Refining Function Pointers’ Targets With Class Hierarchy Tree Reconstruction and Object-Flow Analysis

Final Year Project Plan

Binrui Dong
(3035534816)

Supervisor
Chenxiong Qian

September 29, 2021
Contents

1 Background .................................................. 3
   1.1 Control Flow Integrity .................................. 3
   1.2 CFG and Function Pointers .............................. 4
   1.3 Multi-Layer Type Analysis ............................... 5

2 Objective .................................................. 7

3 Methodology ............................................... 7
   3.1 LLVM .................................................... 7
   3.2 Class Hierarchy Tree Reconstruction .................... 9
   3.3 Object-Flow Analysis ................................... 10
   3.4 Evaluation ............................................... 11

4 Schedule .................................................. 12
1 Background

This section will first introduce the background of Control Flow Integrity technique in subsection 1.1, then discuss the complications brought by function pointers in subsection 1.2, and introduce the state-of-the-art.

1.1 Control Flow Integrity

To date, many large modern software are written in C/C++ programming languages, from operating system kernels such as Linux and FreeBSD, to user space applications, including databases (e.g. MySQL, MongoDB), web browsers (e.g. Firefox, Chromium) and video game engines (e.g. Unreal Engine). C/C++ allows programmers to directly manipulate memory, which helps in building high performance software programs, however, it also leaves door open for potential security vulnerabilities [2]. Attackers may make use of oversights in the program (such as lack of input validation, array bounds checking, etc.) to corrupt memory, and divert the control-flow of the program away from its original legitimate region to malicious code.

In response, Control-Flow Integrity (CFI) technique [1] attempts to safeguard a program by restricting the control-flow of the program to its valid space [7]. The technique is composed of two phases, an ahead-of-time analysis phase, and a run-time enforcement phase [2]. During analysis phase, static analysis is performed on source code or binary of the program to compute its Control-Flow Graph (CFG), which basically represents the set

- bool lt(int x, int y) {
    return x < y;
}
- bool gt(int x, int y) {
    return x > y;
}
- sort2(int a[], int b[], int len) {
    sort(a, len, lt);
    sort(b, len, gt);
}

Figure 1: A piece of C code on the left, and its Control Flow Graph [1].
of legitimate control-flow state transfers, such as function calls and returns. Then, during run-time enforcement phase, each control-flow state transfer is verified by checking if there is such an edge in CFG. If the transfer does not exist in the pre-computed CFG, then the control-flow integrity of the program must have been compromised, and actions can be taken as desired, for example, terminate the program execution, generate a memory dump for further investigation, send an alert to users and administrators, etc.

1.2 CFG and Function Pointers

```c
// Defines function pointer type
typedef int (*fptr_t)(int, int);

// Defines two functions of the type
int funcA(int a, int b) { return a + b; }
int funcB(int a, int b) { return a * b; }

// An indirect call happens here
int funcX(fptr_t p) { return p(2, 3); }

int main() { return funcX(&funcA); }
```

Figure 2: An example of function pointers and indirect function call

The construction of CFG is complicated by function pointers. For direct function calls like `memcpy(a, b, n);` and `fopen("file.txt", "r");`, it’s known for sure that a function call to `memcpy` and `fopen` will happen at these places respectively. However, in case of an indirect function call through function pointers, the target function is not immediately known to static analysis. Figure 2 illustrates an example of such scenario. In the body of function `funcX()`, static analysis cannot immediately determine which target function is `p` referring to.

But function pointers are extensively used in practical software development. For example, in Linux kernel, in order to support `open()` system call on different file-systems, the kernel may store function pointers to the underlying implementations of such functionality for `ext4`, `xfs`, `btrfs`, etc., and dynamically invoke the corresponding underlying function during run-time
depending on the file-system used [3]. Function pointers are even essential in C++ run-time polymorphism. Each class, if contains virtual methods, maintains a virtual function table that stores function pointers to the actual concrete function implementations. During run-time, when invoking a virtual method of a class, the virtual function table of the class instance is looked up and an indirect call through the consulted function pointer takes place.

Previous approaches to deal with function pointers in constructing CFG includes rough type matching [5]. In the example of figure 2, the type of function pointer $p$ is $\text{int}(\text{int}, \text{int})$, i.e., a function takes two $\text{int}$ parameters and returns an $\text{int}$ value. Then all functions with the same type are considered to be possible target functions. Hence, the indirect call $p(2, 3)$; may transfer to both $\text{funcA}$ and $\text{funcB}$, even though it can be seen that only $\text{funcA}$ will be called and $\text{funcB}$ will never be invoked.

