BinTyper: Type Confusion Detection for C++ Binaries

Dongjoo Kim
CIST, School of Cybersecurity, Korea University
dongjukim@korea.ac.kr

Seungjoo Kim *
CIST, School of Cybersecurity, Korea University
skim71@korea.ac.kr

Abstract

Type confusion bug (or bad casting) is a popular vulnerability class to attack C++ software like the web browser, document reader. This bug occurs once a program typecasts and uses an object as an incompatible type. An attacker could exploit this vulnerability to execute malicious code in the target software.

Previous researches to detect type confusion bugs have been performed at the source-level. It inserts codes that verify type compatibility in the typecasting operator to perform detection at runtime. These approaches cannot be applied in binary-level, because high-level information such as class hierarchy and the typecasting operator does not exist in the compiled binary. However, many popular softwares such as Adobe Reader, Microsoft office are provided with binary only.

In this paper, we propose BinTyper, a type confusion detection tool that can be used in binary-level. BinTyper splits the internal layout of classes into multiple *area*s via static analysis. After that BinTyper recovers the minimum type information required for the binary to be executed without triggering the type confusion bug via dynamic analysis. Based on this information, the target binary can be executed with the verification to detect the type confusion bug.

Keywords - Type confusion; Bad casting; Type safety; Typecasting; Black-box testing; Binary analysis

*Corresponding author

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00532,Development of High-Assurance(>=EAL6) Secure Microkernel)

1 Introduction

Object-oriented programming is a paradigm for expressing a program through a set of objects and interactions between objects. Object-oriented programming is used for complex and large-scale software development because it is easy to maintain and reusable. Among the object-oriented programming languages such as C++, JAVA, and Python, many of the performance-critical software such as Chrome and Firefox use C++ for development.

One of the important features to support polymorphism of object-oriented languages is typecasting between objects. By typecasting, the same object can be converted from the original type to the target type. This allows developers to write code concisely and intuitively by expressing various derived classes as a common parent class. In this case, if type-specific work is needed for a specific derived class, typecasting(downcasting) from the parent class to the derived class can be performed.

Upcasting works safely because the derived class implies the parent class. However, when downcasting a parent class to a derived class, it is not known whether the target object is actually typecastable to a derived class(so-called typecasting compatibility). If an object is typecasted to an incompatible class type, the program treats the object as an incorrect type (so-called type confusion bug or bad casting). The result is unintended behavior, and an attacker can abuse it to develop an exploit.

C++ provides *dynamic_cast* as a typecasting operator that verifies compatibility at runtime. However, *dynamic_cast* is slower than *static_cast*, a typecasting operator that verifies compatibility only at compile time (more than 10 times slower in our experiment). For this reason, performance-critical software such as OS and Web browser perform typecasting through *static_cast*. Therefore it makes a type confusion bug may occur since *static_cast* does not verify typecasting compatibility at runtime. The following examples show examples

of Type confusion bugs found in popular software such as web browsers: ChakraCore (CVE-2020-1219 [6]), Adobe Reader (CVE-2019-8221 [5]), Vbscript (CVE-2017-8618 [4])

Previous researches for runtime type confusion bug detection have been performed at the source code level. These researches detect type confusion bugs occurring at runtime by inserting code that verifies typecasting compatibility at the point where the typecasting operator is used in the process of compiling C++ source code. Google's UBSan [18] replaced *static_cast* with *dynamic_cast* to verify typecasting compatibility based on RTTI. CaVer [31], TypeSan [26], and HexType [28] verify typecasting compatibility based on the Custom Type metadata structure. However, the utility of these researches is limited to White-box testing. It is difficult to apply the above researches[31][26][28] to Blackbox-testing because Blackbox-testing is usually performed without the source code and only the binary is given.

Previous researches focusing on Black-box testing (GFlags [10], Application Verifier [1], Electric Fence [9], RetroWrite [21], Valgrind [19], DrMemory [7]) are used for memory corruption bug detection like object lifetime issue(Use-after-free) and boundary issue (Buffer overflow, Out-of-bound access). Since these researches are not designed to detect type confusion bugs, they have a limitation that they can detect type confusion bugs in specific situations only (e.g. access beyond type-confused object boundary).

In this paper, we present BinTyper, a runtime type confusion tool that can be applied to C++ binaries. This tool performs static analysis to analyze the class hierarchy and the layout inside the class. And with dynamic analysis, it identifies the information of the target object to correctly execute assembly instructions that interact with the object without causing a type confusion bug. After that, it executes the target binary based on the identified information and detects the runtime type confusion bug.

In summary, we make the following contributions for type confusion bug detection:

- We present BinTyper, a tool that detects runtime type confusion bugs against C++ binaries. To the best of our knowledge, BinTyper is the first study to detect a type confusion bug at the binary-level.
- We organize key challenges on detection type confusion bug binary-level.
- We propose Area-based Type confusion detection, a method that can be applied for binary type confusion bug detection.
- We present a method to analyze the condition of the object required by the instruction for the correct execution that interacts with that object.

