virtual machines (VMs)

2.5 SQLite3

A free to use database that gives you the ability to easily create and manage databases is SQLite3. Early learners of SQL as well as developers find SQLite3 ideal because it provides them with a simple database engine to port in their applications. It should be noted that even though SQLite3 is not a full-featured database, it has support for a large set of the commonly used SQL standard.

Chapter 3

Research Problems

3.1 Software Exploitation

Many programs often have bugs in their codes. Some are small and unnoticeable while others can leave open doors to people with malicious intents. These malicious intents can be materialised using *software exploitation.* A software exploitation is when someone takes advantage of those bugs to gain read or write privileges in a specific programme. Moreover, a bug that gives read privileges, if used the right way, it can give a person the ability to extract information stored in the program's storage space. If the information targeted is of personal character (e.g. password), it can have a big impact on the user. On the other hand, a bug that gives write privileges can be used to store information in the program's storage space.

An example of use of such bug is known as *buffer overflow*. As its name suggests, buffer overflow is when the value given to a specific variable in a program exceeds the predetermined length. If left unchecked, this can lead to the variable being saved over other nearby important values.

```
int password_valid = 0;

void authenticate_root(char *passwd) {
        unsigned long marker = 0xdeadbeef;
        char password[16];

        strcpy(password, passwd);

        fprintf(stderr, "%p\n", &marker);
        fprintf(stderr, "Validating password: %s\n", password);

        if (!strcmp(password, "e5ce4db216329f4f"))
                password_valid = marker;

}

int main(int argc, char *argv[]) {

        authenticate_root(argv[1]);

        if (password_valid != 0) {
                printf("Welcome administrator.\n");
        } else {
                printf("Access denied.\n");
        }

        return 1;
}
```
Example 1: Program with buffer overflow bug

In example 1 we see a program that takes a string as argument. It sends the string to the function 'authenticate_root' that checks if it is the same with the correct password. If it is not the same the programme gives the “Access denied” as output and otherwise it gives “Welcome administrator”.

We can see that (in example 1) the function ‘authenticate_root’ has a bug that gives write privileges. The function takes as an argument a string with unspecified length. Then it declares a table of characters and allocates to it space for 16 characters. Then it uses the function ‘strcpy’ to copy the argument into the table. By doing this the argument is saved into the stack but without checking if the given string has the right length.

```
* 0xffffd298:  0xf7e0de18       0x41414141       0x41414141       0xff004141
* 0xffffd2a8:  0xf7ffcd00       0xdeadbeef(m)    0x00000000       0xffffd384
* 0xffffd2b8:  0xffffd2d8(ebp) 0x08048565(ret) 0xffffd52f       0xffffd384
* 0xffffd2c8:  0xffffd390       0x080485c1       0xf7fb53dc       0xffffd2f0
* 0xffffd2d8:  0x00000000       0xf7e19276       0x00000002       0xf7fb5000
* 0xffffd2e8:  0x00000000       0xf7e19276       0x00000002       0xffffd384
* 0xffffd2f8:  0xffffd390       0x00000000       0x00000000       0x00000000
* 0xffffd308:  0xf7fb5000       0xf7ffdc04       0xf7ffd000       0x00000000
```

Example 2: Stack memory representation

In example 2 we see the structure of the stack when we use the argument "AAAAAAAAAA" into the program from example 1. The character ‘A’ is translated into ‘41’ in hexadecimal to be saved in the stack. The value marked with ‘ret’ is the return value that the function is going to use after it finishes, to return to the correct flow of the program. If we continue to add “A”s in our argument, we can overwrite all the values in the stack until we reach the return address. With this opportunity we can change the value of the return address and direct the flow of the program to the point where it benefits us. For the above example we can direct it to the point where our argument is accepted from the program instead of being rejected. This is also known as *Control Flow* attack.

By taking this a step further we can do what is known as *Code Injection*. Just like Control flow, the Code Injection is based on changing the value of the return address. However, instead of directing the flow of the programme towards another line of code we direct it to the storage space.

```
   0:        31 c0                    xor    %eax,%eax
   2:        50                       push   %eax
   3:        68 2f 2f 73 68           push   $0x68732f2f
   8:        68 2f 62 69 6e           push   $0x6e69622f
   d:        89 e3                    mov    %esp,%ebx
   f:        31 c9                    xor    %ecx,%ecx
  11:        31 d2                    xor    %edx,%edx
  13:        b0 0b                    mov    $0xb,%al
  15:        cd 80                    int    $0x80
```

Example 3: Assembly code translated to hexadecimal form

In example 3 we took some lines of code that execute a specific process and we converted them into hexadecimal. Now we repeat the same process as example 2 but use these numbers as our input string. Then if we direct the return address to the address where our code is saved then the programme will start executing it. This way we forced the programme to execute code that was not pre-programmed to do, just by taking advantage of a bug and using a specific input string.

3.2 How to Defend

One of the ways that can be used to defend against these kinds of attacks is by making the storage space *Non-Executable*. If the programme is prevented from executing code that is forced into its memory, then the Code Injection attack can't work.

To overcome this obstacle, a new technique called Return Oriented Programming (ROP) Attack was invented. This kind of attack is based around the idea of code reuse. Instead of injecting the code that we need into the memory we instead force the programme to run its own lines of code, but in the order that benefits us. The first thing that is needed for this attack is to find the Gadgets within the lines of the programme. Gadgets are a small number of commands that are next to each other and always end with the command "return". Once we locate these Gadgets, we take note of the address of their first command. Next, we decide the order we want those Gadgets to be executed. Then we follow the same process we used in Control Flow attack and we insert the addresses we have in the programme's memory in the order that executes our goal. Lastly, we overwrite the return address with the address of our first gadget. From that point on, every time the programme reaches the "return" command it will read our address and go to the next Gadget until we succeed on our goal.

3.3 Defending ROP

In order to counter this kind of threat the randomization tools were invented. The job of these tools is to take the code of a given programme, break it in pieces and put those pieces in different places in the memory. Every time the programme runs each of its parts have a different address in the memory. This makes the search for the Gadgets much harder since they have different address every time the programme is executed.

One such tool is the Compiler-assisted code randomization (CCR). Relying on compiler-rewriter cooperation, CCR can be described as a generic and highly efficient code transformation approach that aims for fast and robust diversification of binary executables on end-user systems. It uses a minimal set of metadata that can be embedded into executables to facilitate rapid fine-grained code randomization at the basic block level and maintain compatibility with existing mechanisms that rely on referencing the original code. The motivation behind this design is based on two ideas. First, taking into account the variable of practicality, the code diversification approach should allow existing models and features to run as initially planned. However, code randomization can be described as a highly intrusive technique. Which points to the second idea behind its design that supports the importance of compatibility. To avoid potential conflict caused by code randomization with already set operations the compiled libraries are enhanced with metadata that allow their upcoming randomization to be executed during load time or installation. Randomization at load time is presented as a part of the loader's modifications to code. It should be noted though that the longer rewriting time required for this method could result in user-perceived delays. In order to avoid this issue, keeping a reserve of pre-randomized variants, for example a service generating them in the background could be a solution [18]. Further, most software is presented in the form of compiled binaries and during installation on each endpoint it is subjected to some customization (i.e. post-processing). In this case the randomization can take the form of a post-processing task during installation [18].