As illustrated by the above example, this approach will inevitably yield a lot of false positive transfers, which in reality will never happen during a valid program execution. This negatively impacts the performance of CFI enforcement, since more edges need to be stored and to be checked. Moreover, this also weakens the effectiveness of CFI technique, as some invalid transfers could be regarded as valid and causing the malicious attack to remain unnoticed. A more precise approach to resolve target functions of function pointers is desirable.

### 1.3 Multi-Layer Type Analysis


The technique attempts to improve the accuracy of resolving indirect call target functions by looking into the multi-layer type hierarchy information of the chain of $\text{structs}$ or $\text{classs}$ in which the function pointer is stored.

In a simplified example illustrated in figure 3 for the indirect call $\text{c.a.p()}$ in $\text{main()}$ function, Multi-Layer Type Analysis will notice that the function pointer $p$ is stored in a $\text{struct}$ of type $A$, which is in turn stored in a parent
// Defines function pointer type
typedef int (*fptr_t)(int, int);

// Defines two functions of the type
int funcB(int a, int b) { return a + b; }
int funcC(int a, int b) { return a * b; }

// Defines three structs
struct A { fptr_t p; };
struct B { struct A a; };
struct C { struct A a; };

// Defines two instances of structs
struct B b = { .a = { .p = &funcB } };
struct C c = { .a = { .p = &funcC } };

int main() { return c.a.p(2, 3); }

Figure 3: Code example of Multi-Layer Type Analysis

struct of type C. Then the scope of possible target functions for this indirect call is limited to those functions whose address are stored to pointers with the same multi-layer type hierarchy information, that is, function pointers that are also stored in a struct of type A in a parent struct of type C. In the example in figure 3, the only function satisfying such condition is funcC, since its address is stored to a struct of type A in a parent struct of type C at line 15. Although function funcB also has the same function type (taking two int parameters and returns an int value), the address of funcB is never assigned to a struct of type A in a parent struct of type C.

This approach is based on the intuition that in practical software development, function pointers are often stored in multi-layer objects of objects [3], and this is indeed effective in field test. The authors evaluated their technique on three real-world system programs, Linux, FreeBSD and Firefox, and observed 86% to 98% reduction in the target functions reported for indirect calls [3].
2 Objective

This project aims to further refine the resolution of target functions of function pointers with object-flow analysis, which will be elaborated in the next section.

3 Methodology

This section will first introduce the technology stack the project will be built upon in subsection 3.1, then discuss the two methodologies used by the project, class hierarchy tree reconstruction, and object-flow analysis, in subsection 3.2 and subsection 3.3 respectively, and finally discuss evaluation of the work in subsection 3.4.

3.1 LLVM

LLVM is a compiler and toolchain technologies framework composed of a collection of projects, most notably are

- Clang, an optimizing C/C++/Objective-C compiler
- LLVM Core, a set of libraries offering architecture independent optimization and code generation support for different CPU architectures

who are designed to work closely with an architecture independent and programming language independent intermediate representation called LLVM IR, which is similar to assembly language, but is portable and preserves certain high-level language information, such as strong types and functions.

Figure 4 shows the process of compiling a program from source code (e.g. a piece of C/C++ or Rust source code) to a binary executable that is runnable on a target architecture (e.g. x86) in LLVM. The compiler, Clang for C/C++ or rustc for Rust programs, will take source code as input, and generate LLVM IR stored in LLVM bitcode (.bc) file format. LLVM backend will then generate assembly code for target architecture from the LLVM bitcode.
double square(double num) {
    return num * num;
}

define dso_local double @_Z6squared(double %0)
    local_unnamed_addr #0 {
        %2 = fmul double %0, %0
        ret double %2
    }

square:
    sub esp, 12
    movsd xmm0, qword ptr [esp + 16]
    mulsd xmm0, xmm0
    movsd qword ptr [esp], xmm0
    fld qword ptr [esp]
    add esp, 12
    ret

Figure 4: LLVM Compiling Pipeline

Figure 5: C++ code, its LLVM IR, and its assembly code
Figure 5 illustrates an example of a C++ function being compiled to LLVM IR and x86 assembly code. Notice that function body, function name (_Z6squared instead of square is because of C++ name mangling to support function overload) and both function types and variable types are fully preserved in LLVM IR. In x86 assembly code, however, these information are mostly lost.

The project will be implemented as a LLVM pass. A LLVM pass is a program that takes LLVM IR as input, and can perform analysis, optimizations and transformations on the LLVM IR code read [10]. In the scope of this project, the LLVM pass to be built will only need to perform read-only analysis on LLVM IR.