2 Background

2.1 Type system of C++

This section provides background information on the C++ language type system and the type confusion bug, which is a bug caused by the characteristics of the type system.

2.1.1 C++ class

C++ is an object-oriented language and has a class concept. The class supports member variables and member functions as user-defined types. Developers can create objects based on the class, access member variables of each created object, and call its methods. Classes can be constructed by inheriting other classes. The inherited class is called the child class or the derived class, and the classes that are the target of inheritance are called the parent classes. Child class has the characteristics of inherited parent class: It holds member variables of parent class and methods of the parent class. The child class can define additional member variables and methods of its own, which do not exist in the parent class. Since the child class includes the parent class, an object of the child class can be stored in a variable of the parent class type. In this case, when an operation on the parent class type (member variable access, method call, etc.) occurs, access is performed to the area of the parent class inside of the child class. Also, C++ supports polymorphism using the concept of virtual functions. Even if a method of the same name is called, the method actually called depends on the type of the object actually stored in the variable.

2.1.2 Typecast and Type confusion bug

The C++ language supports typecast operations to convert the type of variables. There are four types of typecast operations: *reinterpret_cast*, *static_cast*, *dynamic_cast*, and *C-style cast*. *reinterpret_cast* is used in the form of *reinterpret_cast<destination_type> (source_variable)*. This typecast operation does not verify whether conversion between *source_variable type* and *destination_type* is possible(typecasting compatibility). *static_cast* is used in the form of *static_cast<destination_type> (source_variable)*. *static_cast* verifies *source_variable type* and *destination_type* compatibility at compile time. The *static_cast* checks whether the type of the *source_variable type* and the *destination_type* are compatible with the inheritance relationship through the class hierarchy. *dynamic_cast* is used in the form of *dynamic_cast<destination_type> (source_variable)*. *dynamic_cast* is a typecast operation that verifies compatibility at runtime. At the time of typecasting, the type conversion compatibility

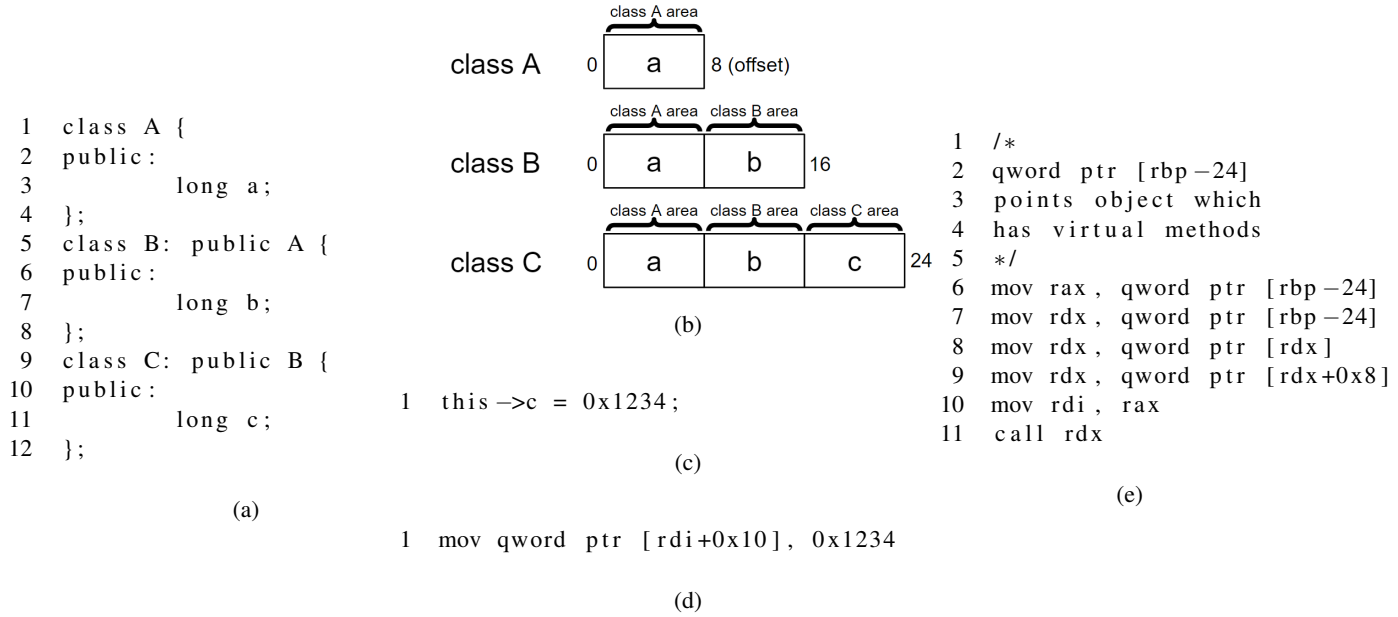


Figure 1: Class layout and its assembly representation. (a) Sample of C++ classes. (b) Class layout. (c) C++ code accessing member varaible. (d) Assembly representation of member variable access. (e) Assembly representation of virtual function call.

