May 2021

Individual Diploma Thesis

LARGE-SCALE AUTOMATED TYPE CONFUSION VULNERABILITY ADDITION

Nikolaos Georgiou

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2021

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

LARGE-SCALE AUTOMATED TYPE CONFUSION VULNERABILITY ADDITION

Nikolaos Georgiou

Supervisor

Elias Athanasopoulos

The Individual Diploma Thesis was submitted for partial fulfillment of the requirements for obtaining the degree of Computer Science of the Department of Computer Science of the University of Cyprus

Thanks

I wish to thank my supervisor Elias Athanasopoulos, for his valuable help and guidance throughout the fulfillment of my thesis and for the support he gave me in difficult times where I thought I was at a dead end.

Summary

In this thesis, we would analyze a way we can automatically inject a large amount of type confusion bugs inside programs written in C++. We will describe what a type confusion bug is and how an attacker can use it to harm the normal execution of a program. Also, we will see why it is useful nowadays to have tools that can inject a big number of vulnerabilities inside programs and why are those tools useful for the evaluation of bug-finding tools. Tools, like the one we propose here can be used to inject cheaply and usually fast a large number of bugs inside programs and then use the executables we produce to test how good are some bug-finding tools in finding vulnerabilities inside programs and help them become better.

In this thesis, we will describe the whole process we follow to inject our bugs inside programs, from the time we get the source files of a program, until the time we generate an executable of the program, that has the same functionality as the original program, but with the right input files we can trigger type confusion bugs we put inside the program and force it to crash. Also, we will mention difficulties we faced during our research and what methodology we used to solve them. Finally, we will use some test programs and a bug-finding tool to evaluate our implementation and make our conclusions.

Contents

Chapter 1

Introduction

1.1 **Problem description**

Ever since computers were created, the main problem for developers has been finding non obvious bugs in their programs that can be used by attackers to either harm the normal execution of their program, either steal information or money from companies or ordinary users and many more criminal activities. For this reason, various bug-finding tools have been proposed, studied and implemented and their main aim is to locate pieces within the program code, that are likely to be used by attackers for malicious use. Those tools use various techniques (like fuzzing) to detect various types of non obvious bugs, such as buffer overflow, use-after-free, and more. The main problem with these tools is their evaluation, specifically how well they are at locating pieces of code that can actually be used for malicious purposes. This task is not so easy because finding a good sample of code with real bugs and in a big quantity is very difficult and usually bug-finding-tools have been trained to locate known bugs that can be found in real world source codes. On the other hand, manually creating programs with bugs is a very difficult and lengthy process and it can give a relatively small sample of code with which we can evaluate a bug-finding-tool.

1.2 **Related work**

Various researches have been done in order to solve the above problem and they attended to automatically inject a large number of bugs inside programs by changing either the source code of the program either its bytecode. One of those is LAVA: Large-scale Automated Vulnerability Addition [1], where it can be used to inject automatically a large number of buffer overflow bugs and those bugs can be triggered only by a specific input. Another

research, tried to automatically inject inside programs a large number of data race bugs [2]. Both researches were trying not only to inject a big number of bugs inside real life programs, but also embed those bugs inside the program in such manner so it can be difficult for bug-finding-tools to find them.

1.3 **Our scope**

Our work is consecrated in automatically injecting a large number of type confusion bugs inside programs written in C/C++ and those bugs will be triggered by specific input that will be given when the program is executed. In the beginning we studied type confusion bugs in both C and C++, but after we concentrated only in C++, because it is more common to find those types of bugs in C++ programs, where we have the sense of classes and inheritance. We assume that we are provided with the source code of a program and from this source code we use LLVM to inject our bugs in the Intermediate Representation of this source code. Before we inject the bugs inside the program we first try to dynamically identify the values that each variable of the program has, when it gets a specific input, so we can choose among them the variables which values change between different inputs, that are given to the program and use those values to trigger our bugs. After we collect all this information, we inject inside the program a Type Confusion bug in every place that meets all the conditions that we will analyze later in this thesis. In order to test our implementation, we injected our bugs inside some test programs and after we gave the executables we produced to a fuzzer, which tried to identify the bugs we inserted. After letting the fuzzer to run for days, we saw that it found only a small portion of the bugs we inserted, showing us that our bugs are both traceable from fuzzers and hard to find. As a result, in the future our implementation can be used in order to inject automatically a large number of type confusion bugs inside programs and use them to evaluate the effectiveness of bug-finding tools in tracing type confusion bugs.

Chapter 2

Background

2.1 Type confusion bugs

A type confusion vulnerability as its name indicates is a logical bug, which results from a confusion between object types.
First of all, we will analyse the different casting options that C++ offers us:

1) Static casting:
   In static casting the validity of the casting is checked in compile time without having any run-time information.
   Example: ToClass pointer_var = static_cast<ToClass>(Object)

2) Dynamic casting:
   In dynamic casting, the validity of the casting is checked in run-time based on the vtable pointer of the object to be casted. It is more safe from static casting, but it requires some more checks during execution time and therefore is not used in applications that want to achieve better time performance.
   Example: ToClass pointer_var = dynamic_cast<ToClass>(Object)

Type confusion bugs arise when we try to do an illegal non-safe downcasting from a parent class or struct to a child class or struct (here as non-safe we refer to a static cast) and as a result the pointer to an object that we have may behave in a non predictable manner. In order for a type confusion exploitation to be successful usually an attacker needs to have control over two pointers of different types that point to the same memory area. By this the attacker can interpretate the fields of the object that is saved in a single memory area with two different ways and as a result have the power to manipulate the program in the way that best fits his intentions. We will refer in this thesis to casts between classes and/or structs, because

in C++ classes and structs are almost the same thing, their only difference is that structs by default have their members and bases as public, in contrast classes have their members and bases by default as private. In order to better explain type confusion bugs we provide the example bellow (example retrieved from https://bufferoverflows.net/type-confusion-vulnerabilities):

```
class Parent {};

class Child1: public Parent {
public:
   virtual void execute(const char *command) {
      system(command);
   }
};

class Child2: public Parent {
public:
   virtual void sayHello(const char *str) {
      cout << str << endl;
   }
};
int main() {
   Parent* p_child1 = new Child1();
   Parent* p_child2 = new Child2();
   Child2* temp;
(1)   temp = static_cast< Child2*>( p_child2); // Safe Casting to the same type "Child2"
   temp ->sayHello("Hello World"); // String passed to sayHello() function

(2)   temp = static_cast< Child2*>( p_child1); // Unsafe Downcasting to class "Child"
   temp ->sayHello("/usr/bin/xcalc"); // String passed to exec() function
                                      // which will turn into a command to execute calculator
   delete p_child2;
   delete p_child1;
   return 0;
}
```

Memory area of each object

p_child1

vtable*
void execute(char *)

p_child2

vtable*
void sayHello(char *)

Example 1: A simple program that demonstrates how a type confusion bug is generated.

In the example above we have three classes, the first is the parent class named “Parent” and from this class inherit the two other classes “Child1” and “Child2”. Both classes “Child1” and “Child2”, have one function that is void and takes as a parameter a string, in the class “Child1” the function executes in the command line the string that is passed and in “Child2” it prints in the terminal the string. In our “main” function, we create an Object of type “Child1” and we upcast it to a “Parent” pointer and we create an Object of type “Child2” that we also upcast to a “Parent” pointer. After, we create a new pointer to a class of type “Child2” named “temp”, in our first attempt (1)* we make “temp” to point to a memory area where we have an Object of the same type and this is done through a down casting of a pointer to a “Parent” object that points to an Object of type “Child2” and as a result the down casting is safe and when we call the function sayHello(…) it prints the message in the terminal as expected. On the other hand in our second attempt (2)*, we cast a pointer of “Parent” Object that holds an Object of “Child1”, to a pointer of “Child2” and as a result we make an unsafe down casting. So after making the unsafe down casting, we believe that the pointer “temp” points to an object of type “Child2”, but it points to an Object of type “Child1” and as a result when we call the function sayHello(…) from pointer “temp”, it does not call sayHello(…), but it calls execute(…) which executes the calculator of the system. Same exploitation could happen if we used variables inside our classes.

Type confusion vulnerabilities are commonly found as the cause of various attacks and in contrast to well-known bugs, like buffer overflow, they are not so wildly known by programmers and therefore programmers do not pay so much attention in avoiding introducing such bugs inside their programs.

2.2 LLVM

The LLVM project is a collection of modular and reusable compiler and toolchain technologies, which can be used to develop a front end for any programming language and a back end for any instruction set architecture, as it is described in its official site[3]. For the

needs of this document we will describe only a small subset of the functionalities offered by LLVM and specifically we will describe how LLVM allows us to retrieve a language-independent intermediate representation (IR) of the source code of the program and process or analyse it through many passes. First of all, IR (Intermediate Representation) is a language-independent representation that serves as a portable, high-level assembly language, that we can use in order to process the code that is generated in high level in any programming language and after from this generate the assembly that will be used by the machine to execute the program. The IR does not uses registers but rather is has an infinite set of temporaries of the form %0, %1, etc and it provides some more high level information about the variables and the operations we do to them, rather what we get in assembly code. One type of instructions, being used in LLVM IR, that we must mention and is very important for this thesis are the Bitcast instructions. A bitcast instruction is used in IR language to cast a variable from one type to another without changing any bits, the type can be either a primitive type (e.g. integer, double, float, etc) or a struct or a class or a union. An example of a "Hello World" in LLVM IR is:

```
@.str = private unnamed_addr constant [13 x i8] c"Hello World\0A\00", align 1

; Function Attrs: noinline norecurse optnone uwtable
define dso_local i32 @main(i32 %0, i8** %1) #4 {
  %3 = alloca i32, align 4
  %4 = alloca i32, align 4
  %5 = alloca i8**, align 8
  store i32 0, i32* %3, align 4
  store i32 %0, i32* %4, align 4
  store i8** %1, i8*** %5, align 8
  %6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @.str, i64
0, i64 0))
  ret i32 0
}

declare dso_local i32 @printf(i8*, ...) #1
```

<u>Example 2:</u> A simple Hello World example in LLVM IR. Here we provide only the main function that contains only a call to "printf" that prints to the terminal the string "Hello World". In the beginning of the function we allocate some space in memory to save the

values that we take as parameters (e.g. %3 = alloca i32, align 4 *) and we will refer after to them by the unique register number they have ("*%3*", "*%4*" and "*%5*"). Then, we store the values of the parameters of the function in the areas we allocated and we call the "printf" function with a constant string that is declared in this module.*

LLVM offers the ability to write some "Passes", which are programs written in C++, that are executed on the IR of a program and make some optimizations on it. For our purposes our passes will be only transformation passes, this means that they will change, add or remove instructions in the intermediate representation (IR) of the program we want to insert the type confusion bugs. In LLVM we have modules, functions, blocks and instructions. A module represents a single unit of code, that is to be processed together, it contains things like global variables, function declarations and implementations. Functions in LLVM have the same meaning as in the high level language, they contain blocks of instructions that each function must execute in order to implement the functionality that is described in the high level declaration of this function. Each function contains one or more blocks of LLVM instructions, a block is simply a container of instructions that execute sequentially. Finally we have LLVM instructions, LLVM provides a variety of instructions and each instruction has its own functionality, instructions can allocate memory, store in memory, make arithmetic computations and many more. Some instructions that we will mention later and have an important role in the implementation of the system are:

Alloca instructions: Allocate space in memory for a variable

%3 = alloca i32, align 4

Store instructions: Store a constant value or a value from one register to the destination register

store i32 0, i32* %3, align 4

Call instructions: They call functions declared inside the module and pass the parameters they need

%6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([13 x i8], [13 x i8]* @.str, i64 0, i64 0))

Bitcast instructions: Used for casting between any type of variables (primitive and object types)

%5 = bitcast %class.User* %4 to %class.Login*

Chapter 3

Architecture

3.1 Overview

At a high level, we add type confusion bugs to programs in the following manner. Given the source code of a program and some sample input files for that program we:

1. Compile the program and during the compilation we apply a transformation pass, which will help us dynamically trace information about the bitcast instructions and the available variables before them.
2. Take the executable that is generate from the previous step and run the program at least twice for each input file that we have. Then we retrieve the information we need in order to find for each possible type confusion bug, which variables can be used in order to trigger it.
3. With the information we get from the previous step, we run a program that tries to identify for each bitcast instruction, which variables that are initialized before this bitcast instruction take their values from the input files that the program reads.
4. Finally, we recompile the program, but this time we apply a different transformation pass, which will inject the type confusion bugs inside the program.