3.4 This Thesis

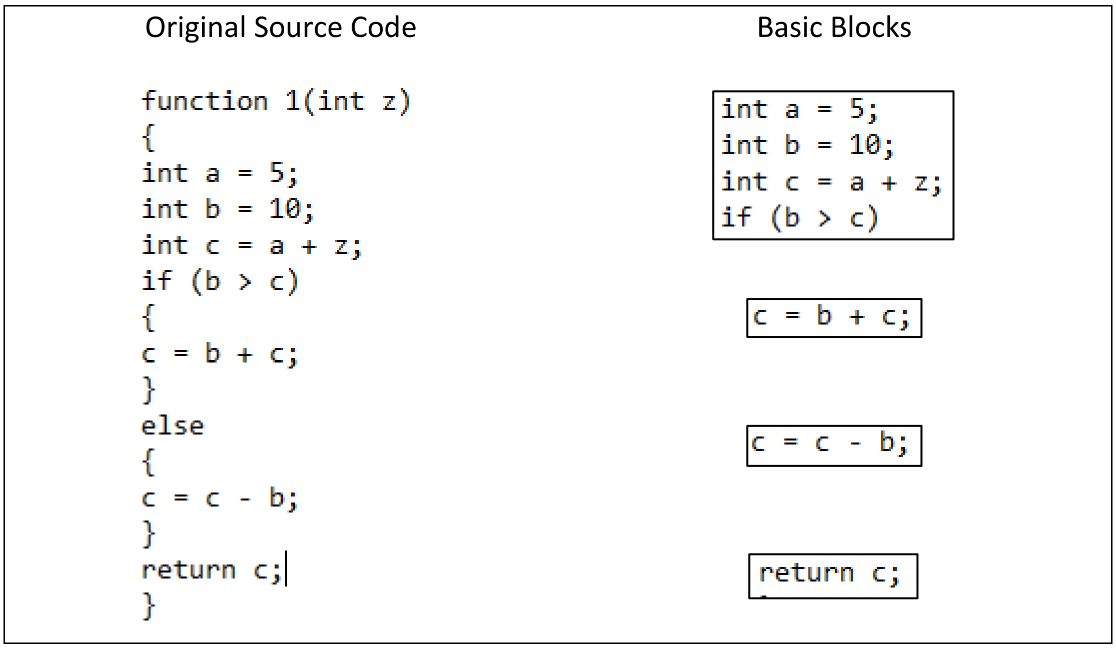
Our objective in this thesis is to explore and test the CCR-binary in order to find hidden weaknesses in the code randomization. We achieve this by measuring part of the code remain untouched by the randomization of the CCR.

Chapter 4

Architecture

4.1 LLVM Pass and Basic Block Count

The first step of our work begins by finding the Basic Blocks (BBs) in the functions of our target program. Basic blocks are a sequence of code that is executed one after the other and have no branches except from one entry and one exit point. During the compilation of the program the compiler creates those basic blocks by bundling together consecutive instructions until it reaches a 'goto' or a 'jump'. The 'goto/jump' instruction is used to change the flow of the program in case of a loop or to call and return from a function. If the addresses of those basic blocks are known by an attacker, then they can be used as gadgets.



```
function 1(int z)
{
int a = 5;
int b = 10;
int c = a + z;
if (b > c)
{
c = b + c;
}
else
{
c = c - b;
}
return c;
}
```

```
int a = 5;
int b = 10;
int c = a + z;
if (b > c)
```

```
c = b + c;
```

```
c = c - b;
```

```
return c;
```

Example 4: Function broken down to basic blocks

For this reason, the CCR creates metadata during the compilation that are then used to change the positions of those basics blocks within the function by shuffling them. With this, the address of the BBs in a function keeps changing every time the attacker runs the executable making it difficult for him to use them. However, this raised the question of what happens when a function has only one BB.

To answer this question, we used the LLVM Infrastructure to create an LLVM Pass that is going to be used by the Clang compiler (point 2.3). During the compilation of our target program, Clang will call our pass each time it encounters a function. The job of our LLVM Pass is to count the BBs that are created by the compiler within the function and give them to us as output together with the function's name.

4.2 Function Randomization

Using SQLite3 we insert the final output of our pass in a temporary database in order to make them more manageable. SQLite3 gives us the option using a simple query to search in our result for the functions that have only one BB. Now that we have identified which functions we need to focus on, we will use the command objdump.

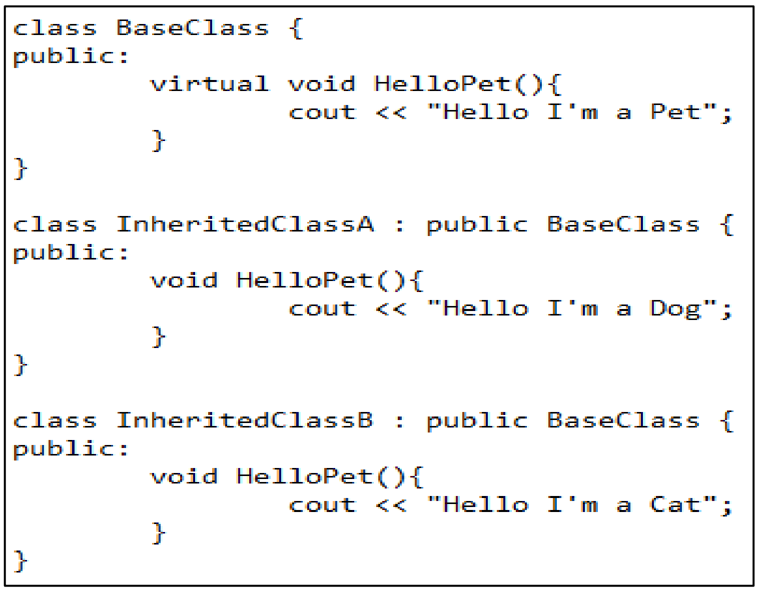
Objdump is a command line program that can be used as a disassembler on an executable and gives the user the ability to look at it in assembly form. This kind of view can be very useful because it shows various information about the content of an object that you can't find in the actual code. In our case, we will use it to observe addresses of our specific functions in the memory to find out what happens after each randomization.

After a couple of tests, we notice that the address of each function changes every time we run a new randomization. This means that CCR not only moves the BBs within the function but also moves the entire function within the executable's memory making it even harder for someone that searches for gadgets.

4.3 Virtual Functions and Virtual Table

The next important part of the process are virtual functions. In languages were object-oriented programming can be achieved (e.g. C++), the virtual functions or virtual methods are mainly used to achieve runtime polymorphism. Polymorphism is when an object or function is declared only once but has many different versions that are used for different purposes. For example, a "shape" can either be a triangle, a square or a circle.

A virtual function is declared in the basic class of a program. Each derived class can then redefine this function and have its own version that works accordingly with the needs of the class. When a reference is used to an object of that derived class from the basic class then the virtual method makes sure that it invokes the version that is associated with that object's class.



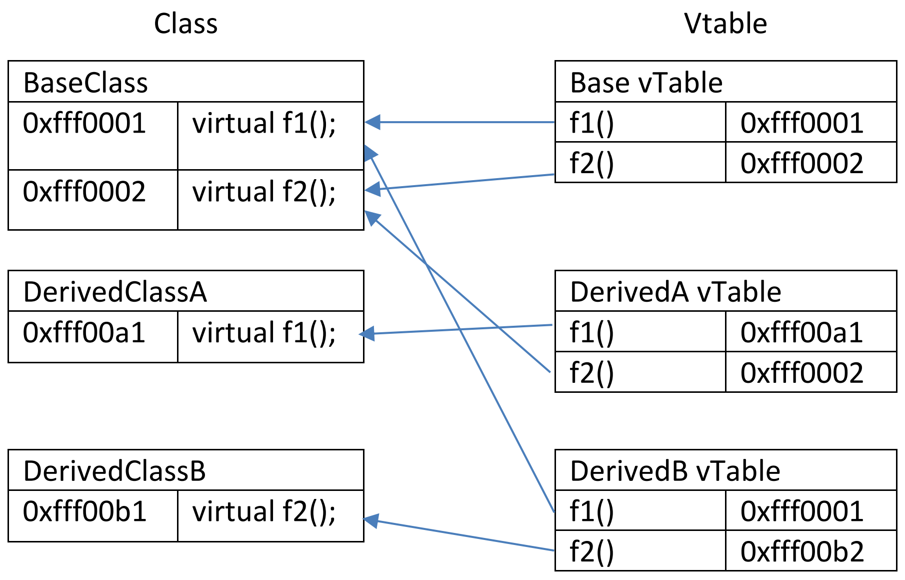
```
class BaseClass {
public:
        virtual void HelloPet(){
                cout << "Hello I'm a Pet";
        }
}

class InheritedClassA : public BaseClass {
public:
        void HelloPet(){
                cout << "Hello I'm a Dog";
        }
}

class InheritedClassB : public BaseClass {
public:
        void HelloPet(){
                cout << "Hello I'm a Cat";
        }
}
```

```
int main (int argc, char* argv[]){

BaseClass Object1 = new BaseClass
BaseClass Object2 = new InheritedClassA
BaseClass Object3 = new InheritedClassB

Object1->HelloPet(); //output: "Hello I'm a Pet"
Object2->HelloPet(); //output: "Hello I'm a Dog"
Object3->HelloPet(); //output: "Hello I'm a Cat"

return 0;
}
```

Example 5: Virtual Functions

In Example 5 we see that the BaseClass declares and defines a virtual function. The classes InheritedClassA and InheritedClassB read the virtual function from the BaseClass and they redefine it. When an object from each class calls on that function then only the appropriate version of that function is called.