between the actual type of the object stored in the *source_variable* and the *destination_type* is verified. To verify this type conversion compatibility, *dynamic_cast* searches RTTI information of an actual object stored in *source_variable*. *reinterpret_cast* and *static_cast* typecast operations exist only in the source code and these are used to verify type compatibility at compile time, and the typecast operator information is not left in the compiled binary. On the other hand, when *dynamic_cast* typecast operation is used, additional code for performing type verification at runtime is inserted into the binary, so information about the typecast operation remains in the compiled binary. C-style cast is used in the form of *(destination_type)source_variable*. When C-style cast is used, the compiler tries typecast in the order of *const_cast* (typecast operation to remove const of variables), *static_cast*, and *reinterpret_cast* and uses the first successful typecast. That is, *C-style cast* is the same as one of *static_cast* and *reinterpret_cast*.

The type confusion bug occurs when a variable is typecasted to an incompatible destination type. Typecast operations, except *dynamic_cast*, verify type compatibility only at compile time. Therefore, if a source variable that is not compatible with the destination type is provided at runtime, the compatibility of the type for type conversion cannot be verified. As a result, the program is executed with the variable being considered as the wrong type, leading to unintended behavior. Although it is possible to prevent incorrect type conversion by using *dynamic_cast* operation that verifies type compatibility at runtime, *dynamic_cast* searches for RTTI to verify type conversion compatibility, which leads to a decrease in program performance. Therefore, large-scale software [16][11] does not use *dynamic_cast* for most type conversions. As a result, the compatibility of typecasting can be verified only at compile time and there is a possibility of a type confusion bug.

2.2 Assembly representation of C++ codes

At the binary level, C++ source code is converted into assembly instructions. This section describes how class-related C++ source code is expressed in assembly instructions. This section is intended for compilers using the Itanium C++ ABI [14].

2.2.1 Class Layout and Inheritance

Figure 1(a) and (b) show the C++ class source code sample and the layout of the class. To create objects, the compiler allocates memory. The size of the allocated memory is determined by the size of member variables defined in the class, and the object is placed in the allocated memory. Figure 1(a) shows class definition and memory layout of created object. The member variables of the class are located in order from the object's starting point (this). If the class has a virtual method (virtual function), the pointer to the VTable is located at

the initial starting point of the object, and the member variables are located immediately afterward (the pointer to the VTable becomes the first member variable). The VTable will be explained further later.

When a class inherits a parent class, the child class can not only have the member variables and methods of the parent class but also additionally have its own member variables and methods. The class layout of child classes is organized as shown in Figure 1(b). The class layout of the child class consists of the member variables of the child class located after the class layout of the parent class.

2.2.2 Class member variables and methods

Class member variables are located consecutively from the beginning of the object allocated in memory. The access to the member variable of the object is made by accessing the address of the object starting address (*this* pointer) plus a specific offset (determined according to the member variable). Figure 1(c) and (d) show C++ sample code and assembly expression to access member variable of an object. The member variable *c* is located at offset 0x10 and the rdi register points to this pointer. Therefore, the code to assign 0x1234 to member variable c is expressed as Figure 1(d).

2.2.3 Class constructor and destructor

Class constructors and destructors are special methods called when an object is created or destroyed. The child classes that inherit the parent class has the following characteristics: (1) Call the constructor of the parent class before executing the code of its constructor; (2) Call the destructor of the parent class after executing the code of its destructor do. The constructor and destructor of the child class call the constructor and destructor of the parent class with their *this* pointer as the *this* pointer of the method call. Since this pointer passed to the constructor and destructor of the parent class is the same as this pointer passed to the constructor and destructor of the child class, the same object can be initialized or cleaned up.

2.2.4 Virtual function table and Virtual function call

C++ uses the concept of virtual functions to implement polymorphism. A class having virtual methods (polymorphic class) has a data structure called VTable for each class. VTable stores virtual method addresses. VTable is stored in the read-only section of the binary file. In the constructor of the polymorphic class, the address of the VTable is stored as the first member variable at the starting point of the object. The virtual method call is made through the following process as shown in Figure 1(e): (1) The first member variable of the object is read and the address of the VTable is obtained; (2) In VTable, the offset to the corresponding virtual method is added to obtain the address of the actual virtual method function; (3) The obtained virtual method function is called. Therefore, the virtual method to be called may differ depending on the object actually provided at runtime.

3 Solution Overview

3.1 Threat model and Assumptions

BinTyper detects a type confusion bug at the point where the target application accesses a member variable of a polymorphic object. The target application has a type confusion bug and it is triggered once a malformed input is given.