We will discuss each step in the following four sections.

3.2 Retrieve info for the available variables before each bitcast instruction

The most difficult part of our work was to determine how we will inject our type confusion bugs inside the target program in such way so it could be difficult for a bug-finding tool to

find them. In order to do this we wanted our type confusion bugs to be triggered only when some variables that are already inside the original program take a specific value, that is changed between different inputs that are given when we execute the program. In other words, those variables take their values during execution time from the files that the program reads and those values must change from file to file. This is done, because if we do not insert a condition for triggering the bugs or the variable values that trigger the bugs does not change for different input files, then the bugs will be triggered every time we run the program and they will be easily identified by the bug-finding tools (we do not want that). For instance, if before a type confusion bug the program assigns to a certain variable the second value that is read from an input file and this value changes between different input files, ideally we will want to inject a type confusion bug for each value that this variable may get. In order to do this we tried different approaches. First, we tried to use the same mechanism that was used in the LAVA paper [1] and find DUAs (Dead, Uncomplicated and Available data) that we can use, but we did not have much success on this, since LAVA was implemented for C source files and we are working with C++ and it was using some mechanisms that we could not understand or we could not use since they were incompatible with our problem. Also, we tried to use PANDA to do a dynamic taint analysis over the programs we want to inject the bugs, but this also did not give us any useful results.

After all, what we did in order to find the variables that we need and their according values, is by injecting inside the program multiple calls to "printf" function, which will print in the terminal the values of each variable that have been assigned with a value before the execution of each bitcast instruction. Specifically to this, we take the source files of the program and we compile them using Clang and an LLVM transformation pass, which firstly goes and finds all the Integer, Double, Float, Char and Boolean variables that the program initializes before each bitcast instruction. After the pass goes and inserts multiple calls to "printf" before each bitcast instruction that meets some certain specifications (we will analyze those specifications in more depth in the implementation chapter), which will print in the terminal the value that each of those variable has before the bitcast instruction is executed, along with some information that will let us identify which variable is used in the printing. When the above compilation of the program finishes we get an executable of the program with the exact functionality of the original program, but with some additional prints to the terminal. As a result, when we will run the produced executable with any given input file, the program will run normally, but during its execution it will print on the terminal information about all the bitcast instructions it executes and with the bitcast instructions it will print the values of the integer, boolean, char, float and double variables that have been assigned with any value

and are declared before that bitcast instruction (the variables we print do not have any other relation with a bitcast instruction, rather than they are in the same function with the bitcast instruction and they are declared and assigned with a value before the bitcast instruction is executed)

3.3 Retrieve the values of selected variables for different input files

When we will compile the source files of the target program with the pass that is described in the previous section, we will then take each unique input file of the program and we will execute the program with each file at least twice. With each run of the program, we will collect information about the bitcast instructions that are executed during the execution of the program and we will retrieve the values of the available variables before each bitcast instruction. The reason that we will execute the program for each input file at least twice, is because in the next step that we will process the output of the program with each file, we will want to know what are the values of each variable, that we print in the terminal, if we run the program multiple times with the same input (this is mainly for identifying variables that may get random values). In our experiments we decided that running the program 2 times for each unique input file is good enough (but as we said we could do it more than twice) and that is because it is very rare for variables that take random values to have the same value in two different runs of the program with the same input file, but if we want to make this probability even smaller we can run the program more than 2 times for each unique input file, but this will add an additional time overhead in the whole process. Therefore, in our experiments we run the program with each unique input file twice and for every run, we save the messages that the program outputs in the terminal inside text files (different text file for each execution of the program). For example, if we have 2 unique input files to execute the program, then for the first input file we will produce two text files with the output of the program when it is executed with this input file (each file will be a separate run of the program) and we will produce another two text files for the second input file.

3.4 Select variable-value pairs for the conditional triggering of the bugs

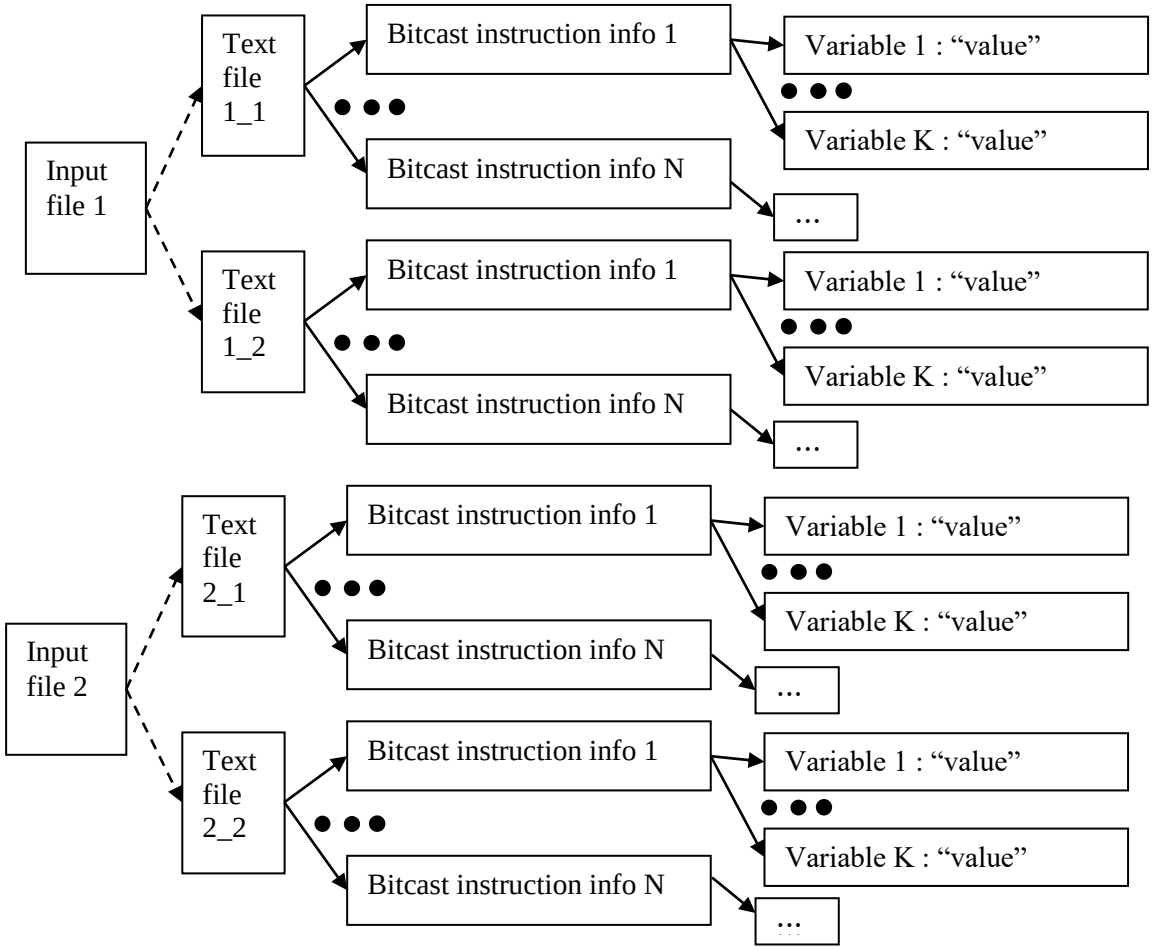
In this step, we collect all the text files that are produced from the previous step and we run a program in C++, which process the text that is saved inside them, in order to determine which of those variables take their values from the input files that we used to run the program. After we process the text files, we save the information they contain as it is shown in the figure 1. For each file, we save the information for each bitcast instruction that is contained

inside the program and for each bitcast instruction we save information about the variables that are available before that bitcast instruction and for each variable we save the value that this variable has, just before the bitcast instruction is executed. Also, from the text files we retrieve information about which of the available variables before each bitcast have been assigned with a value and do not contain default values. When we save those information in lists, we process them with the following order:

1. We delete variables that have not been assigned with any value. That means that those variables contain either their default values, either non-predictable values.
2. We delete variables that in the same program execution have different values, in other words we found that the bitcast instruction was executed multiple times in the same run and those variables had at least twice different values before the execution of the bitcast instruction. Those variables are most probably iterators in loops or take random values.
3. We delete variables that in different executions of the program with the same input file, they have different values. That means that those variables take a random value and we cannot use them for the conditional triggering of the bugs, since we cannot predict their values.
4. We delete variables that in different executions with different input files have always the same input. That means, that most probably the value those variables have before each bitcast instruction is not based on the input file provided and therefore we cannot use them for the triggering of the bugs, because they would trigger always the bugs no matter the input file and as a result bug-finding tools could easily identify them.

If any variables survive the above process, that means that those variables take their values from the input files that we provide to the program and as a result we will use them to conditionally trigger our type confusion bugs. For each value of each variable, we will insert a type confusion bug that will be triggered by this specific value. The conditions that will trigger the type confusion bugs will be inserted with the bugs, they are not already declared inside the program and they will check if the variables we identified with the above processing are equal with their according values that we retrieved during the dynamic analysis of the target program. For instance, if from the above process survive 2 variables for a specific bitcast instruction, where for the first variable we got 2 possible values and for the second variable we got one possible value, then we will insert 3 same type confusion bugs before this bitcast instruction, where the two of them will be triggered by the two different values of the first variable respectively and the third bug will be triggered by the value of the

second variable. We save information about the variables and their values, which we will use for the conditional triggering of the bugs, inside a text file so we can use them after when we will want to insert the type confusion bugs into the program.



<u>Figure 1:</u> A visual representation of the way we save the data that we retrieve from the text files that contain the information that the program prints in the terminal. In this example, we have two unique input files and each file we execute it twice and for each execution we get information about the bitcast instructions that are inside the program and for each bitcast instruction we get the variables that are available before each bitcast and the values they have just before the bitcast instruction is executed.

3.5 Insert the type confusion bugs

The fourth and final step to type confusion bug injection is to actually inject the bugs inside the program. In order to do this we take the source files of the program and we compile them again with the help of Clang, but this time we will use another LLVM pass, which is responsible in injecting the bugs.