In order to make sure that the right version of the virtual method is called, most compilers use a virtual table. Each time a virtual method is defined the compiler adds a hidden variable to it that points to a table of pointers for each individual virtual function. The reason behind this is that at compilation it is not clear if the function from the base class is called or if it is one of the implementations in the derived classes. This can only be known at run time and the virtual table helps in dynamically making the calls to the right function. These tables can be very common in languages such as C++.



Example 6: Virtual Tables

In example 6 we see 3 classes with their own virtual tables. The vTables save the address of the correct version of each function for each class. In case of DerivedClassA when it needs to call the function f1() it will then execute its own version. But if it needs to call the function f2() then the version from BaseClass will be executed. The same logic applies to DerivedClassB.

For the sake of our experiment we must also test how the CCR handles the virtual functions within a C++ program. And after a couple of tries we found that they are handled exactly the same as any other regular function. Even if they have 1 or more BBs the CCR changes their position during the randomization.

We should note, however, that the virtual tables can’t be examined with the use of the objdump as we did with the functions. To check how the CCR behaves with the virtual table we must use the GNU Debugger (GDB).

GDB’s purpose is to give to the user the ability to monitor a line by line execution of the wanted program. Among other services the GDB offers, the user can also examine and alter the values of the internal variables of the program as well as change its execution. For example, change call functions that are not in the program’s normal execution sequence. The GDB help a lot when it comes to solving questions about the output of a program or finding hidden mistakes made by developers. It can be used for debugging many programming languages like C or C++ and runs on many Unix-like systems.

By using GDB for our experiment we were able to look on how the virtual table was handled by CCR. From our research we found that the vtable remains untouched by the randomization of CCR. This means that it can always be found at the same address in memory and the virtual functions addresses that it has always appear in the same order. This creates a vulnerability to CCR’s randomization because if someone knows where he can find all the addresses for the program’s gadgets then he can use them for his own gain.

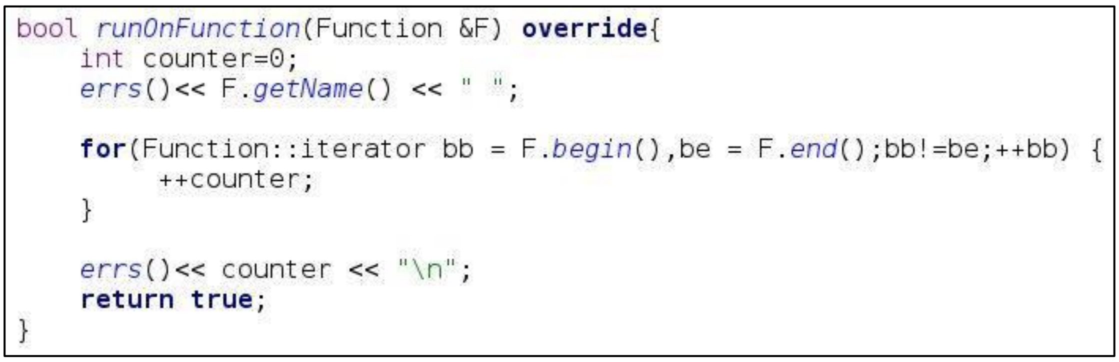
Chapter 5

Implementation

5.1 LLVM Pass

The first step for the implementation of our proof of the above conclusion is the creation of the LLVM pass that returns the Basic Block count of each function of our programme. The LLVM pass is created in the LLVM source folder under the path "../llvm/lib/Transforms". There we create the folder that will host our pass. Inside the folder we create 3 files. A "Pass.cpp" file that our pass will be written and 2 files that are necessary for our pass to run. A "Pass.export" file that is always empty and a "CMakeList.txt" file that has a standard code for all the passes. The only difference in our CMakeList is that it needs the names of our "export" and "cpp" files. Its main use is to enable LLVM when it' s built to create a vector to our pass.

Inside the "Pass.cpp" file we create a function that is called every time our pass encounters a new function in the target programme during its execution. The function consists of a counter, 2 error messages that show the output to the terminal and a loop that increases the counter. The first message gives the name of the function we are working on. Then the loop starts from the 1st basic block of the function and ends at its last basic block and along the way it increases the counter. Lastly the second message gives the final value of the counter.



```
bool runOnFunction(Function &F) override{
    int counter=0;
    errs()<< F.getName() << " ";

    for(Function::iterator bb = F.begin(),be = F.end();bb!=be;++bb) {
        ++counter;
    }

    errs()<< counter << "\n";
    return true;
}
```

Figure 1: LLVM Pass function

5.2 Bash Script

The script starts taking into consideration the fact that Unix systems have a path that includes the addresses of all the libraries needed for the execution of basic functions. The command "export PATH" is used to export the path and update the already existing path with the path of the libraries of LLVM. These are the libraries that are going to be used to execute the pass that was initially created. Then, the name of the file needed to run the pass is requested. It is hypothesized that given file name is written in C++, hence the extension .cpp is planted in the code. At this point, using the compiler "clang++" the given program is compiled and a .bc file is created which is an LLVM bitcode file format. Following this, the command "opt-load" runs the already created pass from the LLVM using as input the output file that was created in the previous step and saves the results in a .txt file. It is important to store our data in a compact way, that is why a command from SQLlite3 is used to read the .txt file and use the data to create a table. The table has one column for the names of the functions and another for the basic blocks.

```
sqlite3 labrats/$name.db <<EOF
create table BBCount(funcName TEXT, count INT);
.separator " "
.import labrats/$name.txt BBCount
.header on
.mode column BBCount
select * from BBCount;
.exit
EOF
```

Figure 2: Bash Script Code – SQLite3 call instruction

Moreover, a viable keyword is created ("keyword =_shuffled") to be used in the renaming process of the output files and a counter is set to keep track of the desired number of attempts. Following the instructions provided in Github [18] for CCR, a command is run within the docker ("sudo docker run") that will in turn execute the CCR. It should be noted that within this command a path to a shared folder is included to allow our PC to communicate with the docker but also to provide CCR a path to store its results. This way, our command compiles the given program from the beginning along with the CCR and saves the executable within the "save" folder.

```
counter=1
sudo docker run  -v /home/anronis/Diplom/:/CCR/shareFolder --rm -it
kevinkoo001/ccr:0.8 /bin/bash -c "ccr++ -o ./shareFolder/labrats/$name ./
shareFolder/labrats/$name.cpp" > out
while [ $counter -le 10 ]
do
sudo docker run  -v /home/anronis/Diplom/:/CCR/shareFolder --rm -it
kevinkoo001/ccr:0.8 /bin/bash -c "python ./randomizer/prander.py -s -b ./
shareFolder/labrats/$name" > out
```

Figure 3: Bash Script Code – Compile with CCR and randomizer loop

However, the output obtained by the randomizer needs a place to store the name needed for the renaming process. That is why 2 variables are created to store the name and the renaming process can finally be executed, thus being able to move to the next step of creating a second counter. This counter will be used to track the number of the function that is being checked now and another variable is created to store the sum of the functions of the program. This number will later be used by executing an SQLite3 command that sends an SQL script to retrieve the specific number we need. Following this, the command "echo" is used to notify the user of the number of the current attempt in the program and then starts a second while-loop that will run for each function within the program.

```
counter2=1
ftotal=$(sqlite3 labrats/$name.db "select COUNT(0) from BBCount where 
count=1;");
echo "Shuffle " $counter ":"
    while [ $counter2 -le $ftotal ]
    do
    fname=$(sqlite3 labrats/$name.db "select funcName from (select (select 
COUNT(0) from BBCount t1 where t1.count=1 and t1.funcName <= t2.funcName) 
as RowNumber,funcName from BBCount t2 where t2.count=1 ORDER BY funcName) 
where RowNumber=$counter2;");
    objdump -d labrats/$name$keyword$counter | grep '<'$fname'>:'
    ((counter2++))
    done
```

Figure 4: Bash Script Code – Find specific functions with 1 BB

The desired functions within the loop are those that have only one basic block and the command "objdump" searches the function within the randomized executable in order to return its address. Using the GDB, which is the C++ debugger a randomized executable is run that finds the addresses of the tables of the programme with the purpose to examine them after the execution of the script to prove that there is no change. At this point, the loop ends. Finally, the process used in the second while-loop is repeated with the exception that the CCR executable is used. This executable, which is not randomized, is responsible for getting the addresses of the functions of the original file with the higher purpose of comparing them with those of the functions from the previous randomized executables.