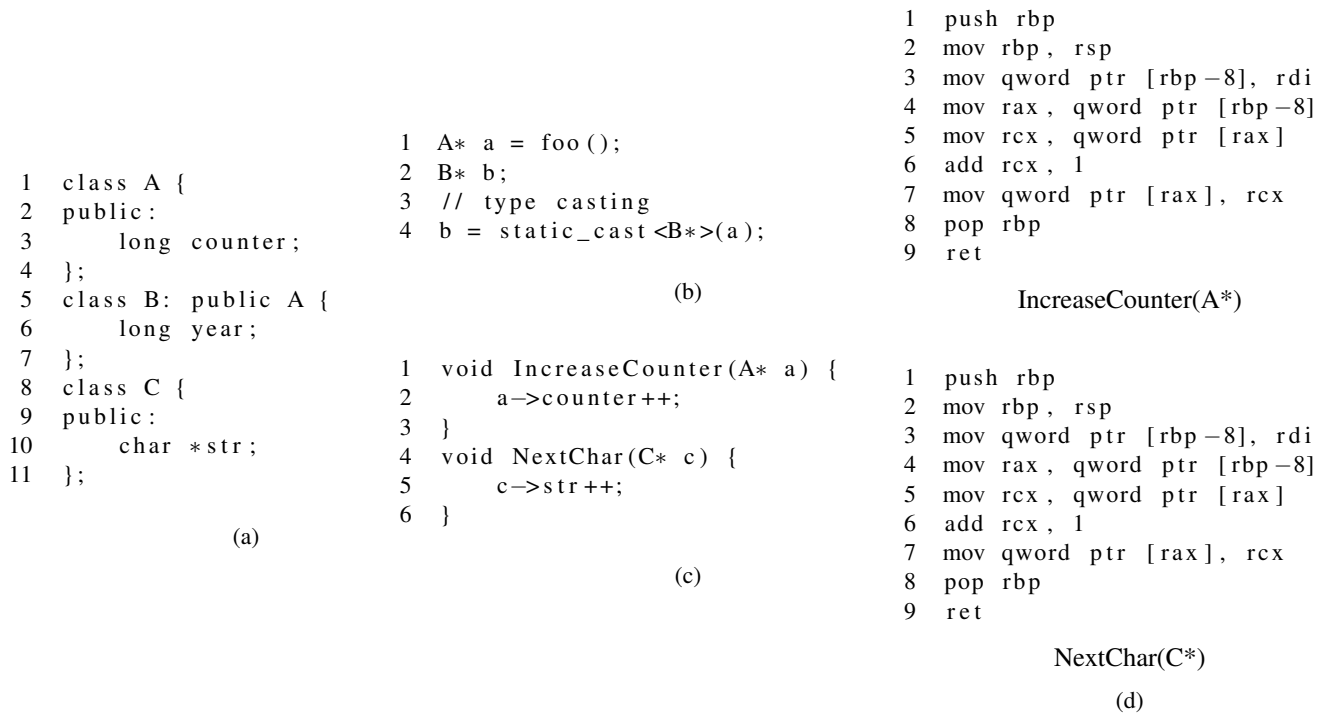
We assume that no source code is given since the RTTI information doesn't exist in every binary. We target binaries compiled based on Itanium C++ ABI [14]. Various previous researches have been performed on Itanium C++ ABI, which is used by major Linux C++ compilers such as GCC and Clang/LLVM.

3.2 Key Challenges

we must solve key challenges in detecting type confusion bugs in binaries since there is no high-level information such as source code. We listed the challenges below and Figure 2 shows examples of the challenges.

C1: Vanished Type Casting Operators Figure 2(b) shows an example of downcasting. The typecasting on Line 4 is downcasting. Therefore, when the actual object pointed to by variable *a* is not class *B* and derived class of *B*, a type confusion bug will occur. Previous source-level researches for detecting type confusion bugs [31][26][28] attempted to detect type confusion bugs by adding verification code to typecasting operators. The inserted code checks whether the actual type of the source object can be typecasted to the destination type. However, the typecasting operators except *dynamic_cast* exist only in the source code. As a result, there is no typecasting operator in compiled C++ binaries, which makes it difficult to determine when to perform detection of type confusion errors.

C2: Missing Class Information Figure 2(b) shows typecasting from class *A* to class *B* on line 4. In order to check the safety of this typecasting, inheritance relationship information(class hierarchy) between the actual object type and the destination type of typecasting



```
class A {
public:
    long counter;
};
class B: public A {
    long year;
};
class C {
public:
    char *str;
};
```

(a)

```
A* a = foo();
B* b;
// type casting
b = static_cast<B*>(a);
```

(b)

```
void IncreaseCounter(A* a) {
    a->counter++;
}
void NextChar(C* c) {
    c->str++;
}
```

(c)

```
push rbp
mov rbp, rsp
mov qword ptr [rbp-8], rdi
mov rax, qword ptr [rbp-8]
mov rcx, qword ptr [rax]
add rcx, 1
mov qword ptr [rax], rcx
pop rbp
ret
```

IncreaseCounter(A*)

```
push rbp
mov rbp, rsp
mov qword ptr [rbp-8], rdi
mov rax, qword ptr [rbp-8]
mov rcx, qword ptr [rax]
add rcx, 1
mov qword ptr [rax], rcx
pop rbp
ret
```

NextChar(C*)

(d)

Figure 2: Examples of key challenges. (a) Sample of C++ classes. (b) Type casting example. (c) Source code of sample functions. (d) Assembly code of sample functions, generated by x86-64 clang 9.0.0 compiler.

is needed. As described in the Background section, the C++ compiler removes high-level information during compilation. As a result, there is no class hierarchy information in the compiled C++ binary.