The first and most challenging thing that this pass needs to clarify, is the hierarchy of the classes and structs that are declared inside the source code of the program. As we mentioned in the background chapter, type confusion bugs are introduced when an illegal down casting has been made, but in order to introduce an illegal down casting in the program we must know which classes/structs are the "parent" classes/structs and which are the "children". Knowing the hierarchy of the classes/structs, will not only help us with the castings, but is also crucial in order to make our own created "Buggy" class (which we inject in every module) to inherit from the wright classes and structs, so we can downcast the objects that the program uses in the bitcast instructions into a pointer that points to a "Buggy" object and therefore make the illegal calls that will introduce type confusion vulnerabilities before each bitcast instruction. In order to learn the hierarchy of the classes/structs, inside the program that we want to insert our bugs, we run some system commands through our pass that call Clang's functionalities that dump the layout of some of the classes that are inside the program. This method does not give us always all the information we need for the hierarchy of the classes inside the program, so we use another one technique to find the hierarchy of the classes/structs and we will cover in much depth later in this thesis. When we learn the hierarchy of the classes/structs, we declare inside the program a new class called "Buggy" like the one in the example 3 (see below), which will inherit from the classes/structs that are the roots in each hierarchy tree of the program we process and our own created "Buggy" class will contain only one member, which will be a function pointer to a void function that takes no arguments. In order to compile the program without any errors, the program we want to inject our bugs must not contain any class named "Buggy", otherwise it will have conflict with the "Buggy" we create.

Next, we go and read the variables and their values that we can use for the conditional triggering of the bugs (which we produced from the third step of this whole process) and based on this information we go and find where inside the LLVM IR of the program those bitcasts instructions are and we insert our type confusion bugs, just before those bitcast instructions. As we mentioned before we will insert one type confusion bug for each value of each variable we retrieve from the processing we have done previously.

An example of the final result that we may get from this final step is represented in example 3. In example 3 we give our final result in a high level C++ language in order to better understand what we want to achieve (in reality our bugs are inserted in the LLVM IR of the source code). In the example, we have one hierarchy tree, where in the root we have the class

"Base" and this class has one child named "Foo". In this example the program expects from the command line the first argument to be a string and the second argument to be an integer. After, it makes a safe upcasting from an object of type "Foo" to a pointer of type "Base", as a result this operation will be translated to the corresponding bitcast instruction in LLVM IR: %10 = bitcast %class.Foo* %3 to %class.Base*. Our pass will go and create inside the program our "Buggy" class, which will inherit from class "Base". We assume that the variable that we will use to conditionally trigger the bug is the variable named "id" and therefore only when this value equals to 1234, our bug will be triggered, otherwise the program will be executed as it was supposed to run (the condition is inserted with the bug and it was not there in the original declaration of the program). The type confusion bug we insert tries to upcast an object of type "Child" to a pointer of type "Base" and then illegally downcast it to a pointer of type "Buggy". Finally, when the illegal down casting is done, we call the function pointer member of the class "Buggy", which value we can determine by using the first command line argument we give to the program and as a result, we can change the control flow of the program. This is a type confusion vulnerability, because the program calls the function pointer of a pointer that points to a "Buggy" type object, but actually in the memory we have saved an object of type "Foo", where its first member is not a function pointer but a table with characters. As a result, an attacker can save in the character table of object named "child_object" an address in the memory and then by calling the function pointer of the "Buggy" pointer he can make the program jump and execute any instruction of the program the attacker wants.

```
class Base {}; // Parent Class

class Foo : public Base { // Child of "Base" Class
public:
  char name[10];
};

class Buggy : public Base { // Child of "Base" Class
public:
   void (*function_pointer)();
};

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

```
  Foo child_object;
  int id = atoi(argv[2]);
  strcpy(child_object.name, argv[1]);
/***Begin of type confusion bug***/
  If (id == 1234) {
     parent_p = &child_object;
     Buggy* buggy_p;
     buggy_p = static_cast<Buggy*>( parent_p); // Unsafe Casting to sibling class "Buggy"
     (buggy_p -> function_pointer)();    //Now we can change the control flow of the
                                                    //program as we want
  }
/***End of type confusion bug***/
parent_p = &child_object; //bitcast instruction
…
   return 0;
}
```

<u>Example 3:</u> The final result (in high level C++ language) after running through all the steps that described in Section 3.1. For simplicity here, we suppose that the input comes from the command line, but for the programs that we studied and we will use later in the evaluation the input was coming through input files. In this example, in argv[1] we expect a string and in argv[2] an integer number. The instruction "parent_p = &child_object;" will be translated to a bitcast instruction in LLVM IR and as a result we will insert a type confusion bug before this instruction by using the classes that participate in this bitcast instruction and the class we injected inside the program named "Buggy". For the conditional triggering of the bug, we will use the variable "id".

Chapter 4

Implementation

Our implementation operates in five stages to inject and validate type confusion vulnerabilities in Linux C++ source code.

1. Compile the program and during the compilation we apply in the IR of every source file a transformation pass, which injects multiple calls to function “printf” before each bitcast instruction that we care about. As a result, when we will run the program it will print in the terminal information about the bitcast instruction and every variable that is initialized before it.
2. Take the executable that is generate from the previous step and run the program twice for each input file that we have and save to text files everything that is printed on the terminal from the executed program.
3. With the text files that we obtain from the previous step, we run a program that tries to identify for each bitcast instruction which variables that are initialized before each bitcast instruction, take their values from the input files that the program reads.
4. Finally, we take the source files of the program we want to inject the bugs and we insert type confusion bugs before each bitcast instruction of the program with the help of another LLVM pass and those bugs are triggered only if the values of some specific variables are the same with those we retrieve from step 3.
5. Test the final executable of the program to see if the bugs are inserted correctly.

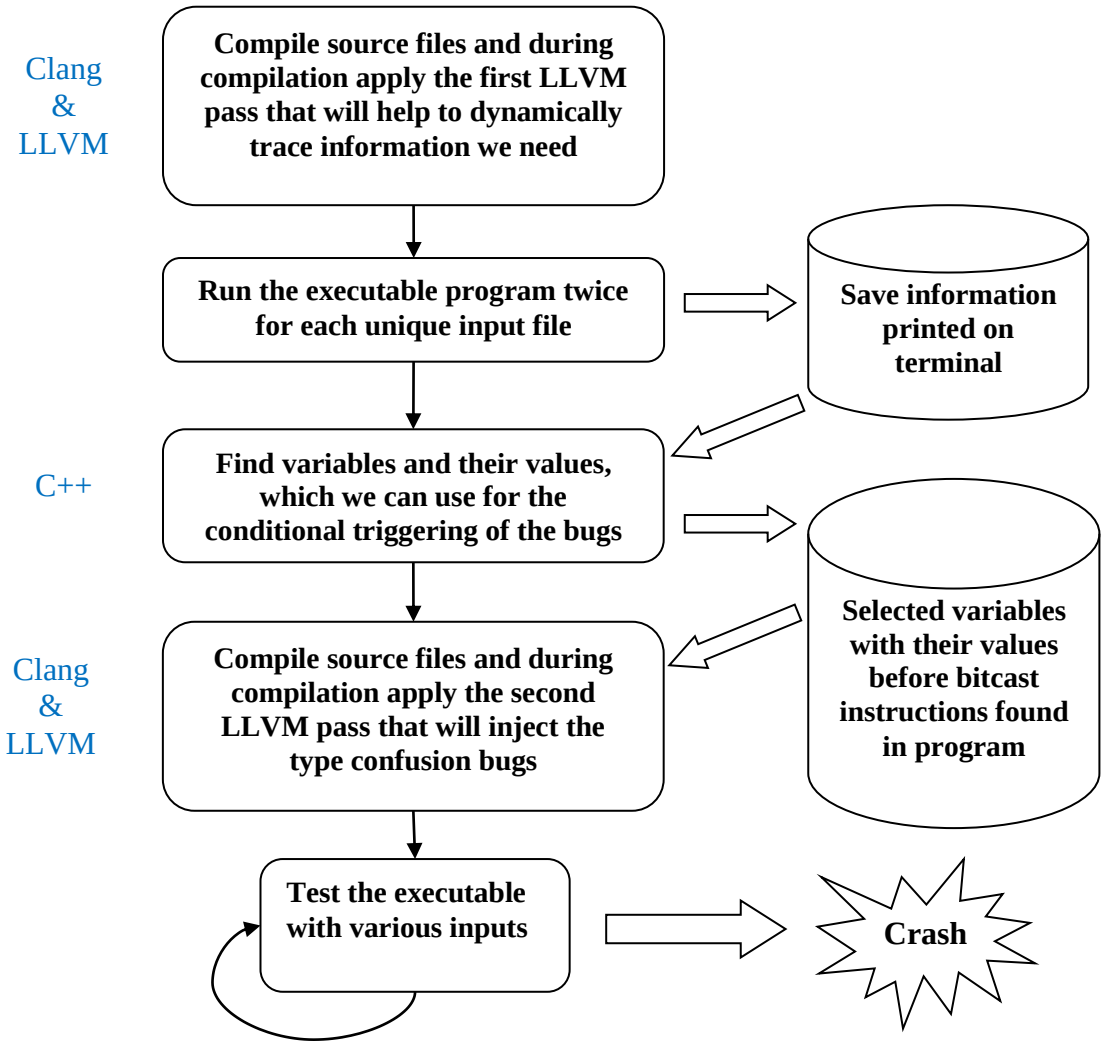


Figure 2: Our implementation architecture. Clang and LLVM are been used to inject multiple calls to "printf" function, so we can dynamically trace information about available variables we may use before each bitcast instruction, which is a candidate to inject our bugs. Next, we find variables in the program that take their values from the input files of the program and we use them to trigger the bugs that we inject. Clang and LLVM are used again to inject the type confusion bugs and finally we do some test to see if our bugs force the program to crash.

4.1 Injecting instructions inside programs to print information about available variables

To trace at run time, what value each available variable before each bitcast instruction has, we are not using any external tool or a virtual machine like it is done in other papers [1][2][4], but instead we insert instructions in the LLVM intermediate representation of the source files of the program. The instructions that we insert are simple calls to C-like "printf" function, that will print in the terminal (during execution time of the program), the information we need for each variable and each bitcast instruction that we care and is

available in the program we want to insert our bugs. One advantage of this technic, is that by retrieving information about variables before bitcast instructions during the execution of the program, guaranties us that the bugs we will insert before a bitcast instruction are reachable during the execution of the program. Generally, we want our bugs to be in the execution path of the program if we give it a valid input and therefore if we get information about a bitcast instruction on the terminal, it means that this bitcast instruction is executed during the run of the program. On the other hand, a disadvantage of this method is that is very time and memory consuming, especially if we process a program which executes bitcast instructions in loops and therefore it will make a lot of printings on the terminal. Also, it cannot be used for programs that are designed to run for large amount of times, like server programs, because this would create large output files which will be very difficult to read and process.

Now we will analyse how this LLVM transformation pass works. The pass will be executed in each module, where in this case a module is a source file of the program we get, written in C++ programming language. First, the pass will try to see if the function "printf" is already declared inside the module, if yes then it gets a pointer to this function, otherwise it creates the declaration of "printf" inside the module and gets a pointer to this newly created function.

After, we iterate through each function that is inside the module and we check to see if this function is a constructor. If it is a constructor then we skip this function and we move to the next, without analysing any further this function. The reason we do this, is because when a class or struct inherits from another class/struct, in the constructors of this class the first thing that is done is to allocate space in memory for the members of this class and after make a cast to an object of the parent type and call the constructor of the parent, passing the object that is casted. As a result, we have a bitcast instruction in the beginning of the constructor and if we try to insert a type confusion bug right before this bitcast instruction, then this bug cannot be exploitable. The reason that the bug will not be exploitable, is because we will not have the ability to write anything in the area of the object we use in the illegal down casting (since this object is not created yet and only the allocation of the memory it needs is done so far) and therefore when we will call the function pointer of our "Buggy" class it will jump either to a random area in the memory or to a non-executable area. As a result, we have chosen to generally do not insert type confusion bugs in the constructors of the classes/structs in order to avoid the possibility of inserting a non-exploitable bug.

Following, if the function is not a constructor of a class/struct, then we go and scan linearly every instruction inside every block of this function. In this pass we only care for Store and

Bitcast LLVM instructions, if we reach a store instruction, then if this instruction stores a value to an Integer, Character, Boolean, Float or Double type variable, then we save this store instruction into a list (in LLVM intermediate representation the types of the variables we search for are i8, i16, i32, i64, float and double, where Char and Bool types in C++ are 8bit integers). Every function has its own list containing the Store instructions described above and the list of a function will contain all the store instructions we found, and meet our criteria, until the instruction we reach in our linear scanning of the function instructions. The reason we search for store instructions, is because we want to find variables that are assigned at least once with a value before a bitcast instruction. As we said, our purpose is to find variables declared before bitcast instructions that take their values from the file that the program reads, in order to use them for the conditional triggering of the bugs.

On the other hand, if we reach a Bitcast instruction, then if this instruction casts an object of a class or struct type to another class or struct type, not union types or primitive types, and we have found at least one store instruction before this instruction inside the same function, then we will insert calls to "printf" before this bitcast instruction which will print information about this instruction and the variables that have been assigned with a value before this instruction. If the bitcast instruction does not meet any of the above criteria, then we go to the next instruction of the function. The reason we check if we found a store instruction before a bitcast instruction, is because we want to insert bugs that will be triggered by a specific input and if there are not any variables that are assigned with a value before a bitcast instruction we could not do that and therefore there is no reason to retrieve any information about that specific bitcast instruction. There is not any constraint regarding the distance of a store instruction and a bitcast instruction, we only care that the store instruction is before the bitcast instruction (in execution order) and both instructions are inside the same function.

When the pass will find a bitcast instruction, which meets our specifications, then it will insert multiple calls to "printf" function just before this instruction. First, it will insert a call that prints into the terminal the IR of this specific bitcast instruction and the function that this instruction was found. By having the name of the function that an LLVM instruction was found and the text that refers to the intermediate representation of that instruction we can identify later that instruction when we will have to process again all the instructions inside the module in order to inject the bugs (in LLVM each instruction inside a function has a unique identifier in front of the instruction that we use to identify the instruction). After, the LLVM pass will insert calls to "printf" that print for every store instruction it found before that bitcast instruction, the name of the function where the instruction was found, along with

the allocation instruction in LLVM IR of the variable that is used in the store instruction and the value that this variable has. For variables of type i8 (8bit integer) and i16 (16bit integer) types we cast those values to 32bit integers and we call "printf" for a 32bit integer value. For variables with i32 (32bit integers) and i64 (64bit integers) types we call "printf" passing those values as they are and the same is done for variables of type Double. Also for variables with Float type we first cast them to Double types and then we call "printf" to print the value as a Double type. Finally, for every store instruction we go and insert before that instruction a call to "printf", which will print to the terminal the LLVM IR of the Alloca instruction of the destination variable that is used inside the store instruction. By this, when we will search for variables that take their input from the input files that reads the program, we will know if a variable was assigned with a value before we reach a bitcast instruction during the execution of the program or it had its default value. In figure 3 on the left side, we see the LLVM intermediate representation of a function before we apply our pass and on the right side, we see how our LLVM pass inserts the calls to "printf". As we can see, before the store instruction we call the "printf" function, which will print on the terminal the "alloca" instruction that is being used to allocate space for the variable we store a value through that specific store instruction. Also, before the bitcast instruction we will print messages that will help us identify the beginning and the end of the information that is about the specific bitcast instruction, where the information we print inside them are the IR of the bitcast instruction and the IR of the "alloca" instruction for which we have a store instruction before the bitcast instruction, along with the value that saved on the space that the "alloca" instructions allocates in memory.