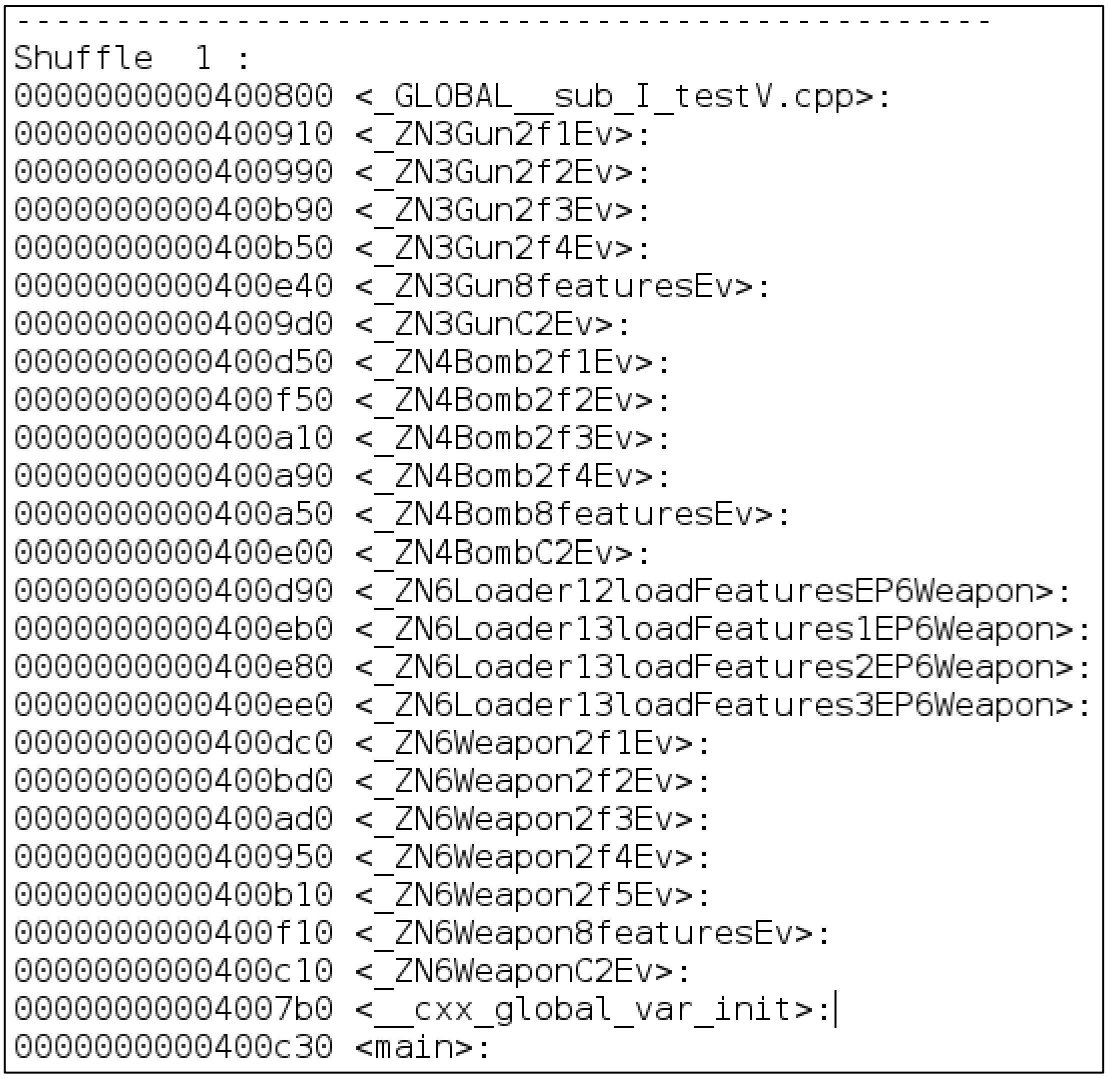
```
echo "-------------------VTables----------------------"
gdb ./labrats/$name$keyword$counter -ex 'info variable vtable' -ex 'x/10a 
0x0000000000401098' -ex 'q'
echo "-------------------------------------------------"
```

Figure 5: Bash Script Code – Find Vtables addresses

Chapter 6

Evaluation

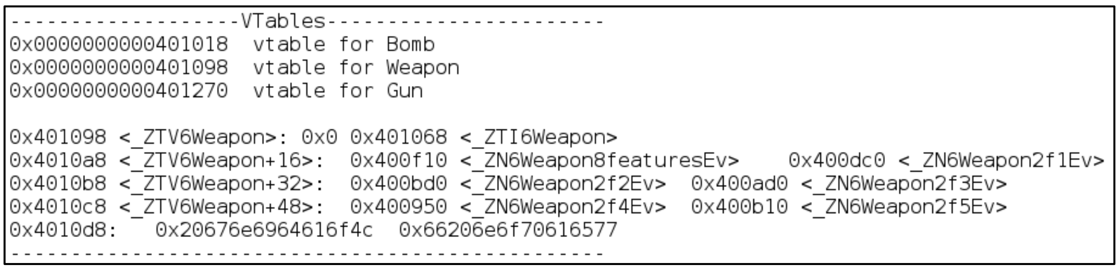
Initially, the script starts with requesting the name of the file. When you provide the name, it returns a table that includes the functions' names and the basic block counts. In the example below, each function has one basic block due to the fact that there is no code input inside. Furthermore, the functions that have one basic block each are presented in detail along with the address in hexadecimal format. The name of the function is also presented next to this output. In sum, this process is repeated 10 times and depending on the requested shuffles the address change is visible to the user. Finally, after the 10th shuffle the addresses from the original file that was not subjected to CCR's randomization are presented.



```
-------------------------------------------------
Shuffle  1 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400910 <_ZN3Gun2f1Ev>:
0000000000400990 <_ZN3Gun2f2Ev>:
0000000000400b90 <_ZN3Gun2f3Ev>:
0000000000400b50 <_ZN3Gun2f4Ev>:
0000000000400e40 <_ZN3Gun8featuresEv>:
00000000004009d0 <_ZN3GunC2Ev>:
0000000000400d50 <_ZN4Bomb2f1Ev>:
0000000000400f50 <_ZN4Bomb2f2Ev>:
0000000000400a10 <_ZN4Bomb2f3Ev>:
0000000000400a90 <_ZN4Bomb2f4Ev>:
0000000000400a50 <_ZN4Bomb8featuresEv>:
0000000000400e00 <_ZN4BombC2Ev>:
0000000000400d90 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400eb0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400e80 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400ee0 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400dc0 <_ZN6Weapon2f1Ev>:
0000000000400bd0 <_ZN6Weapon2f2Ev>:
0000000000400ad0 <_ZN6Weapon2f3Ev>:
0000000000400950 <_ZN6Weapon2f4Ev>:
0000000000400b10 <_ZN6Weapon2f5Ev>:
0000000000400f10 <_ZN6Weapon8featuresEv>:
0000000000400c10 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400c30 <main>:
```

Figure 6: Program's functions and their addresses

Under the information section of each function there is a virtual table (vtable) that presents the 3 vtables of the code, as well as its addresses. Below you can find the detailed output one of the three vtables that presents the addresses of the virtual functions that are included in the table. It should be noted that across all 10 shuffles all vtable addresses never change, hence we have a vulnerability in the system.



```
-------------------VTables-----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>: 0x0 0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:  0x400f10 <_ZN6Weapon8featuresEv>    0x400dc0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:  0x400bd0 <_ZN6Weapon2f2Ev>  0x400ad0 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:  0x400950 <_ZN6Weapon2f4Ev>  0x400b10 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c  0x66206e6f70616577
-------------------------------------------------
```

Figure 7: Program’s Vtables and their addresses

Chapter 7

Related Work

Following examples from other researchers we used disassembly to retrieve the addresses for the functions from the expendables. However, as it is argued that static disassembly is an “unsolved problem” [2] that is often misrepresented to seem overly pessimistic. In a study that evaluated nine “state-of-the-art” disassemblers on 981 real-world compiler-generated binaries the authors presented the issue of mismatch between literature and expectations [2]. Meaning that one should proceed with caution when evaluating vulnerabilities in the retrieval of addresses, just as it was done in this paper.