C3: Unknown Runtime Type Information Figure 2(c) shows the source code of two functions, *IncreaseCounter* and *NextChar*. Each function requires a different type of argument and accesses the member variable of each argument: (1) The *IncreaseCounter* function receives a class *A* argument and increments the value of the *int* type member variable named *counter* by 1; And (2) *NextChar* function receives class *C* argument and increases the value of *char** type member variable named *str* by 1. Figure 2(d) shows the assembly code generated from the source code of Figure 2(c) by the C++ compiler. The high-level information in the source code has been removed, resulting in *IncreaseCounter* and *NextChar* functions having the same assembly code, even though the function operates on different classes and different member variables. Due to this, it is difficult to determine the type of object required for benign execution of assembly code which do not trigger type confusion bug.

3.3 Our approach

Figure 3 shows an overview of our approach. BinTyper solves the aforementioned challenges by combining static and dynamic analysis. BinTyper executes binaries with detection of type confusion bugs targeting polymorphic classes. Key ideas of BinTyper is as follows.

Class information is recoverable During compilation, the high-level information existing in the source code including the class inheritance structure is removed. As a result, the compiled binary does not directly reveal class information or inheritance (parent-child) information written in the source code. Nevertheless, it is possible to identify this information indirectly from assembly expressions for implementing C++ class concepts such as constructor/destructor calls and virtual function tables. Previous researches [24][30][42][35][38][23] show that indirect information can be used to recover class information and class inheritance relationship information from binaries.

Assembly representation doesn't be shared Figure 2(c) and (d) show the source code of two functions and the assembly code generated from the source code. The same assembly representation is created although

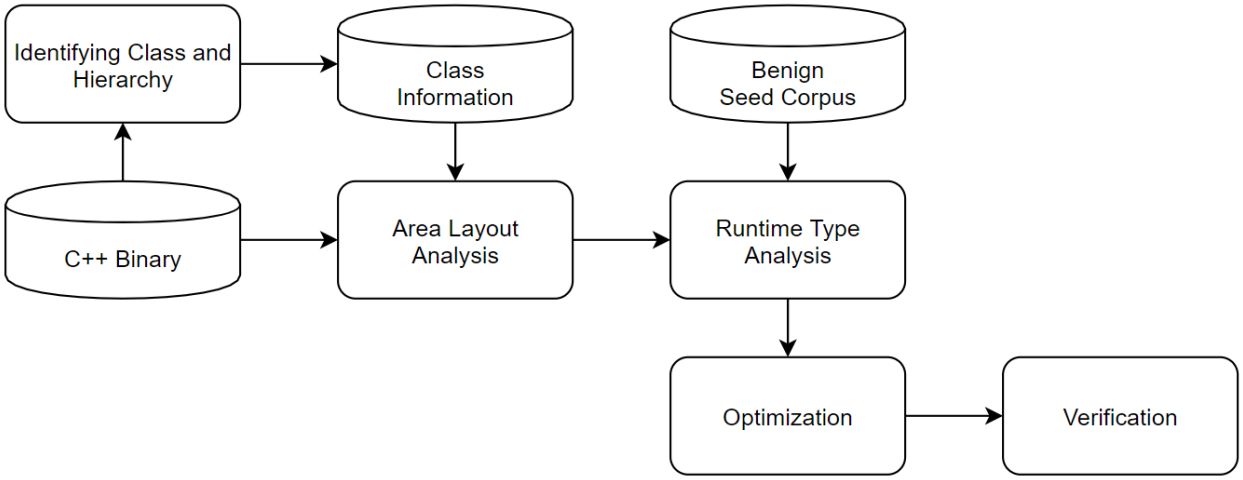


Figure 3: An overview of BinTyper

the two functions operate on different variable types. Although both functions are expressed in the same assembly code, the assembly expression corresponding to each function is created separately and the functions are called separately. That is, the variable type used by the assembly code corresponding to a single function is uniquely designated.

Class Object is composed of several areas Figure 1(b) shows the internal structure of the class object in memory. The class object is composed of the parent class area and the own class area, and the own class area is consecutively located after the parent class area. The parent class area means the space where member variables defined in the parent class are located, and the own class area means the space where member variables defined in the corresponding class are located. This area composition information (we will refer it as Area Layout Information) inside these class objects is accumulated when multiple inheritances are made. For example, suppose there are child classes B of class A and child class C of B as shown in Figure 1(a) and (b), and each class has one member variable named *a*, *b*, *c*. In this case, the area layout of class *C* is composed of class *A* area, class *B* area, and class *C* area. Member variable *a* is located in the class *A* area, member variable *b* is located in the class *B* area, and member variable *c* is located in the class *C* area. This Area Layout Information can be inferred through class hierarchy, class constructor, and member functions.