The software project to be analyzed will first need to be compiled to LLVM bitcode files. For example, a C++ source file main.cpp can be compiled to LLVM bitcode by clang++ main.cpp -c -emit-llvm -o main.bc. Then the LLVM pass will read those compiled LLVM bitcode files to performs function pointer targets analysis on those LLVM IR code.

Since LLVM IR preserves types and functions from program source code, this approach offers more information available to the analysis, in comparison to analyzing directly on the executable binary in target architecture machine code (e.g. x86).

3.2 Class Hierarchy Tree Reconstruction

Class hierarchy tree reconstruction refers to recovering information of C++ class inheritance information from LLVM IR, in order to refine the scope of possible target functions of a class method invocation.

In figure 6 the superclass Animal has a method Name. Class Pet inherits from class Animal and overrides method Name. Class Dog inherits from class Pet and overrides method Name again.

When analyzing the function body of funcA() in figure 7 with class hierarchy tree information, it can be deduced that the possible target functions of object->Name() can only be either Pet::Name() or Dog::Name(), but not Animal::Name(), because object is of type Pet, and Animal is neither the class Pet itself nor its subclass, so Animal::Name() must not be a target function.
class Animal {
  public:
    virtual std::string Name() { return "Animal"; }
};

class Pet : public Animal {
  public:
    std::string Name() override { return "Pet"; }
};

class Dog : public Pet {
  public:
    std::string Name() override { return "Dog"; }
};

Figure 6: An example C++ class hierarchy

void funcA(Pet *object) {
  std::cout << object->Name() << std::endl;
}

void funcB() {
  Pet *object = new Dog;
  funcA(object);
  delete object;
}

Figure 7: Calling class methods

3.3 Object-Flow Analysis

Class hierarchy tree reconstruction can eliminate overridden superclass methods in target functions resolution, but it can be made more accurate, by analyzing not only the semantic type of an object as written in source code, but also what concrete types the instance could possibly take.

In the example in figure 6 and figure 7, funcA() is only called in funcB(). In funcB(), although pointer object is declared with type of Pet, it is sub-
sequently assigned with a Dog instance at line 6, and is passed as parameter to funcA(). In this example, the only type object can actually take in funcA() is Dog, and further refine the target functions set of method call object->Name() at line 2 in funcA() to only contain Dog::Name().

Object-flow analysis keeps track of the set of possible concrete types that an object may take, in order to realize the intuition described above.

A possible approach to implement object-flow analysis is iterative search. Starting with an initial set of possible concrete types an object can take, it follows the logic flow of the program and detects if there is any value assignments to the object. There could be an instance of another concrete type assigned to the object. If this happens, then add the new concrete type to the set. Repeat the process, until the set is stabilized, i.e., no more type can be added to the set.

When analyzing a large code base, however, running time and memory consumption of the above method could be a concern. A big C++ project may contains thousands of compilation units and hundreds of thousands of functions and variables. It is hardly feasible to fit everything in memory during analysis. Some optimizations and trade-offs may need to be made, in order to make the object-flow analysis practical in real-world use scenario.

3.4 Evaluation

After the LLVM pass is implemented, the analysis will be performed on real-world large scale C++ projects, such as Chromium, an open source web browser engine with around 12 million lines of C++ code as analyzed around August 2019 [6].

The performance of the LLVM pass developed in the project, as well as the target functions resolution results produced, will be analyzed and compared to other previous approaches in resolution of target functions of function pointers, such as type matching [5] and multi-layer type analysis [3] described in subsection 1.3.
4 Schedule

<table>
<thead>
<tr>
<th>Time Point</th>
<th>Task</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>September 2021</td>
<td>Submission of first phase deliverable:</td>
<td>Done</td>
</tr>
<tr>
<td></td>
<td>• Project Plan</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Project Web Page set-up</td>
<td></td>
</tr>
<tr>
<td>October 2021</td>
<td>Research on previous work</td>
<td>Working</td>
</tr>
<tr>
<td>November 2021</td>
<td>Preliminary system design</td>
<td>Pending</td>
</tr>
<tr>
<td>January 2022</td>
<td>Submission of Intermediate Report</td>
<td>Pending</td>
</tr>
<tr>
<td>February 2022</td>
<td>System implementation</td>
<td>Pending</td>
</tr>
<tr>
<td>March 2022</td>
<td>Performance evaluation</td>
<td>Pending</td>
</tr>
<tr>
<td>April 2022</td>
<td>Submission of Final Report</td>
<td>Pending</td>
</tr>
</tbody>
</table>

Table 1: Project Schedule
References