Additionally, it was aimed to identify the functions with one BB using the LLVM pass. From a reverse-engineering point of view, function identification is a common but rather important challenge in this process [6]. Research suggests that the initial assumption that functions can be identified by many binary programs can be misleading [6]. In line with previous research, in this paper function identification was challenging, however not as challenging as it would be if stripped binaries were used [6].

Binary code analysis is commonly adopted to support the process of information gathering about a program’s content and the way it is structured [15]. So, it can be described as an important pillar of many applications and underlies tools such as debugging [15]. Moreover, binary code analysis can be a complex process underlying the code randomization functions [6, 2] and the universality of this process as well as its accuracy are often doubted in research [15, 5, 8]. As researchers suggest, the type of information extracted by binary code analysis (e.g. BBs, functions, modules) are of vital importance in vulnerability testing and the quality and availability of the information may affect the outcome of our purpose [15].

In the mid-2000s, the popularity of the usage of buffer overflow attacks was peaking [4]. However, among the most commonly observed security vulnerabilities were unfortunate side effects of memory errors such as heap and integer overflows [4]. Moreover, even though there is a variety of defence mechanisms offered by the literature to deal with the above, they often come with shortcomings (e.g. high overheads) [4]. Just as Bhatkar S. [4] suggests, using code randomizations techniques such as the one his team of researchers tested (Address Space Randomization - ASR) can be employed to defend against these types of attacks. It should be noted though that such techniques are susceptible to information leakage and brute-force attacks.

Recent researches took interest in the implementations during load-time and compile-time [4, 1, 17] approaches. However, static binary rewriting of stripped binaries can be an option despite the challenges posed by its accuracy and universality by using methods such as dynamic binary instrumentation [16, 12, 21].

As described in this paper, a common way to deal with code reuse attacks is to "disguise" the content of fractions of the code using randomization. However, this still allows an attacker to scan a process and potentially read executable memory [3]. As a result, an attacker can assemble exploits on the desired target. To combat such issues, a security measure discussed in the literature is "Execute-no-Read (XnR) [3] which allows the execution of code but does not allow the code to be read as data [3]. This suggestion "damages" the self-disassembly that is crucial in the use of attacks like JIT-ROP [3]. It should be noted, however, that sometimes hardware support is a serious issue in the use of such defences [3].

It is suggested that the shift of focus on code diversification was the reason behind the interest in the use of ROP attacks [18] that eventually created the need of use of execute-only memory protections [3, 7]. The fine-grained randomization technique used in this thesis is presented as a necessity of the code diversification and execute-only approaches [18] and as it is suggested by the research on the topic, code pointers can still be retrieved from other data sections and be used to pinpoint the desired addresses [9, 13, 22].

However, recent practical techniques such as "Readactor" [10] are presented as resilient to both dynamic and static ROP attacks by giving attention to the direct and indirect memory disclosures that the attacker might use to put pointers on data pages or directly read code pages respectively [10].

Overall, there is a great deal of literature around known vulnerabilities in code randomization techniques, code reuse attacks and binary analysis and this paper aimed to draw from already existing work and evaluate already-tested hypotheses.

Chapter 8

Conclusion

The initial goal was to assess the reliability of CCR on defending against Control-flow attacks. Using an LLVM pass it was aimed to find the functions that have only one basic block in order to prove that CCR's randomization method can discourage any attacker from attempting to reusing those functions as gadgets. In line with previous research it was found that vtables remain fixed in their respective addresses. Meaning that despite the popularity of code randomization technique used in this thesis, the danger of overwriting the addresses inside the vtables still remains.

Bibliography

1. Anand, Kapil, Matthew Smithson, Khaled Elwazeer, Aparna Kotha, Jim Gruen, Nathan Giles, and Rajeev Barua. "A compiler-level intermediate representation based binary analysis and rewriting system." In Proceedings of the 8th ACM European Conference on Computer Systems, pp. 295-308. 2013.

2. Andriesse, Dennis, Xi Chen, Victor Van Der Veen, Asia Slowinska, and Herbert Bos. "An in-depth analysis of disassembly on full-scale x86/x64 binaries." In 25th {USENIX} Security Symposium ({USENIX} Security 16), pp. 583-600. 2016.

3. Backes, Michael, Thorsten Holz, Benjamin Kollenda, Philipp Koppe, Stefan Nürnberger, and Jannik Pewny. "You can run but you can't read: Preventing disclosure exploits in executable code." In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 1342-1353. 2014.

4. Bhatkar, Sandeep, Daniel C. DuVarney, and R. Sekar. "Efficient Techniques for Comprehensive Protection from Memory Error Exploits." In USENIX Security Symposium, pp. 17-17. 2005.

5. Balakrishnan, Gogul, Thomas Reps, David Melski, and Tim Teitelbaum. "Wysinwyx: What you see is not what you execute." In Working Conference on Verified Software: Theories, Tools, and Experiments, pp. 202-213. Springer, Berlin, Heidelberg, 2005.

6. Bao, Tiffany, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. "{BYTEWEIGHT}: Learning to Recognize Functions in Binary Code." In 23rd {USENIX} Security Symposium ({USENIX} Security 14), pp. 845-860. 2014.

7. Chen, Yaohui, Dongli Zhang, Ruowen Wang, Rui Qiao, Ahmed M. Azab, Long Lu, Hayawardh Vijayakumar, and Wenbo Shen. "NORAX: Enabling execute-only memory for COTS binaries on AArch64." In 2017 IEEE Symposium on Security and Privacy (SP), pp. 304-319. IEEE, 2017.

8. Cifuentes, Cristina, and Mike Van Emmerik. "Recovery of jump table case statements from binary code." Science of Computer Programming 40, no. 2-3 (2001): 171-188.

9. Conti, Mauro, Stephen Crane, Lucas Davi, Michael Franz, Per Larsen, Marco Negro, Christopher Liebchen, Mohaned Qunaibit, and Ahmad-Reza Sadeghi. "Losing control: On the effectiveness of control-flow integrity under stack attacks." In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 952-963. 2015.

10. Crane, Stephen, Christopher Liebchen, Andrei Homescu, Lucas Davi, Per Larsen, Ahmad-Reza Sadeghi, Stefan Brunthaler, and Michael Franz.

11. "Readactor: Practical code randomization resilient to memory disclosure." In 2015 IEEE Symposium on Security and Privacy, pp. 763-780. IEEE, 2015.

12. Crane, Stephen, Andrei Homescu, and Per Larsen. "Code randomization: Haven't we solved this problem yet?". In 2016 IEEE Cybersecurity Development (SecDev), pp. 124-129. IEEE, 2016.

13. Davi, Lucas Vincenzo, Alexandra Dmitrienko, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "Gadge me if you can: secure and efficient ad-hoc instruction-level randomization for x86 and ARM." In Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security, pp. 299-310. 2013.

14. Davi, Lucas, Christopher Liebchen, Ahmad-Reza Sadeghi, Kevin Z. Snow, and Fabian Monrose. "Isomeron: Code randomization resilient to (just-in-time) return-oriented programming." In NDSS. 2015.

15. Driesen, Karel, and Urs Hölzle. "The direct cost of virtual function calls in C++." In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 306-323. 1996.

16. Harris, Laune C., and Barton P. Miller. "Practical analysis of stripped binary code." ACM SIGARCH Computer Architecture News 33, no. 5 (2005): 63-68.

17. Hiser, Jason, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. "ILR: Where'd my gadgets go?". In 2012 IEEE Symposium on Security and Privacy, pp. 571-585. IEEE, 2012.

18. Homescu, Andrei, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. "Profile-guided automated software diversity." In Proceedings of the 2013 IEEE/ACM International Symposium on Code Generation and Optimization (CGO), pp. 1-11. IEEE, 2013.

19. Koo, Hyungjoon, Yaohui Chen, Long Lu, Vasileios P. Kemerlis, and Michalis Polychronakis. "Compiler-assisted code randomization." In 2018 IEEE Symposium on Security and Privacy (SP), pp. 461-477. IEEE, 2018.