Type confusion bug is triggered once an area does not exist In Figure 4, (a)-(f) convert the argument type to *B** and call functions *func1* and *func2*. The *func1* function accesses the member variable *a* defined in class *B*'s parent class *A*. The *func2* function accesses the member variable *b* defined in class *B*, a derived class. The function calls in (c) and (d) do not cause any problems because the object with actual class type B is converted to class *B* type. In (a), the object of actual class type *A* is downcasted to class *B* type, but no problem actually occurs. This is because it accesses only the class *A* area of the object area passed as the *func1* function argument. Therefore, if a class *A* area exists in the object passed as an argument, it operates correctly without type confusion bug. In this example, the actual class type of the object passed as an argument is *A*, so there is no error because class *A* area exists. The function call in (e) also converts the object of actual class type *C* to class *B* type. It works without problems because class *A* area exists too. On the other hand, (b) and (f) are problematic typecasting. This is because the *func2* function requires a class *B* area, but neither class *A* nor class *C*, the actual class types of (b), and (f), have a class *B* area. Previous researches [31][26][28] detect type confusion bug based on the typecasting operator of the source code. However, since the typecasting operators are removed during compilation, this approach is difficult to apply at the binary-level. Instead, as another example shows, we can detect the type confusion bug regardless of C1 by checking whether the area which is needed for proper execution exists in the actual class type of the object. The existence of a specific area in an object indicates that the class type of the object is a class corresponding to the area or a child class of a class corresponding to the area. Therefore, if the area which is needed for proper execution does not exist in the actual class type of the object, it means that a type confusion bug has occurred. This is the key idea of BinTyper. In this paper, we will refer to the method which performs

```
1  class A {
2  public:
3    int var_a;
4  }
5
6  class B: public A {
7  public:
8    int var_b;
9  }
10
11 class C: public A {
12 public:
13   int var_c;
14 }
15
16 void func1(B* b) {
17   // Access area of class A
18   b->var_a = 0xaaaaaaaa;
19 }
20
21 void func2(B* b) {
22   // Access area of class B
23   b->var_b = 0xbbbbbbbb;
24 }
25
26 int main() {
27   A a;
28   B b;
29   C c;
30   func1((B*)&a); // (a) Non-problematic
31   func2((B*)&a); // (b) Problematic
32   func1((B*)&b); // (c) Non-problematic
33   func2((B*)&b); // (d) Non-problematic
34   func1((B*)&c); // (e) Non-problematic
35   func2((B*)&c); // (f) Problematic
36 }
```

Figure 4: Examples of key challenges

type confusion bug detection by checking the existence of the area as Area-based Type confusion bug detection.

BinTyper detects a type confusion bug at the time of execution of an assembly instruction accessing an object. Detection of the type confusion bug is performed by area-based type confusion bug detection that checks whether a specific area exists in the object accessed by the assembly instruction. Since it can be performed without a typecasting operator, it can be applied regardless of C1. Here, the 'Specific area' means the area to be accessed when the corresponding assembly instruction is executed normally without triggering a type confusion bug. This cannot be determined by analyzing single assembly instruction snippets statically. This is because the assembly code generated from each of the source codes using different types may be the same because the high-level information existing in the source code is removed during the compilation process (C3). To solve this, BinTyper applied both dynamic analysis and static analysis. In dynamic analysis, Runtime Access Information is logged while the target binary is normally executed without the occurrence of a type confusion bug. This includes the following information: the assembly instruction executed, the type information of the object accessed by the assembly instruction, and the offset from the starting address of the accessed object. After that, we analyze the Area Layout Information which is the area structure inside the class object. For this, class hierarchy information is required. Class inheritance and class hierarchy information do not exist in the binary (C2), but various researches to recover the inheritance hierarchy through static analysis from the compiled binary have been previously proposed and we can use this to recover the class hierarchy information. The area accessed by each instruction can be identified based on the Runtime Access Information and Area Layout information. After that, Area-based type confusion bug detection can be performed based on this information.

4 Design and Implementation

We developed BinTyper, a tool for detecting type confusion bugs in compiled C++ binaries. It consists of 5 steps as shown in Figure 3: *Identifying Class and Hierarchy*, *Area Layout Analysis*, *Runtime Area Analysis*, *Optimization*, and *Verification*.

Below we describe the details of each step.

4.1 Identifying Class and Hierarchy

The first step is to identify the Class and Class Hierarchy through static analysis. There are previous researches [24][30][42][35][38][23] for identifying a class and recovering hierarchy from a compiled binary. We can reconstruct the class hierarchy based on these researches. BinTyper operates on the polymorphic class, extracts the virtual function table, and uses it as a unique representation for each polymorphic class. After that, by performing static analysis, constructor-destructors are identified, and Overwrite analysis [35] is applied. Finally, the class hierarchy is recovered by inferring the inheritance relationship based on the results.