```
; Function Attrs: noinline optnone uwtable
define dso_local void @_Z11finalResultf(float %0) #4 {
  %2 = alloca float, align 4
  %3 = alloca %class.Login*, align 8
  %4 = alloca %class.User, align 4
  store float %0, float* %2, align 4
  call void @_ZN4UserC2Ev(%class.User* %4) #3
  %5 = bitcast %class.User* %4 to %class.Login*
  store %class.Login* %5, %class.Login** %3, align 8
  %6 = load %class.Login*, %class.Login** %3, align 8
  call void @_ZN5Login5helloEv(%class.Login* %6)
  ret void
}
```

```
@str10 = private unnamed_addr constant [34 x i8] c"\0ABegin of bitcast instruction!!!\0A\00", align 1
@str11 = private unnamed_addr constant [70 x i8] c"_Z11finalResultf, %%5 = bitcast %%class.User* %%4 to %%class.Login*\0A\00", align 1
@str12 = private unnamed_addr constant [36 x i8] c" %%2 = alloca float, align 4 : %f\0A\00", align 1
@str13 = private unnamed_addr constant [11 x i8] c"Finish!!!\0A\00", align 1
@str14 = private unnamed_addr constant [114 x i8] c"\0A[Dynamically Trace Bitcasts]Executed store instruction for this: _Z11finalResultf: %%2 = alloca float, align 4\0A\00", align 1
```

	; Function Attrs: noinline optnone uwtable
	define dso_local void @_Z11finalResultf(float %0) #4 {
	%2 = alloca float, align 4
	%3 = alloca %class.Login*, align 8
	%4 = alloca %class.User, align 4
	%5 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([114 x i8], [114 x i8]* @str14, i32 0, i32 0))
	store float %0, float* %2, align 4
	call void @_ZN4UserC2Ev(%class.User* %4) #3
	%6 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([34 x i8], [34 x i8]* @str10, i32 0, i32 0))
	%7 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([70 x i8], [70 x i8]* @str11, i32 0, i32 0))
	%8 = load float, float* %2, align 4
	%9 = fpext float %8 to double
	%10 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([36 x i8], [36 x i8]* @str12, i64 0, i64 0), double %9)
	%11 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([11 x i8], [11 x i8]* @str13, i32 0, i32 0))
	%12 = bitcast %class.User* %4 to %class.Login*
	store %class.Login* %12, %class.Login** %3, align 8
	%13 = load %class.Login*, %class.Login** %3, align 8
	call void @_ZN5Login5helloEv(%class.Login* %13)
	ret void
	}

Figure 3: On the left, we have a function in LLVM IR, which has a bitcast instruction and before this bitcast instruction it has a store instruction (both are with blue colour). On the right, we see how our pass that injects multiple calls to "printf" will modify this function (in bold we see the instructions that the pass inserts).

4.2 Dynamically tracing information for inserting type confusion bugs

As we mentioned earlier, in order to inject bugs to a program except from the source files of the program, we need at least two different input files that the program reads in order to execute normally. Those files will help us identify which variables inside the program take their values from the input file that the program reads and who not, so it is very important to have a good sample of those input files. After, when we compile the program (that we want to inject the bugs in it) with the LLVM pass that injects multiple calls to "printf", we go and

run the executable that is produced 2 times with each input file of the program we have. When we execute the program, we save all the messages that the program prints on terminal inside different text files. For instance, if we have 2 different input files of the program, then we will run the program two times for each input file and therefore we will get 4 text files with the messages the program printed on the terminal in each execution. An example of the output that we might get after running the program in which we applied our LLVM pass, is shown in figure 4, where after the phrase "[Dynamically Trace Bitcasts]" we get the allocation instruction of the variable which a store instruction is executed ("alloca" instructions give us information about the variables that the program uses) and between the phrases "[Begin of bitcast instruction]" and "[Finish]" we get information about the values of the available variables before a specific bitcast instruction. After we get those files, we run a program that tries to identify the variables we are looking for.

Picture 1

```
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %6 = alloca i16, align 2
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %7 = alloca i32, align 4
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %8 = alloca i64, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %9 = alloca float, align 4
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %10 = alloca double, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %12 = alloca i8, align 1
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %14 = alloca i64, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %16 = alloca i64, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %14 = alloca i64, align 8

[Begin of bitcast instruction]
main,  %39 = bitcast %class.Foo* %17 to %class.Base*
  %6 = alloca i16, align 2 : 10
  %7 = alloca i32, align 4 : 12345
  %8 = alloca i64, align 8 : 4200790
  %9 = alloca float, align 4 : 2.100000
  %10 = alloca double, align 8 : 5.500000
  %12 = alloca i8, align 1 : 108
  %13 = alloca i8, align 1 : 0
  %14 = alloca i32, align 4 : 0
  %16 = alloca i64, align 8 : -559038737
[Finish]
```

```
[Begin of bitcast instruction]
main,  %39 = bitcast %class.Foo* %17 to %class.Base*
  %6 = alloca i16, align 2 : 10
  %7 = alloca i32, align 4 : 12345
  %8 = alloca i64, align 8 : 4200790
  %9 = alloca float, align 4 : 2.100000
  %10 = alloca double, align 8 : 5.500000
  %12 = alloca i8, align 1 : 108
  %13 = alloca i8, align 1 : 0
  %14 = alloca i32, align 4 : 1
  %16 = alloca i64, align 8 : -559038737
[Finish]
```

Picture 2

```
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %6 = alloca i16, align 2
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %7 = alloca i32, align 4
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %8 = alloca i64, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %9 = alloca float, align 4
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %10 = alloca double, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %12 = alloca i8, align 1
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %14 = alloca i64, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %16 = alloca i64, align 8
[Dynamically Trace Bitcasts]Executed store instruction for this: main:  %14 = alloca i64, align 8