20. Pappas, Vasilis, Michalis Polychronakis, and Angelos D. Keromytis. "Dynamic reconstruction of relocation information for stripped binaries." In International Workshop on Recent Advances in Intrusion Detection, pp. 68-87. Springer, Cham, 2014.

21. Shastry, Bhargava, Fabian Yamaguchi, Konrad Rieck, and Jean-Pierre Seifert. "Towards vulnerability discovery using staged program analysis." In

International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment, pp. 78-97. Springer, Cham, 2016.

22. Shioji, Eitaro, Yuhei Kawakoya, Makoto Iwamura, and Takeo Hariu. "Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks." In Proceedings of the 28th annual computer security applications conference, pp. 309-318. 2012.

23. Schuster, Felix, Thomas Tendyck, Christopher Liebchen, Lucas Davi, Ahmad-Reza Sadeghi, and Thorsten Holz. "Counterfeit object-oriented programming: On the difficulty of preventing code reuse attacks in C++ applications." In 2015 IEEE Symposium on Security and Privacy, pp. 745-762. IEEE, 2015.

24. Driesen, Karel, and Urs Hölzle. "The direct cost of virtual function calls in C++." In Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, pp. 306-323. 1996.

25. Xu, Jun, Pinyao Guo, Mingyi Zhao, Robert F. Erbacher, Minghui Zhu, and Peng Liu. "Comparing different moving target defense techniques." In Proceedings of the First ACM Workshop on Moving Target Defense, pp. 97-107. 2014.

26. Bounov, Dimitar, Rami Gökhan Kici, and Sorin Lerner. "Protecting C++ Dynamic Dispatch Through VTable Interleaving." In NDSS. 2016.

27. Cipresso, Teodoro, and Mark Stamp. "Software reverse engineering." In Handbook of Information and Communication Security, pp. 659-696. Springer, Berlin, Heidelberg, 2010.

Appendices

1. **Myscript**

```
#!/bin/bash

clear
export PATH=$PATH:/home/anronis/Diplom/build/bin
echo "HELLO!!!"
read -p "Please Enter name of doc: " name
clang++ -S -emit-llvm labrats/$name.cpp -c -o labrats/$name.bc
opt -load /home/anronis/Diplom/build/lib/BBCount.so -BBCount <
labrats/$name.bc > /dev/null > out 2> labrats/$name.txt
sqlite3 labrats/$name.db <<EOF
create table BBCount(funcName TEXT, count INT);
.separator " "
.import labrats/$name.txt BBCount
.header on
.mode column BBCount
select * from BBCount;
.exit
EOF
echo "-------------------------------------------------"
keyword=_shuffled
counter=1
sudo docker run  -v /home/anronis/Diplom/:/CCR/shareFolder --rm -it
kevinkoo001/ccr:0.8 /bin/bash -c "ccr++ -o ./shareFolder/labrats/$name
./shareFolder/labrats/$name.cpp" > out
while [ $counter -le 10 ]
do
sudo docker run  -v /home/anronis/Diplom/:/CCR/shareFolder --rm -it
kevinkoo001/ccr:0.8 /bin/bash -c "python ./randomizer/prander.py -s -b
./shareFolder/labrats/$name" > out
target1=$name$keyword
target2=$target1$counter
mv labrats/$target1 labrats/$target2
counter2=1
ftotal=$(sqlite3 labrats/$name.db "select COUNT(0) from BBCount where
count=1;");
echo "Shuffle " $counter ":"
    while [ $counter2 -le $ftotal ]
    do
    fname=$(sqlite3 labrats/$name.db "select funcName from (select
(select COUNT(0) from BBCount t1 where t1.count=1 and t1.funcName <=
t2.funcName) as RowNumber,funcName from BBCount t2 where
t2.count=1 ORDER BY funcName) where RowNumber=$counter2;");
    objdump -d labrats/$name$keyword$counter | grep '<'$fname'>:'
    ((counter2++))
    done
echo "-------------------VTables-----------------------"
gdb ./labrats/$name$keyword$counter -ex 'info variable vtable' -ex 'x/10a
0x0000000000401098' -ex 'q'
echo "------------------------------------------------"
((counter++))
done

counter2=1
ftotal=$(sqlite3 labrats/$name.db "select COUNT(0) from BBCount where
count=1;");
echo "Original:"
while [ $counter2 -le $ftotal ]
do
fname=$(sqlite3 labrats/$name.db "select funcName from (select (select
COUNT(0) from BBCount t1 where t1.count=1 and t1.funcName <=
t2.funcName) as RowNumber,funcName from BBCount t2 where
t2.count=1 ORDER BY funcName) where RowNumber=$counter2;");
objdump -d labrats/$name | grep '<'$fname'>:'
((counter2++))
done

sleep 2
```