4.2 Area Layout Analysis

This step analyzes the area layout information of class objects based on the recovered class hierarchy. Area Layout Information is a set of position information of

```
function AnalyzeAreaInfo(target_cls):
  area_information = []
  parent_size, own_size = 0
  parent_cls = GetParentCls(target_cls)
  if parent_cls is not none:
    parent_size = GetClsSize(parent_cls)
    area_information.append(
      AnalyzeAreaInfo(parent_cls)
    )
  entire_size = GetClsSize(target_cls)
  own_size = entire_size - parent_size
  area_start_at = parent_size
  area_end_at = area_start_at + own_size
  area_information.append(
    [area_start_at, area_end_at]
  )
  return area_information
```

Figure 5: Algorithm to analyze area information

each area(start offset and end offset of the area) constituting a class object. We extended the Minimum Object Size Analysis of Declassifier [23] to analyze the area layout. Figure 5 shows the algorithm that analyzes Area Layout Information. In assembly code, access to member variables is represented by accessing the memory address which is calculated by adding the offset value corresponding to the target member variable to the object base address. Therefore, it is possible to infer the total size of the member variable area by statically analyzing accesses to the member variable memory of the object. From the class hierarchy, the parent class can be identified and the size of the parent area is calculated by the size of the member variable area of the parent class. The size of the own class area can be calculated by subtracting the size of the parent area from the total size of the member variable area of the derived class. For example, if we have a class of size 4 and a child class of this class of size 12, the area layout information of the child class is *{parent: {offset: 0, size:}, Own: {offset:4 , size:8}}*.

4.3 Runtime Area Analysis

To perform area-based type confusion bug detection, it is necessary to specify which area is to be checked. Even for the same assembly codes, the target area required for the normal execution of each code may be different (C3). Therefore, the target area can be correctly identified by accurately analyze the object type that can be used to the corresponding code. To analyze the object type to be used, dynamic analysis is performed in the Runtime Area Analysis step. It executes the target binary, and when the assembly instruction accesses memory, it determines whether it is accessing the object. If it is access to an object, it calculates the offset by calculating the difference between the accessed target address and the starting address of the object and identifies the target area by identifying which area is accessed based on the Area Layout Information. Codes in which the target area is identified become points where area-based type confusion bug detection will be performed in the later step.

4.4 Optimization

From the previous steps, We identified the points to perform Area-based Type confusion bug detection and the target area to check at each point. Before the corresponding point is executed, the type confusion bug can be detected by checking whether the required area exists in the type of the actual object that the instruction will access. When detection is applied, the execution speed of the instruction is slower than the normal execution. As a result, performing area-based type confusion bug detection at all points leads to a huge performance decrease due to frequent verification. Therefore, we reduced the point where detection is to be performed through the optimization step so that efficient verification work can be performed. The idea is as follows: If we have already confirmed the existence of a specific area for an object, we do not need to re-validate the existence of the same area for the same object. We identified the points where the duplicate verification could occur through static analysis and did not perform Area-based Type confusion bug detection at that point.

4.5 Verification

In the verification step, the target binary is executed and Area-based Type confusion bug detection is performed at the identified detection points. When the class constructor is called, BinTyper records the memory address and class type of the target object. When accessing the recorded object, check if the required area exists in the class type of the object where the access occurred, and if not, inform the type confusion bug has been detected.

4.6 Implementation

We implemented BinTyper for Linux x64 binary. We implemented steps that perform static analysis of BinTyper (Identifying Class and Hierarchy, Layout Analysis) using IDAPython [13] and Miasm [15]. The step for performing the dynamic analysis (Runtime Area Analysis, Verification) was implemented using Intel Pin [17].

	BinTyper	Vanilla	Overhead
Execution time	6.532s	0.071s	92x

Table 1: Runtime overhead in PDFium

RVA	Actual accessed area	Required area(s)
0x108be84	0x1337a38	0x1337468

Table 2: Detection result

5 Evaluation

We evaluated BinTyper with Google PDFium [12], the PDF generation and rendering library which is used in Google Chrome browser. BinTyper successfully detected crbug-1043508 [3], the type confusion bug which exists in PDFium library, without source code. We ran our evaluation on an Intel Core i7-6700k with 8GB RAM and Ubuntu 18.04. Our prototype only tracks objects of recovered classes that are allocated in heap-area.

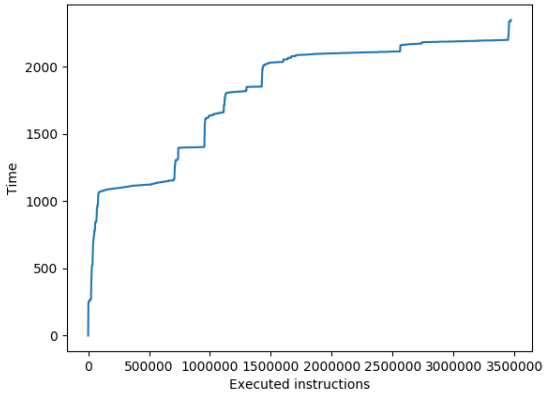


Figure 6: Performance

Runtime overhead Table [1] shows runtime overhead of *Verification step* in PDFium.