[Begin of bitcast instruction]
main,  %39 = bitcast %class.Foo* %17 to %class.Base*
  %6 = alloca i16, align 2 : 10
  %7 = alloca i32, align 4 : 12345
  %8 = alloca i64, align 8 : 4200790
  %9 = alloca float, align 4 : 2.100000
  %10 = alloca double, align 8 : 7.300000
  %12 = alloca i8, align 1 : 110
  %13 = alloca i8, align 1 : 0
  %14 = alloca i32, align 4 : 0
  %16 = alloca i64, align 8 : -559038737
[Finish]
```

Figure 4: In the first picture, we see a part of the output we may get in terminal when executing a program, which we compiled and applied the LLVM pass, which injects multiple

calls to "printf" in order to dynamically trace the values of the available variables before bitcast instructions that meet our criteria. Here we get information about the bitcast instruction: "main, %39 = bitcast %class.Foo %17 to %class.Base*" and the values of 9 different variables that have been initialized before this instruction. From the information we get we know that this bitcast instruction is located in the function with name "main" and its IR is: "* %39 = bitcast %class.Foo* %17 to %class.Base**" that cast a pointer of class "Foo" to a pointer of class "Base". We also get information about which of the variables are assigned with a value and do not contain their default values. On the second picture, we see the messages printed on terminal, when we execute the same program, but with a different input file.*

In order to find variables that are both initialized before a bitcast instruction we care and take their values from the input file of the program, we initially save in lists the information we get after running the program 2 times for each unique input, as it is shown in figure 1 (Page 18). Where we save to a different list, information that come from a single execution of the program given an input file and we group lists that come from different runs of the program, but with the same input file into a single list. For each run of the program we have a list with the bitcast instructions that are executed during the run of the program and for each bitcast instruction we have a list with the "alloca" instructions, along with the values that they had during the execution, that correspond to the available variables before that bitcast instruction. Additionally to what is shown in figure 1, we also have a list for each unique file where we save which store instructions have been executed in the first time we run the program with that specific input file (we have a list for each unique input file and not for each run, because for multiple runs with the same input file we will always get the same information about which store instructions have been executed during the execution of the program).

The first stage of processing those data goes and deletes in each list, that contains information about a run of the program, the variables (in this case their "alloca" instructions) for which we know that none store instruction is executed for storing any value in this variables. In the figure 4, such a variable would be the variable that is related to the "alloca" instruction: "%13 = alloca i8, align 1", for which the program does not print any information with the label "[Dynamically Trace Bitcasts]". Following, if a bitcast instruction is executed more than once during a single run, then we go and compare the values that each variable has in each execution of the bitcast instruction and if a variable has different values in different executions of the same bitcast instruction in the same run of the program, then we delete the "alloca" instruction that corresponds to this variable. We do this, because most probably the

bitcast instruction is inside a loop and the variable that changes its value from one call of the bitcast instruction to the other, is an iterator or takes random values and therefore it is not useful for triggering a type confusion bug we may insert before that bitcast instruction. An example of the "alloca" instructions that we delete during this process is shown in figure 4, where we would delete the "%14 = alloca i32, align 4" instruction, for which in the first execution of the bitcast instruction it has the value 0 and in the second execution it has the value 1. When we finish with the above processing, we delete from our lists any bitcast instructions that have no more valuable variables we can use before their execution.

In the second stage of our processing, we compare the values of each variable that is related to an individual bitcast instruction, between different runs of the program for the same input file. If a variable has a different value in a different run of the program but with the same input file with another run, then we delete this variable from the possible variables we can use for triggering our bugs. This processing is being done in order to identify variables that may get random values. For such variables, we cannot predict the value they may get during the execution of the program and as a result, they are not good variables for triggering our bugs. Also, this stage of processing is the reason why we need to execute the program twice for each unique input file that the program can get. When we finish with the above processing we delete again from our lists any bitcast instructions that have no more valuable variables we can use before their execution.

On the next and final stage, we go and combine all the values of each variable that is related with the same bitcast instruction and therefore for each unique input file (from the sample files we have about the program), we have a list that contains for each bitcast instruction that is executed given that input file, all the values that each variable that passed the two initial stages of processing has. When we create those new lists, we go and search if a bitcast instruction is executed in all the executions of the program with different input files and if we find such cases, then we delete variables that in all runs of the program with different input files they have the same value just before the bitcast instruction is executed. That means, that the value those variables get does not come from the files that the program reads or it may come from the input files, but we cannot know by the sample files we have. For example, from the figure 4 the only "alloca" instructions that we will use for the triggering of the bugs that we will insert before that bitcast instruction, are "%12 = alloca i8, align 1" with values 108 and 110 and "%10 = alloca double, align 8" with values 5.5 and 7.3, we reject all the other variables because they have the same value in different executions of the program with different input files. On the other hand, if a bitcast instruction is not executed by all the input

files that we give to the program, then we will keep all the variables that this bitcast instruction has until this time of processing and all the values that those variables have will trigger a different type confusion bug that we will put right before the bitcast instruction. We do this, because if a bitcast instruction is not executed by all the input files we give to the program, then it means that a value inside the input file is responsible for executing this bitcast instruction and therefore whatever variable we will use for the triggering of the bugs that are injected before this bitcast instruction, indirectly they will be triggered by a value that comes from the input file.

In conclusion, any variables that survive from the above processing, are those that we will use for the triggering of the type confusion bugs we will inject to the program. For each value of each variable, we will put a different type confusion bug just before the bitcast instruction to who is related this variable. The information about the bitcast instructions and the variables that we will use for the triggering of the bugs, are saved into a single text file, which will read our second LLVM pass that will inject the bugs inside the program.

4.3 Inject and test the bugs

We studied various ways we could insert a type confusion bug inside a program. In our first experiments, we tried to insert type confusion bugs inside programs written in C programming language. In order to achieve this we were casting a pointer that was pointing to a struct type into a void() pointer. By doing this we could call the void() pointer and change the control flow of the program based on the value of the first 4 or 8 Bytes, depends on the machine architecture, of the struct we casted (we can see the way we were inserting this kind of type confusion bugs in LLVM IR in figure 5). However, type confusion bugs are not so common in C programming language, it is very rear to find castings from struct types to struct types in programs written in C and therefore we concentrated on programs written in C++ where it gives us the ability to create classes and structs and inherit from another classes or structs. We tried to inject type confusion bugs inside C++ programs with the same way as we were inserting type confusion bugs in C programming language based programs and such bugs were causing the programs in C++ to crash with the same way as they were crashing on programs written in C. However, normally in C++ programming language, we cannot cast a struct or a class into a void pointer and therefore we tried to find another way to insert type confusion bugs in C++ programs.

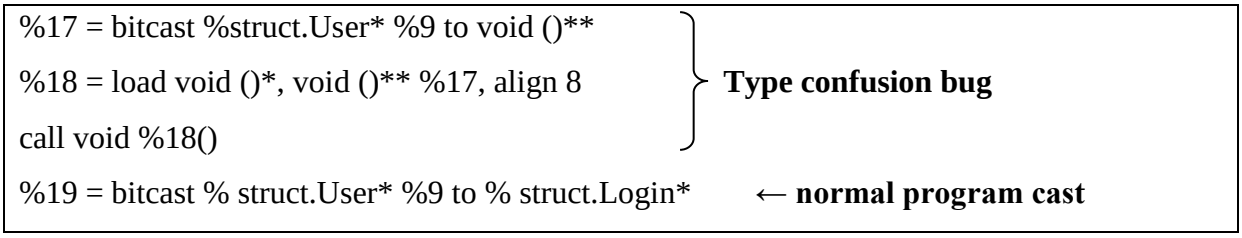


Figure 5: Example of a type confusion bug in programs written in C programming language in LLVM intermediate representation. We simply inject the bug before a normal casting between structs, that takes place inside the program and the bug just casts the struct pointer to a void pointer and calls the void pointer.

Instead of casting pointers to classes/structs into void pointers, we decided to create a new class named "Buggy" inside the program we inject the bugs, which will inherit from the wright classes/structs that are already defined inside the program, in order to be able to cast pointers of other classes or structs to a pointer of "Buggy" type. After, through the pointer to our own created class we can call a function pointer that is a public member of our "Buggy" class, that will change the control flow of the program based on the value of the first member of the class/struct we casted from. The reason we declare only a function pointer as a member of our "Buggy" class, is because this will help us introduce a type confusion vulnerability. The purpose of the function pointer is to change the control flow of the program based on the value of the first 4 or 8 Bytes (depending on the machine architecture if it is 32-bit or 64-bit, accordingly) that are being used to save the values of the members of the object we casted to a "Buggy" type. From then we suppose that the attacker has the ability to change those Bytes and put there any valid value that will make the function pointer change the control flow of the program and execute any instruction the attacker wants. An example of the "Buggy" class we create along with the way we can make a type confusion attack by using it is demonstrated in high-level language on example 3 (page 21).

We will describe now in depth how we inject the type confusion bugs by using an LLVM pass and the information we retrieve from the dynamic analysis of the program that we discussed in the previous subsection. In order to create the class "Buggy" we need to know the inheritance hierarchy for all the classes and structs of the current module we process (again our pass is processing the source code of the program module by module). Unfortunately, in LLVM we did not find a way that we could obtain the inheritance hierarchy of the classes/structs that are declared inside a module. Therefore, to obtain information about the inheritance hierarchy of the classes and structs inside a program written in C++, we run a system command (before our pass starts to process anything), which calls Clang with

options “-cc1 -fdump-record-layouts” (in figure 6 we can see the information we take when we run this command for the program that is demonstrated in example 3). This command returns us a string, which contains the inheritance hierarchy for the most classes and structs that are declared inside the program we pass to it. So we take this string and we process it in order to find the class or struct that is in the root of each inheritance tree that is returned from this command and when we know those classes/structs we create our “Buggy” class, which inherits from those structs or classes. As a result, we will be able to cast from any object that its type is inside those inheritance trees to a pointer that points to a “Buggy” type, by simply upcasting the object to a pointer that points to a class/struct which is the root of its inheritance hierarchy and from it we will downcast to a “Buggy” type pointer. For example, if we have inside the program (where we will inject the bugs) the following classes: A, B, C, D, E and F and their relationships are: B→A (B inherits from A), C→B, D→B and E→F, then our “Buggy” class will inherit from the classes A and F (which are the roots of the two inheritance hierarchies we have inside the program) and if we have an object of type D inside a function, we can upcast it to a type A and then downcast it to a “Buggy” type. When we create the “Buggy” class, we declare only one public member inside it, which will be a function pointer to a void function without parameters. This method of obtaining the inheritance hierarchy of a program it does not give us the inheritance hierarchy of each class/struct that exists in the program, but it gives us information for the most of the classes and structs that we want and it is acceptable for what we want to do. In the case, we find a bitcast instruction that contains classes/structs for which we did not obtained their inheritance hierarchy through the method we described above, then we will try to see if one of the two classes/structs that take part in the casting has as a member an object of the other class/struct. Therefore this means, that the object that contains as a member an object of the other class/struct is the child and the other class/struct is the parent (note that this statement is valid because the two classes/structs are being used in a casting instruction and therefore they have a parent-child relationship). As far as we know, a class or struct cannot contain as a member an object, that its type is a class or struct that inherits from this class/struct, but the opposite is possible and based on this knowledge we make this second attempt to find the inheritance hierarchy of the two classes/structs. Following, in order to make the illegal down castings, we create a new class like “Buggy”, but we call it “LocalBuggy” and this class will inherit only from the class or struct that is the parent between the two classes/structs that participate in the specific bitcast instruction and this “LocalBuggy” class will be used for the type confusion bugs that we will put right before this specific bitcast instruction only. If even after all this process we still do not know the relationship between the classes/structs that take

part in a bitcast instruction, then we will skip that bitcast instruction and we will not insert any type confusion bugs before that instruction.