2. OutputFile

```
funcName           count
--------------------  ----------
__cxx_global_var_init  1
main              1
_ZN4BombC2Ev         1
_ZN3GunC2Ev         1
_ZN6Loader12loadFeatu  1
_ZN6Loader13loadFeatu  1
_ZN6Loader13loadFeatu  1
_ZN6Loader13loadFeatu  1
_ZN6WeaponC2Ev       1
_ZN4Bomb8featuresEv    1
_ZN4Bomb2f1Ev        1
_ZN4Bomb2f2Ev        1
_ZN4Bomb2f3Ev        1
_ZN4Bomb2f4Ev        1
_ZN6Weapon2f5Ev       1
_ZN6Weapon8featuresEv  1
_ZN6Weapon2f1Ev       1
_ZN6Weapon2f2Ev       1
_ZN6Weapon2f3Ev       1
_ZN6Weapon2f4Ev       1
_ZN3Gun8featuresEv     1
_ZN3Gun2f1Ev         1
_ZN3Gun2f2Ev         1
_ZN3Gun2f3Ev         1
_ZN3Gun2f4Ev         1
_GLOBAL__sub_I_testV.  1
-------------------------------------------------
Shuffle  1 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400910 <_ZN3Gun2f1Ev>:
0000000000400990 <_ZN3Gun2f2Ev>:
0000000000400b90 <_ZN3Gun2f3Ev>:
0000000000400b50 <_ZN3Gun2f4Ev>:
0000000000400e40 <_ZN3Gun8featuresEv>:
00000000004009d0 <_ZN3GunC2Ev>:
0000000000400d50 <_ZN4Bomb2f1Ev>:
0000000000400f50 <_ZN4Bomb2f2Ev>:
0000000000400a10 <_ZN4Bomb2f3Ev>:
0000000000400a90 <_ZN4Bomb2f4Ev>:
0000000000400a50 <_ZN4Bomb8featuresEv>:
0000000000400e00 <_ZN4BombC2Ev>:
0000000000400d90 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400eb0 <_ZN6Loader13loadFeatures1EP6Weapon>:
```

```
0000000000400e80 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400ee0 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400dc0 <_ZN6Weapon2f1Ev>:
0000000000400bd0 <_ZN6Weapon2f2Ev>:
0000000000400ad0 <_ZN6Weapon2f3Ev>:
0000000000400950 <_ZN6Weapon2f4Ev>:
0000000000400b10 <_ZN6Weapon2f5Ev>:
0000000000400f10 <_ZN6Weapon8featuresEv>:
0000000000400c10 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400c30 <main>:
-------------------VTables-----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:        0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:     0x400f10      <_ZN6Weapon8featuresEv>
        0x400dc0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:     0x400bd0           <_ZN6Weapon2f2Ev>
        0x400ad0 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:     0x400950           <_ZN6Weapon2f4Ev>
        0x400b10 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
------------------------------------------------
Shuffle  2 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400f50 <_ZN3Gun2f1Ev>:
0000000000400950 <_ZN3Gun2f2Ev>:
0000000000400c30 <_ZN3Gun2f3Ev>:
0000000000400910 <_ZN3Gun2f4Ev>:
0000000000400bf0 <_ZN3Gun8featuresEv>:
0000000000400ea0 <_ZN3GunC2Ev>:
0000000000400d30 <_ZN4Bomb2f1Ev>:
0000000000400cf0 <_ZN4Bomb2f2Ev>:
0000000000400e60 <_ZN4Bomb2f3Ev>:
0000000000400d70 <_ZN4Bomb2f4Ev>:
0000000000400c70 <_ZN4Bomb8featuresEv>:
0000000000400f10 <_ZN4BombC2Ev>:
0000000000400aa0 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400e30 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400a10 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400ee0 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400db0 <_ZN6Weapon2f1Ev>:
0000000000400df0 <_ZN6Weapon2f2Ev>:
0000000000400a40 <_ZN6Weapon2f3Ev>:
00000000004009d0 <_ZN6Weapon2f4Ev>:
```

```
0000000000400990 <_ZN6Weapon2f5Ev>:
0000000000400cb0 <_ZN6Weapon8featuresEv>:
0000000000400a80 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400ad0 <main>:
-------------------VTables----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:        0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:    0x400cb0     <_ZN6Weapon8featuresEv>
        0x400db0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:    0x400df0             <_ZN6Weapon2f2Ev>
        0x400a40 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:    0x4009d0             <_ZN6Weapon2f4Ev>
        0x400990 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
------------------------------------------------
Shuffle  3 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400990 <_ZN3Gun2f1Ev>:
0000000000400d00 <_ZN3Gun2f2Ev>:
0000000000400ee0 <_ZN3Gun2f3Ev>:
0000000000400c50 <_ZN3Gun2f4Ev>:
0000000000400a50 <_ZN3Gun8featuresEv>:
0000000000400f20 <_ZN3GunC2Ev>:
0000000000400910 <_ZN4Bomb2f1Ev>:
0000000000400c10 <_ZN4Bomb2f2Ev>:
0000000000400ab0 <_ZN4Bomb2f3Ev>:
00000000004009d0 <_ZN4Bomb2f4Ev>:
0000000000400c90 <_ZN4Bomb8featuresEv>:
0000000000400b30 <_ZN4BombC2Ev>:
0000000000400cd0 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400b70 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400be0 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400f60 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400950 <_ZN6Weapon2f1Ev>:
0000000000400ba0 <_ZN6Weapon2f2Ev>:
0000000000400d40 <_ZN6Weapon2f3Ev>:
0000000000400af0 <_ZN6Weapon2f4Ev>:
0000000000400ea0 <_ZN6Weapon2f5Ev>:
0000000000400a10 <_ZN6Weapon8featuresEv>:
0000000000400a90 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400d80 <main>:
-------------------VTables----------------------
```

```
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:          0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400a10     <_ZN6Weapon8featuresEv>
        0x400950 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400ba0            <_ZN6Weapon2f2Ev>
        0x400d40 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400af0            <_ZN6Weapon2f4Ev>
        0x400ea0 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
------------------------------------------------
Shuffle  4 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400b50 <_ZN3Gun2f1Ev>:
0000000000400a60 <_ZN3Gun2f2Ev>:
0000000000400e60 <_ZN3Gun2f3Ev>:
0000000000400c10 <_ZN3Gun2f4Ev>:
0000000000400ed0 <_ZN3Gun8featuresEv>:
0000000000400db0 <_ZN3GunC2Ev>:
0000000000400f50 <_ZN4Bomb2f1Ev>:
0000000000400f10 <_ZN4Bomb2f2Ev>:
0000000000400c50 <_ZN4Bomb2f3Ev>:
0000000000400e20 <_ZN4Bomb2f4Ev>:
0000000000400b10 <_ZN4Bomb8featuresEv>:
0000000000400b90 <_ZN4BombC2Ev>:
0000000000400ea0 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400df0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400ae0 <_ZN6Loader13loadFeatures2EP6Weapon>:
00000000004009b0 <_ZN6Loader13loadFeatures3EP6Weapon>:
00000000004009e0 <_ZN6Weapon2f1Ev>:
0000000000400970 <_ZN6Weapon2f2Ev>:
0000000000400910 <_ZN6Weapon2f3Ev>:
0000000000400aa0 <_ZN6Weapon2f4Ev>:
0000000000400a20 <_ZN6Weapon2f5Ev>:
0000000000400bd0 <_ZN6Weapon8featuresEv>:
0000000000400950 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400c90 <main>:
-------------------VTables----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:          0x0      0x401068 <_ZTI6Weapon>
```

```
0x4010a8 <_ZTV6Weapon+16>:      0x400bd0     <_ZN6Weapon8featuresEv>
        0x4009e0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400970             <_ZN6Weapon2f2Ev>
        0x400910 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400aa0             <_ZN6Weapon2f4Ev>
        0x400a20 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
------------------------------------------------
Shuffle  5 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400bf0 <_ZN3Gun2f1Ev>:
0000000000400ce0 <_ZN3Gun2f2Ev>:
0000000000400d20 <_ZN3Gun2f3Ev>:
00000000004009c0 <_ZN3Gun2f4Ev>:
0000000000400c30 <_ZN3Gun8featuresEv>:
0000000000400a40 <_ZN3GunC2Ev>:
0000000000400b70 <_ZN4Bomb2f1Ev>:
0000000000400910 <_ZN4Bomb2f2Ev>:
0000000000400af0 <_ZN4Bomb2f3Ev>:
0000000000400f50 <_ZN4Bomb2f4Ev>:
0000000000400d90 <_ZN4Bomb8featuresEv>:
0000000000400b30 <_ZN4BombC2Ev>:
0000000000400cb0 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400ac0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400d60 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400950 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400f10 <_ZN6Weapon2f1Ev>:
0000000000400980 <_ZN6Weapon2f2Ev>:
0000000000400a00 <_ZN6Weapon2f3Ev>:
0000000000400bb0 <_ZN6Weapon2f4Ev>:
0000000000400a80 <_ZN6Weapon2f5Ev>:
0000000000400c70 <_ZN6Weapon8featuresEv>:
0000000000400ef0 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400dd0 <main>:
-------------------VTables-----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:         0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400c70     <_ZN6Weapon8featuresEv>
        0x400f10 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400980             <_ZN6Weapon2f2Ev>
        0x400a00 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400bb0             <_ZN6Weapon2f4Ev>
        0x400a80 <_ZN6Weapon2f5Ev>
```

```
0x4010d8:     0x20676e6964616f4c 0x66206e6f70616577
-------------------------------------------------
Shuffle  6 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400f10 <_ZN3Gun2f1Ev>:
0000000000400910 <_ZN3Gun2f2Ev>:
00000000004009d0 <_ZN3Gun2f3Ev>:
0000000000400a90 <_ZN3Gun2f4Ev>:
0000000000400a50 <_ZN3Gun8featuresEv>:
0000000000400e20 <_ZN3GunC2Ev>:
0000000000400e60 <_ZN4Bomb2f1Ev>:
0000000000400d80 <_ZN4Bomb2f2Ev>:
0000000000400950 <_ZN4Bomb2f3Ev>:
0000000000400a10 <_ZN4Bomb2f4Ev>:
0000000000400990 <_ZN4Bomb8featuresEv>:
0000000000400d10 <_ZN4BombC2Ev>:
0000000000400d50 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400ce0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400ea0 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400ad0 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400ca0 <_ZN6Weapon2f1Ev>:
0000000000400b00 <_ZN6Weapon2f2Ev>:
0000000000400c60 <_ZN6Weapon2f3Ev>:
0000000000400f50 <_ZN6Weapon2f4Ev>:
0000000000400dc0 <_ZN6Weapon2f5Ev>:
0000000000400ed0 <_ZN6Weapon8featuresEv>:
0000000000400e00 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400b40 <main>:
-------------------VTables-----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:          0x0       0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400ed0     <_ZN6Weapon8featuresEv>
        0x400ca0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400b00              <_ZN6Weapon2f2Ev>
        0x400c60 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400f50              <_ZN6Weapon2f4Ev>
        0x400dc0 <_ZN6Weapon2f5Ev>
0x4010d8:     0x20676e6964616f4c 0x66206e6f70616577
-------------------------------------------------
Shuffle  7 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
00000000004009d0 <_ZN3Gun2f1Ev>:
0000000000400a90 <_ZN3Gun2f2Ev>:
```

```
0000000000400e40 <_ZN3Gun2f3Ev>:
0000000000400ed0 <_ZN3Gun2f4Ev>:
0000000000400d50 <_ZN3Gun8featuresEv>:
0000000000400b00 <_ZN3GunC2Ev>:
0000000000400950 <_ZN4Bomb2f1Ev>:
0000000000400f10 <_ZN4Bomb2f2Ev>:
0000000000400dc0 <_ZN4Bomb2f3Ev>:
0000000000400a10 <_ZN4Bomb2f4Ev>:
0000000000400d10 <_ZN4Bomb8featuresEv>:
0000000000400cd0 <_ZN4BombC2Ev>:
0000000000400e80 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400d90 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400ad0 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400c60 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400990 <_ZN6Weapon2f1Ev>:
0000000000400a50 <_ZN6Weapon2f2Ev>:
0000000000400910 <_ZN6Weapon2f3Ev>:
0000000000400c90 <_ZN6Weapon2f4Ev>:
0000000000400f50 <_ZN6Weapon2f5Ev>:
0000000000400e00 <_ZN6Weapon8featuresEv>:
0000000000400eb0 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400b40 <main>:
-------------------VTables-----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:         0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400e00     <_ZN6Weapon8featuresEv>
        0x400990 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400a50            <_ZN6Weapon2f2Ev>
        0x400910 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400c90            <_ZN6Weapon2f4Ev>
        0x400f50 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
--------------------------------------------------
Shuffle  8 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400c10 <_ZN3Gun2f1Ev>:
0000000000400bd0 <_ZN3Gun2f2Ev>:
0000000000400ed0 <_ZN3Gun2f3Ev>:
0000000000400b90 <_ZN3Gun2f4Ev>:
0000000000400a40 <_ZN3Gun8featuresEv>:
0000000000400e10 <_ZN3GunC2Ev>:
0000000000400da0 <_ZN4Bomb2f1Ev>:
0000000000400a00 <_ZN4Bomb2f2Ev>:
```

```
0000000000400aa0 <_ZN4Bomb2f3Ev>:
0000000000400f50 <_ZN4Bomb2f4Ev>:
0000000000400e90 <_ZN4Bomb8featuresEv>:
0000000000400980 <_ZN4BombC2Ev>:
0000000000400910 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400ae0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400c50 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400de0 <_ZN6Loader13loadFeatures3EP6Weapon>:
00000000004009c0 <_ZN6Weapon2f1Ev>:
0000000000400b10 <_ZN6Weapon2f2Ev>:
0000000000400940 <_ZN6Weapon2f3Ev>:
0000000000400f10 <_ZN6Weapon2f4Ev>:
0000000000400e50 <_ZN6Weapon2f5Ev>:
0000000000400b50 <_ZN6Weapon8featuresEv>:
0000000000400a80 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400c80 <main>:
-------------------VTables-----------------------

0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:         0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400b50     <_ZN6Weapon8featuresEv>
        0x4009c0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400b10              <_ZN6Weapon2f2Ev>
        0x400940 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400f10              <_ZN6Weapon2f4Ev>
        0x400e50 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
-------------------------------------------------
Shuffle  9 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400bd0 <_ZN3Gun2f1Ev>:
0000000000400a50 <_ZN3Gun2f2Ev>:
0000000000400ed0 <_ZN3Gun2f3Ev>:
0000000000400cf0 <_ZN3Gun2f4Ev>:
0000000000400910 <_ZN3Gun8featuresEv>:
00000000004009a0 <_ZN3GunC2Ev>:
0000000000400f50 <_ZN4Bomb2f1Ev>:
0000000000400b50 <_ZN4Bomb2f2Ev>:
0000000000400b90 <_ZN4Bomb2f3Ev>:
0000000000400f10 <_ZN4Bomb2f4Ev>:
0000000000400d70 <_ZN4Bomb8featuresEv>:
0000000000400b10 <_ZN4BombC2Ev>:
0000000000400c10 <_ZN6Loader12loadFeaturesEP6Weapon>:
```

```
0000000000400cc0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400a20 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400950 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400ad0 <_ZN6Weapon2f1Ev>:
0000000000400d30 <_ZN6Weapon2f2Ev>:
00000000004009e0 <_ZN6Weapon2f3Ev>:
0000000000400a90 <_ZN6Weapon2f4Ev>:
0000000000400c40 <_ZN6Weapon2f5Ev>:
0000000000400c80 <_ZN6Weapon8featuresEv>:
0000000000400980 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400db0 <main>:
-------------------VTables-----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:         0x0      0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400c80     <_ZN6Weapon8featuresEv>
        0x400ad0 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400d30              <_ZN6Weapon2f2Ev>
        0x4009e0 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400a90              <_ZN6Weapon2f4Ev>
        0x400c40 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
------------------------------------------------
Shuffle  10 :
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400be0 <_ZN3Gun2f1Ev>:
0000000000400de0 <_ZN3Gun2f2Ev>:
00000000004009c0 <_ZN3Gun2f3Ev>:
0000000000400910 <_ZN3Gun2f4Ev>:
0000000000400c60 <_ZN3Gun8featuresEv>:
0000000000400ca0 <_ZN3GunC2Ev>:
0000000000400c20 <_ZN4Bomb2f1Ev>:
0000000000400f10 <_ZN4Bomb2f2Ev>:
0000000000400e20 <_ZN4Bomb2f3Ev>:
0000000000400d10 <_ZN4Bomb2f4Ev>:
0000000000400980 <_ZN4Bomb8featuresEv>:
0000000000400ed0 <_ZN4BombC2Ev>:
0000000000400ce0 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400ea0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400950 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400db0 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400e60 <_ZN6Weapon2f1Ev>:
0000000000400a40 <_ZN6Weapon2f2Ev>:
0000000000400d70 <_ZN6Weapon2f3Ev>:
```

```
0000000000400ba0 <_ZN6Weapon2f4Ev>:
0000000000400a00 <_ZN6Weapon2f5Ev>:
0000000000400f50 <_ZN6Weapon8featuresEv>:
0000000000400d50 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400a80 <main>:
------------------VTables----------------------
0x0000000000401018  vtable for Bomb
0x0000000000401098  vtable for Weapon
0x0000000000401270  vtable for Gun

0x401098 <_ZTV6Weapon>:         0x0     0x401068 <_ZTI6Weapon>
0x4010a8 <_ZTV6Weapon+16>:      0x400f50      <_ZN6Weapon8featuresEv>
        0x400e60 <_ZN6Weapon2f1Ev>
0x4010b8 <_ZTV6Weapon+32>:      0x400a40             <_ZN6Weapon2f2Ev>
        0x400d70 <_ZN6Weapon2f3Ev>
0x4010c8 <_ZTV6Weapon+48>:      0x400ba0             <_ZN6Weapon2f4Ev>
        0x400a00 <_ZN6Weapon2f5Ev>
0x4010d8:    0x20676e6964616f4c 0x66206e6f70616577
----------------------------------------------
Original:
0000000000400800 <_GLOBAL__sub_I_testV.cpp>:
0000000000400e90 <_ZN3Gun2f1Ev>:
0000000000400ed0 <_ZN3Gun2f2Ev>:
0000000000400f10 <_ZN3Gun2f3Ev>:
0000000000400f50 <_ZN3Gun2f4Ev>:
0000000000400e50 <_ZN3Gun8featuresEv>:
0000000000400a70 <_ZN3GunC2Ev>:
0000000000400bd0 <_ZN4Bomb2f1Ev>:
0000000000400c10 <_ZN4Bomb2f2Ev>:
0000000000400c50 <_ZN4Bomb2f3Ev>:
0000000000400c90 <_ZN4Bomb2f4Ev>:
0000000000400b90 <_ZN4Bomb8featuresEv>:
0000000000400a30 <_ZN4BombC2Ev>:
0000000000400ab0 <_ZN6Loader12loadFeaturesEP6Weapon>:
0000000000400ae0 <_ZN6Loader13loadFeatures1EP6Weapon>:
0000000000400b10 <_ZN6Loader13loadFeatures2EP6Weapon>:
0000000000400b40 <_ZN6Loader13loadFeatures3EP6Weapon>:
0000000000400d50 <_ZN6Weapon2f1Ev>:
0000000000400d90 <_ZN6Weapon2f2Ev>:
0000000000400dd0 <_ZN6Weapon2f3Ev>:
0000000000400e10 <_ZN6Weapon2f4Ev>:
0000000000400cd0 <_ZN6Weapon2f5Ev>:
0000000000400d10 <_ZN6Weapon8featuresEv>:
0000000000400b70 <_ZN6WeaponC2Ev>:
00000000004007b0 <__cxx_global_var_init>:
0000000000400910 <main>:
```