```
*** Dumping AST Record Layout
         0 | class Base (empty)
           | [sizeof=1, dsize=1, align=1,
           |  nvsize=1, nvalign=1]

*** Dumping AST Record Layout
         0 | class Foo
         0 |   class Base (base) (empty)
         0 |   char [10] name_
           | [sizeof=10, dsize=10, align=1,
           |  nvsize=10, nvalign=1]
```

Figure 6: The results we get when we run the command "clang -cc1 -fdump-record-layouts example.cpp" for the program given in example 3.

Right after the LLVM pass creates the "Buggy" class it reads the information, which we produced during the dynamic analysis of the program, about the bitcast instructions and the available variables that we can use for triggering the bugs that we will insert inside the program. The pass, then saves those information in a list and after, it scans all the instructions inside every block of every function of the current module, in order to find the bitcast and alloca instructions that are contained inside the information it got from the dynamic analysis and save pointers to those instructions. Next, when the LLVM pass scans all the instructions inside the current module, it iterates through all the bitcast instructions that we have retrieved from the dynamic analysis of the program and inserts type confusion bugs before those bitcast instructions, if it is possible.

As we mentioned earlier, to inject type confusion bugs based on the classes/structs that are participating in a casting instruction we must know who is the root class/struct of the inheritance hierarchy of the two classes/structs that are participating in the bitcast instruction. In order to do this, the LLVM pass firstly checks to see if it knows the inheritance hierarchy of at least one of the two classes/structs from the information it got when it run the Linux command that called clang with options "-cc1 -fdump-record-layouts". If it received the inheritance hierarchy of one of the classes that participate in the bitcast instruction from the

command it run, then it retrieves the class/struct that is in the root of this inheritance hierarchy. Then, it inserts before the bitcast instruction an if-else statement, for each value of each variable that we retrieved from the dynamic analysis of the program. For each if-else statement, we just check if the value of the corresponding variable that is declared before the bitcast instruction and is inside the variables that have been chosen during the analysis of the variables, is equal to the value we retrieved from the dynamic analysis. If the condition is true, then the pass creates a new block in which the program will jump inside and there we put a type confusion bug, otherwise if the condition is false we call the "printf" function which will print a message on the terminal that will notify that the program did not triggered the bug (we do that, just to know if a bug is not triggered during the execution of the program). The block with the type confusion bug, upcasts the object that is being used in the bitcast instruction to the class/struct type that we identified as the root of the inheritance hierarchy of the two classes/structs participating in the bitcast instruction. After, it casts the object to a "Buggy" type and finally calls the function pointer that is a public member of the "Buggy" class. If the variable that we use in the if-else statement is of integer type (not Float or Double) and the value we get from the dynamic analysis of the program is negative, then in the else statement the pass does not inserts a call to "printf", but instead inserts another if-else statement which compares if the value of the variable equals with the unsigned form of the value we retrieved. If the condition is "true", then the program jumps to the block that contains the type confusion bug, that was created for the first if-else, otherwise if the condition is "false" then it calls "printf". The reason we do this, is because in LLVM IR for integer types we cannot know if the value they contain is in signed or unsigned form. As a result, when we inserts calls to "printf" (in the pass that helps us to identify dynamically the values of the variables) we print the values of the integer variables in their signed form, so if an integer variable has a positive value this value remains the same either we signed it or not, but if the integer variable has a negative value this value is different in its signed and unsigned form. Therefore, we insert the if-else statements in such way, so if an integer variable has a positive value (either is signed or unsigned) the bug we insert will be triggered from the first if-else statement, also it will be triggered from the first if-else statement if the variable has a negative value and is signed, otherwise if the variable has a negative value and it is unsigned, the bug will be triggered through the second if-else statement. It is important to say, that we insert one and only one type confusion bug for each value of each variable that we retrieved from the dynamic analysis of the program and when we compare the value of an integer variable with its unsigned form and this condition is "true", we jump to the bug we created for the signed form of this specific value. A brief example that shows the LLVM IR

instructions we create for different values we take from the dynamic analysis is demonstrated in Appendix A.

If the LLVM pass does not have information about the inheritance hierarchy of the two classes/structs that participate in a bitcast instruction, based on the information it received running the command "*clang -cc1 -fdump-record-layouts [name of the source file].cpp'*", then it makes the secondary check we discussed previously (if one of the two classes/structs has as a member an object that its type is the other class/struct, then the class/struct that contains the other class/struct as a member is the child). If from this check, it finds who is the parent class/struct from the two classes/structs we have in the bitcast instruction, then it creates a new class named "LocalBuggy" that inherits from the parent class/struct and has a public function pointer as the "Buggy" class. Then it follows the same steps to insert the type confusion bugs before the bitcast instruction as described above, but with the only change that it will not make a down casting to the "Buggy" type, but to the "LocalBuggy" type. If we must create a "LocalBuggy" more than once for a module, then LLVM assigns in the end of each "LocalBuggy" a unique id, which will allow LLVM to know which "LocalBuggy" to use in each case.

When our LLVM pass process all the source files of the program we provide, it will generate an executable, which will have the same functionality as the original program with the only difference that if we provide to it input files with some specific values, we will trigger some of the type confusion bugs we have inserted. In order to test that our bugs work properly, we implemented some toy programs (that take their inputs from files) where we applied our passes and then we run the final executable we produced with the right values in the input files we provided. The results we got is that each type confusion bug was triggered as we expected and caused the program to crash. In order to be sure that the crashing of the program was due to our bugs and not any other reason, we used a debugger which gave us the possibility to write in the first bytes of the class objects we used in our bugs, some valid instruction addresses and when we executed the program it did not crashed when it reached the bugs, but instead it changed its control flow to the addresses we specified.

Chapter 5

Evaluation

To evaluate our work we created three test programs in C++ where we injected our type confusion bugs and then we run a bug finding tool with the executables we produced and counted how many of the bugs we inserted it could found. We run our tests, in a pc running Ubuntu- 20.04.2.0, with 8 GB Ram, Intel core i7 (8^{th} generation) and HDD hard disk.

5.1 Description of the programs we tested

The first program we created was a simple program written in C++ where we inserted two classes (where the one of them was inheriting from the other one) and some simple functions, that were making some basic computations and printings based on the numbers and characters the program was reading from a file. On the second program, we implemented a Sudoku solver written in C++, that was reading a Sudoku grid from a file and then it was trying to solve it. For the first two test files, we inserted in various locations some safe castings between classes and structs we declared inside the programs, so our LLVM passes can track those casts as possible locations for inserting the type confusion bugs. In addition, we created a third program that had the same functionality as the first test program, but in this we declared some public members inside the classes and structs we had, where those members were function pointers that were pointing to a specific function that is declared outside the class/struct definition. We made this, because in the first two programs the type confusion bugs that we are going to insert will always lead the programs to crash, because the program will jump to places that we cannot execute and therefore it will send a segmentation fault. On the other hand, on our third program when we will make the illegal down castings to our “Buggy” class, then we will call the function pointer of the buggy class which will call the function pointer of the object we casted (because of the type confusion) and therefore the

program will jump to the function we declared and execute normally without crashing. As a result, we can check if the bug-finding tool can detect type confusion bugs that do not make the program crush.

5.2 Description of the bug-finding tool

The bug-finding tool we used for our evaluation is "American Fuzzy Lop plus plus (afl++)" [5], who it is an extension of Google's AFL. This bug finding tool uses a technic called "Fuzzing" in order to detect bugs inside the program. Generally, fuzzers try to automatically find bugs inside programs by generating invalid, unexpected or random data and giving them as inputs to the programs they want to test. The goal of fuzzing is to stress the application and cause unexpected behavior, resource leaks or crashes, something similar is done by "American Fuzzy Lop plus plus (afl++)". For our purposes we are doing black-box fuzzing, that means that we do not compile the source files of the target program with the compilers that are provided by the fuzzer, but instead we give to the fuzzer the executable we produced and the fuzzer with the help of qemu virtual machine tries to find bugs inside the program. In addition, in order for the fuzzers to know what kind of data each of our test program read and therefore be able to generate similar data to exploit the bugs, we provide to the fuzzer a sample input file for each test program that does not trigger any of the bugs that we inserted inside each program.

5.3 Results

	Test Program 1	**Test Program 2**	**Test Program 3**
Time to compile with the first LLVM pass (in seconds)	0,335	0,473	0,315
Time to find variables to use in the conditions (in seconds)	0,02	0,118	0,061
Time to compile with the second LLVM pass (in seconds)	0,324	0,677	0,33
Number of new instructions inserted in first LLVM pass	583	2206	656
Number of new instructions inserted in second LLVM pass	287	1618	419

Figure 7: Table with information about the time our LLVM passes wanted to compile and process the test programs, along with the number of new instructions the passes inserted in each occasion.

	Test Program 1	Test Program 2	Test Program 3
Number of unique type confusion bugs inserted	26	147	38
Number of bugs found by AFL++	26	35	26
Time spent AFL++ to find the bugs	05:02:52 hours	3 days & 11:47:55 hours	2 days & 16:11:00 hours

Figure 8: Here we present the number of bugs we inserted inside each test program and how many bugs did the fuzzer found during the period we let it run.

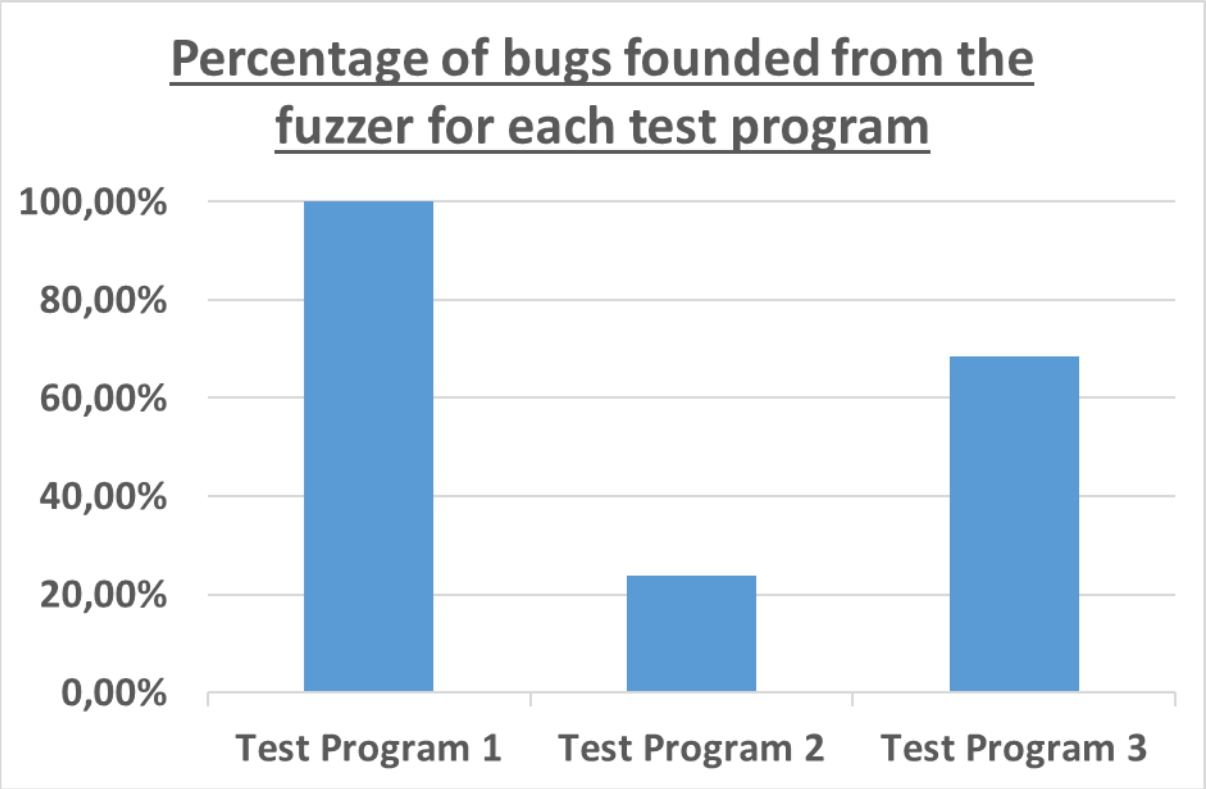


Figure 9: Graph with the percentage of the bugs the fuzzer managed to found for the time we let it run.

For the evaluation, initially we processed each test program to find variables to use for the conditional triggering of the bugs by using our first LLVM pass and the program that finds the values we can use. Then based on the information we collected, we compiled each program with the LLVM pass, that injects the type confusion bugs and then we run the fuzzer with the executables we produced. In figures 7 and 8, we present the data we collected for each test program. From the figure 7, we can see the time we wanted to compile each program with each one of our two passes and the time we wanted to find the variables we can use for the conditional triggering of the bugs. In figure 8, we see the number of bugs we

inserted in each program and how many of them the fuzzer managed to find. As we can see, in the first program the fuzzer managed to find all the type confusion bugs we inserted in just 5 hours. On the other hand, in the second test program, we let the program to run for about 3 days and 11 hours and during this time it only found 35 out of 147 different type confusion bugs we inserted, where this is only 24% of the bugs we inserted. Finally, in our third test program the fuzzer found 68% of the type confusion bugs we inserted during 2 days and 16 hours we let the program to run. In conclusion, from the above results we can see that our bugs are traceable by the fuzzer and therefore they are real bugs that can be found inside programs and also the fuzzer cannot find all of them, which means those bugs are not so easy to find.

Chapter 6

Conclusions

6.1 Problems we faced

During our research, the most difficult part was to find a way to trace variables that are being used during the normal execution of the programs and we could use them for the conditional triggering of the bugs. We did not only wanted any available variables, but we wanted variables that with different input files they were getting a different value, so our bugs could be triggered by specific values given to an input file. For this part, we spend a big time researching during both semesters and we tried different methods used in other related papers, like LAVA [1]. Specifically, we tried to use parts of LAVA, that were responsible for finding DUAs (Dead, Uncomplicated and Available data) inside programs, which is something similar with what we wanted to find, but LAVA was implemented for injecting bugs in C source code and therefore those mechanism did not work for C++ programs. After, when we studied more deeply how LAVA was working, we found that it was using a program called “Panda”, which could dynamically trace variables that were taking their input from input files, and we tried to use “Panda” to find variables that take their values from input files and therefore process those information to find variables we can use to insert our bugs conditionally. After a lot of time trying to understand “Panda”, we managed to run “Panda” to trace variables on some test files we made, but the results we got were not as we expected, “Panda” plugins were giving us the location of the variables in the executable program and not in the LLVM IR as we wanted and therefore they were not so useful to us. After all this unsuccessful attempts, we came up with the idea of using calls to “printf” (which we will inject inside the program in LLVM IR with the help of a pass) in order to

trace the available variables inside the program and determine which of them take their values from the input files.

6.2 Related work

Evaluating bug-finding tools has been an open problem since the introduction of fuzzing in 1990 as a method for discovering bugs inside programs. One method of evaluation a bug-finding tool is to see how many bugs this tool can find inside real life programs, but this is a difficult task since there are not so many real life programs with a significant number of bugs inside them and also usually bug-finding tools are trained to find already known bugs. In 2016, LAVA [1] introduced synthetic bug generation, as a method to overcome the limitations of relying on known vulnerabilities for fuzzer evaluation. LAVA suggested a method to automatically inject a big number of buffer overflow bugs inside programs that are both not easily traceable from bug-finding tools and are exploitable. In order to achieve this they used DUAs (dead-uncomplicated and available data) so the bugs are being triggered only when a specific input is given to the programs. After LAVA, many similar researches have been made for automatically injecting a large number of different types of bugs inside programs, like DRInject [2], where there they tried to inject a large number of data race bugs that were triggered when programs were running concurrently and provided us a large corpora of data race bugs with which we could evaluate bug-finding tools. Injecting a large number of bugs cannot only used in the evaluation of bug-finding tools, but it can be used and in other occasions like in [4] where they used LAVA to inject a big number of bugs for Capture the Flag competitions, where security teams competitively attack and/or defend programs in real time.

In recent years, many bug-finding tools have been proposed in order to trace and report different types of bugs inside programs. One of them is AFL++ [5], which is an extension of the original American Fuzzy Loop and tries to identify bugs inside programs by using fuzzing. AFL is one of the most tested fuzzers and the fuzzing technic is used widely for identifying bugs inside programs. Another way for detecting bugs is like the one presented in paper [7], where they implemented a compiler level detection of type confusion bugs, that relies on an efficient per-object metadata storage service based on a compact memory shadowing scheme. To identify which technic is better for finding bugs is difficult to say since there are many out there and their evaluation is a difficult task, due to the lack of available corpora, as we already discussed. In paper [6] they presented a way for evaluating the efficacy of synthetic bug injection for comparative fuzzer evaluations and also pointed

out several pitfalls of conducting fair fuzzing evaluations that have not been reported in the literature. In addition, they suggested different ways for improving future synthetic bug injection techniques, like the one we suggested in this thesis and can be used to make more realistic the bug injections.

6.3 Limitations and Future work

One of the limitations of our work, is the part that finds variables to conditionally trigger the bugs. This specific part makes too many computations to find possible variables and their values, which we can use for the conditional triggering of the type confusion bugs. As a result, for programs that call many times a casting instruction and have many variables inside them to process, our program becomes very slow. So in the future, we would like to either speed up this part or even find another way to dynamically trace and process, more efficiently the variables and their values inside a given program. Also, we would like to not only use boolean, integers, characters and floating point variables for the conditional triggering of the bugs, but also values in tables and maybe even class or struct members. In addition, in the future we would like to inject more complex type confusion bugs, which would be more similar to type confusion bugs we can find in real life programs.

6.4 Conclusion

In this thesis, we analysed how we managed to automatically inject a big number of type confusion bugs inside programs written in C++. We explained how our LLVM passes work and how we managed to use them to dynamically trace information about casting instructions and variables that are available inside a program and how we injected multiple type confusion bugs inside a program. At the end, we evaluated our work and we saw that it could help to evaluate how good are some bug-finding tools in finding type confusion bugs and even help to make those tools better. We believe that our work will help the community to develop more tools that will automatically inject a large number of bugs inside programs in order to create corpora that can be used to evaluate bug-finding tools.

References

[1] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robert-son, F. Ulrich, and R. Whelan, “Lava: Large-scale automated vulner-ability addition,” inProceedings of the IEEE Symposium on Securityand Privacy, ser. SP ’16, San Jose, CA, USA, 2016.

[2] H. Liang, M. Li and J. Wang, “Automated Data Race Bugs Addition”, April 2020

[3] The LLVM Compiler Infrastructure, https://llvm.org

[4] P. Hulin, A. Davis, R. Sridhar, A. Fasano, C. Gallagher, A. Sedlacek, T. Leek and B. Dolan-Gavitt, “AutoCTF: Creating Diverse Pwnables via Automated Bug Injection”, May, 2017

[5] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research”. In 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug. 2020.

[6] J. Bundt, A. Fasano, B. Dolan-Gavitt, W. Robertson, and T. Leek, “Evaluating Synthetic Bugs”. In 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS ’21), June 7–11, 2021, Hong Kong, Hong Kong.

[7] I. Haller, Y. Jeon, H. Peng, M. Payer, C. Giuffrida, H. Bos and E. van der Kouwe, “TypeSan: Practical Type Confusion Detection”, October, 2016

Appendix A

Here we will demonstrate how with our LLVM pass and information we retrieved by dynamically tracing the program, we inject type confusion bugs inside the source files of a program in LLVM IR.

In this example we assume that the information we get about a specific bitcast instruction (that is inside the program) after processing the information about the available variables before that bitcast instruction and their values we can use, are the following:

```
_Z7examplecbsilfdm,  %20 = bitcast %class.Foo* %18 to %class.Base*
10
  %9 = alloca i8, align 1 : 97
  %9 = alloca i8, align 1 : 112
  %10 = alloca i8, align 1 : 0
  %11 = alloca i16, align 2 : 21
  %12 = alloca i32, align 4 : 2597
  %13 = alloca i64, align 8 : 1000000
  %14 = alloca float, align 4 : 5.400000
  %14 = alloca float, align 4 : 10.500000
  %15 = alloca double, align 8 : 7.356000
  %16 = alloca i64, align 8 : -559038737
```

From the above information we understand that we insert 10 type confusion bugs just before the bitcast instruction that is inside a function named “example” and its IR is “%20 = bitcast %class.Foo* %18 to %class.Base*”. The 10 type confusion bugs we will insert, they will be all the same, but they will be triggered by a different value of each variable we get from the information above. Before inserting the bugs, we create a class named “Buggy” which will inherit from the class “Base” (we suppose that class “Foo” inherits from class “Base” and class “Base” does not inherits from any other class or struct) and inside “Buggy” we will have a public function pointer which will pointe to a void function without parameters. The type confusion bugs we will insert, they all will cast the object saved at %18 from a class “Foo” type to a “Base” type. After it will cast this pointer of a class “Base” to a pointer to a class “Buggy” and it will call the function pointer member of the “Buggy” class. The result of this processing is shown in the next two pictures, where in the first we have the “example”

function before we insert the bugs and in the second the same function after we insert the bugs.

Program LLVM IR before inserting the bugs:

```
; Function Attrs: noinline optnone uwtable
define dso_local void @_Z7examplecbsilfdm(i8 signext %0, i1 zeroext %1, i16 signext %2, i32 %3, i64 %4, float %5, double %6, i64 %7) #4 {
  %9 = alloca i8, align 1
  %10 = alloca i8, align 1
  %11 = alloca i16, align 2
  %12 = alloca i32, align 4
  %13 = alloca i64, align 8
  %14 = alloca float, align 4
  %15 = alloca double, align 8
  %16 = alloca i64, align 8
  %17 = alloca %class.Base*, align 8
  %18 = alloca %class.Foo, align 4
  store i8 %0, i8* %9, align 1
  %19 = zext i1 %1 to i8
  store i8 %19, i8* %10, align 1
  store i16 %2, i16* %11, align 2
  store i32 %3, i32* %12, align 4
  store i64 %4, i64* %13, align 8
  store float %5, float* %14, align 4
  store double %6, double* %15, align 8
  store i64 %7, i64* %16, align 8
  call void @_ZN3FooC2Ev(%class.Foo* %18) #3
  %20 = bitcast %class.Foo* %18 to %class.Base*
  store %class.Base* %20, %class.Base** %17, align 8
  %21 = load %class.Base*, %class.Base** %17, align 8
  call void @_ZN4Base5helloEv(%class.Base* %21)
  ret void
}
```

Program LLVM IR after inserting the bugs:

```
; Function Attrs: noinline optnone uwtable
define dso_local void @_Z7examplecbsilfdm(i8 signext %0, i1 zeroext %1, i16 signext %2,
i32 %3, i64 %4, float %5, double %6, i64 %7) #4 {
  %9 = alloca i8, align 1
  %10 = alloca i8, align 1
  %11 = alloca i16, align 2
  %12 = alloca i32, align 4
  %13 = alloca i64, align 8
  %14 = alloca float, align 4
  %15 = alloca double, align 8
  %16 = alloca i64, align 8
  %17 = alloca %class.Base*, align 8
  %18 = alloca %class.Foo, align 4
  store i8 %0, i8* %9, align 1
  %19 = zext i1 %1 to i8
  store i8 %19, i8* %10, align 1
  store i16 %2, i16* %11, align 2
  store i32 %3, i32* %12, align 4
  store i64 %4, i64* %13, align 8
  store float %5, float* %14, align 4
  store double %6, double* %15, align 8
  store i64 %7, i64* %16, align 8
  call void @_ZN3FooC2Ev(%class.Foo* %18) #3
  %20 = load i8, i8* %9
  %21 = icmp eq i8 %20, 97
  br i1 %21, label %22, label %29

22:                                               ; preds = %8
  %23 = alloca %class.Buggy*
  %24 = bitcast %class.Foo* %18 to %class.Base*
  %25 = bitcast %class.Base* %24 to %class.Buggy*
  store %class.Buggy* %25, %class.Buggy** %23, align 8
  %26 = load %class.Buggy*, %class.Buggy** %23
  %27 = getelementptr inbounds %class.Buggy, %class.Buggy* %26, i32 0, i32 2
  %28 = load void ()*, void ()** %27
```

```
  call void %28()
  br label %31

29:                                               ; preds = %8
  %30 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]* @tcb_notFound, i32 0, i32 0))
  br label %31

31:                                               ; preds = %29, %22
  %32 = load i8, i8* %9
  %33 = icmp eq i8 %32, 112
  br i1 %33, label %34, label %41

34:                                               ; preds = %31
  %35 = alloca %class.Buggy*
  %36 = bitcast %class.Foo* %18 to %class.Base*
  %37 = bitcast %class.Base* %36 to %class.Buggy*
  store %class.Buggy* %37, %class.Buggy** %35, align 8
  %38 = load %class.Buggy*, %class.Buggy** %35
  %39 = getelementptr inbounds %class.Buggy, %class.Buggy* %38, i32 0, i32 2
  %40 = load void ()*, void ()** %39
  call void %40()
  br label %43

41:                                               ; preds = %31
  %42 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]* @tcb_notFound, i32 0, i32 0))
  br label %43

43:                                               ; preds = %41, %34
  %44 = load i8, i8* %10
  %45 = icmp eq i8 %44, 0
  br i1 %45, label %46, label %53

46:                                               ; preds = %43
  %47 = alloca %class.Buggy*
```

```
  %48 = bitcast %class.Foo* %18 to %class.Base*
  %49 = bitcast %class.Base* %48 to %class.Buggy*
  store %class.Buggy* %49, %class.Buggy** %47, align 8
  %50 = load %class.Buggy*, %class.Buggy** %47
  %51 = getelementptr inbounds %class.Buggy, %class.Buggy* %50, i32 0, i32 2
  %52 = load void ()*, void ()** %51
  call void %52()
  br label %55

53:                                               ; preds = %43
  %54 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]*
@tcb_notFound, i32 0, i32 0))
  br label %55

55:                                               ; preds = %53, %46
  %56 = load i16, i16* %11
  %57 = icmp eq i16 %56, 21
  br i1 %57, label %58, label %65

58:                                               ; preds = %55
  %59 = alloca %class.Buggy*
  %60 = bitcast %class.Foo* %18 to %class.Base*
  %61 = bitcast %class.Base* %60 to %class.Buggy*
  store %class.Buggy* %61, %class.Buggy** %59, align 8
  %62 = load %class.Buggy*, %class.Buggy** %59
  %63 = getelementptr inbounds %class.Buggy, %class.Buggy* %62, i32 0, i32 2
  %64 = load void ()*, void ()** %63
  call void %64()
  br label %67

65:                                               ; preds = %55
  %66 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]*
@tcb_notFound, i32 0, i32 0))
  br label %67

67:                                               ; preds = %65, %58
```

```
  %68 = load i32, i32* %12
  %69 = icmp eq i32 %68, 2597
  br i1 %69, label %70, label %77

70:                                               ; preds = %67
  %71 = alloca %class.Buggy*
  %72 = bitcast %class.Foo* %18 to %class.Base*
  %73 = bitcast %class.Base* %72 to %class.Buggy*
  store %class.Buggy* %73, %class.Buggy** %71, align 8
  %74 = load %class.Buggy*, %class.Buggy** %71
  %75 = getelementptr inbounds %class.Buggy, %class.Buggy* %74, i32 0, i32 2
  %76 = load void ()*, void ()** %75
  call void %76()
  br label %79

77:                                               ; preds = %67
  %78 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]*
@tcb_notFound, i32 0, i32 0))
  br label %79

79:                                               ; preds = %77, %70
  %80 = load i64, i64* %13
  %81 = icmp eq i64 %80, 1000000
  br i1 %81, label %82, label %89

82:                                               ; preds = %79
  %83 = alloca %class.Buggy*
  %84 = bitcast %class.Foo* %18 to %class.Base*
  %85 = bitcast %class.Base* %84 to %class.Buggy*
  store %class.Buggy* %85, %class.Buggy** %83, align 8
  %86 = load %class.Buggy*, %class.Buggy** %83
  %87 = getelementptr inbounds %class.Buggy, %class.Buggy* %86, i32 0, i32 2
  %88 = load void ()*, void ()** %87
  call void %88()
  br label %91
```

```
89:                                               ; preds = %79
  %90 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]* @tcb_notFound, i32 0, i32 0))
  br label %91

91:                                               ; preds = %89, %82
  %92 = load float, float* %14
  %93 = fcmp oeq float %92, 0x40159999A0000000
  br i1 %93, label %94, label %101

94:                                               ; preds = %91
  %95 = alloca %class.Buggy*
  %96 = bitcast %class.Foo* %18 to %class.Base*
  %97 = bitcast %class.Base* %96 to %class.Buggy*
  store %class.Buggy* %97, %class.Buggy** %95, align 8
  %98 = load %class.Buggy*, %class.Buggy** %95
  %99 = getelementptr inbounds %class.Buggy, %class.Buggy* %98, i32 0, i32 2
  %100 = load void ()*, void ()** %99
  call void %100()
  br label %103

101:                                              ; preds = %91
  %102 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]* @tcb_notFound, i32 0, i32 0))
  br label %103

103:                                              ; preds = %101, %94
  %104 = load float, float* %14
  %105 = fcmp oeq float %104, 1.050000e+01
  br i1 %105, label %106, label %113

106:                                              ; preds = %103
  %107 = alloca %class.Buggy*
  %108 = bitcast %class.Foo* %18 to %class.Base*
  %109 = bitcast %class.Base* %108 to %class.Buggy*
  store %class.Buggy* %109, %class.Buggy** %107, align 8
```

```
  %110 = load %class.Buggy*, %class.Buggy** %107
  %111 = getelementptr inbounds %class.Buggy, %class.Buggy* %110, i32 0, i32 2
  %112 = load void ()*, void ()** %111
  call void %112()
  br label %115

113:                                              ; preds = %103
  %114 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]* @tcb_notFound, i32 0, i32 0))
  br label %115

115:                                              ; preds = %113, %106
  %116 = load double, double* %15
  %117 = fcmp oeq double %116, 7.356000e+00
  br i1 %117, label %118, label %125

118:                                              ; preds = %115
  %119 = alloca %class.Buggy*
  %120 = bitcast %class.Foo* %18 to %class.Base*
  %121 = bitcast %class.Base* %120 to %class.Buggy*
  store %class.Buggy* %121, %class.Buggy** %119, align 8
  %122 = load %class.Buggy*, %class.Buggy** %119
  %123 = getelementptr inbounds %class.Buggy, %class.Buggy* %122, i32 0, i32 2
  %124 = load void ()*, void ()** %123
  call void %124()
  br label %127

125:                                              ; preds = %115
  %126 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]* @tcb_notFound, i32 0, i32 0))
  br label %127

127:                                              ; preds = %125, %118
  %128 = load i64, i64* %16
  %129 = icmp eq i64 %128, -559038737
  br i1 %129, label %130, label %137
```

```
130:                                              ; preds = %139, %127
  %131 = alloca %class.Buggy*
  %132 = bitcast %class.Foo* %18 to %class.Base*
  %133 = bitcast %class.Base* %132 to %class.Buggy*
  store %class.Buggy* %133, %class.Buggy** %131, align 8
  %134 = load %class.Buggy*, %class.Buggy** %131
  %135 = getelementptr inbounds %class.Buggy, %class.Buggy* %134, i32 0, i32 2
  %136 = load void ()*, void ()** %135
  call void %136()
  br label %143

137:                                              ; preds = %127
  %138 = icmp eq i64 %128, 3735928559
  br i1 %138, label %139, label %140

139:                                              ; preds = %137
  br label %130

140:                                              ; preds = %137
  %141 = call i32 (i8*, ...) @printf(i8* getelementptr inbounds ([32 x i8], [32 x i8]*
@tcb_notFound, i32 0, i32 0))
  br label %142

142:                                              ; preds = %140
  br label %143

143:                                              ; preds = %142, %130
  %144 = bitcast %class.Foo* %18 to %class.Base*
  store %class.Base* %144, %class.Base** %17, align 8
  %145 = load %class.Base*, %class.Base** %17, align 8
  call void @_ZN4Base5helloEv(%class.Base* %145)
  ret void
}
```

From the above we see that the function “example” takes as parameters a Character, a Boolean, a Short integer, an integer, a Long integer, a Float, a Double and an Unsigned Long integer type variables. We will use those variables to trigger our bugs. The first two bugs we insert are triggered from the variable saved at %9, which is the character we take as a parameter, the first bug is triggered when the value of this variable is equal to 97 (that is the character ‘a’) and the second bug when the variable is equal to 112 (that is the character ‘p’). After we trigger a type confusion bug with the boolean variable saved at %10 and is triggered when this variable is equal to 0, ie “False”. The next bug is triggered from the variable saved at %11 (variable of “Short” integer type) and when this value is equal to 21 we trigger another bug. After, we insert a bug that is triggered from the integer variable saved at %12 and is triggered when this variable is equal to 21 and the next variable is trigged from the “Long” integer variable type, saved in %13 and only when this variable equals to 1000000. Additionally, we insert two more type confusion bugs that are triggered from the float variable in %14, the first when the variables value is equal to 5.4 and the second when is equal to 10.5. Also we insert and a type confusion bug for the Double type variable %15 and is executed when its value is 7.356. Finally, we insert a type confusion bug for an “Long” integer type which value is negative, but because we cannot know if this variable is signed or unsigned we insert an if-else condition, that checks if the value of the variable saved at %16 equals to -559038737, if it is true then executes the bug which is at block 130, otherwise it checks if the value of this variable equals with 3735928559, which is the unsigned form of the number -559038737, if this second condition is true then we jump to block 130 and execute the bug, otherwise we print a message on terminal, that says that we did not execute the specific bug.