Performance Figure 6 shows elapsed time according to the number of instructions executed. It includes both the overhead of verification and the overhead of the DBI tool.

Count of tracked objects Figure 7 shows performance according to the number of tracked objects. Our prototype of BinTyper tracks objects allocated on heap memory area.

Information of detected type confusion bug BinTyper provides following information once type confusion bug is detected: *Fault instruction address(RVA)*, *Fault area* which actual object has, and *required area* to execute instruction without type confusion bug. Table 2 shows the these information.

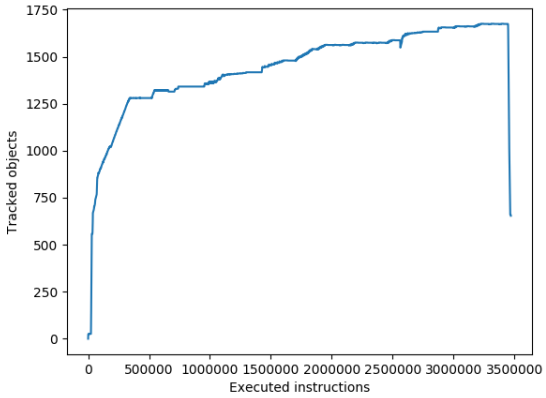


Figure 7: Count of tracked objects

6 Discussion

Object coverage The method for detecting the type confusion bug proposed by BinTyper can be performed for all classes including non-polymorphic classes. but our initial implementation limited the object to a polymorphic class because of the ease of static analysis. Since the polymorphic class has a virtual function table and a virtual method, it is easier to identify the class hierarchy and area layout information through static analysis than non-polymorphic classes. In future work, we will expand BinTyper to work with non-polymorphic objects.

Verification coverage As mentioned in Challenge C3, the compiled binary does not have the Runtime Type Information required for the code to run correctly without triggering a Type confusion bug. BinTyper performs dynamic analysis by executing target binaries with benign corpus to obtain Runtime Type Information and performs Area-based Type confusion bug detection. However, this has a coverage issue. Runtime type information can be obtained for codes executed in the dynamic analysis process only, so there is a limitation that verification cannot be performed for codes that have not been executed. Our future work will improve coverage by adding a Type propagation step that propagates the identified Runtime Type Information to unexecuted code.

Accuracy The accuracy of area-based type confusion bug detection performed by BinTyper is determined by the accuracy of class hierarchy and class information

identified through static analysis. Researches to recover class information from binaries is an area that is still active [24][30][42][35][38][23]. The accuracy can be improved by applying these researches to the BinTyper's static analysis step (Identifying Class and Hierarchy step and Area Layout Analysis step).

Performance Our prototype of BinTyper uses Intel Pin [17] for type confusion bug detection. It instruments the target binary with Intel Pin to track the assigned object and record the relevant information once the type confusion bug is detected. However, the current DBI-based implementation slows down the execution speed of the program and affects performance. Performance can be improved by performing a binary rewriting [8][21] that statically adds verification code to the binary or by performing verification in a low-overhead method [2] without DBI.

7 Related work

In this section, we discuss the work related to BinTyper.

Type confusion detection Previous researches performed detection by inserting a verification code that verifies the compatibility of the actual type of the source object to the destination type of typecasting by instrumenting the typecasting operator of the source code for runtime type confusion detection [31][26][28]. UBSAN [18] obtained object type information based on RTTI and performed verification. CaVer [31], TypeSan [26], and HexType [28] improved the RTTI-based research [18] by performing verification with the Custom Type metadata structure instead of RTTI. However, all of these researches have limitations that they cannot be applied for Black-box testing, which is performed on the binary-level.

Sanitizer Because C++ is not a memory-safe language, various researches have been proposed to detect memory safety violations in runtime. Dr.Memory [7] and memcheck of Valgrind [19] can detect memory error at binary-level with DBI(Dynamic binary instrumentation). ASan [39] and FuZZan [29] add code to detect memory error based on customized metadata structure in the process of compiling the source code. RetroWrite [21] tried to apply ASan [39] to binary through binary-rewriting. However, they are not designed to detect type safety violations like type confusion bug. These researches cannot detect a type confusion bug that occurs without a memory safety violation (e.g. size of the bad-casted target type can be smaller than the size of the source object's actual type).

Class hierarchy recovery High-level information is removed from the compiled C++ binary, so the information existing in the source code such as class name, class hierarchy, and class inheritance relationship does not appear directly. Previous researches [24][30][42][35][38][23] have been proposed to recover class information based on the indirect information remaining to implement the characteristics of OOP such as virtual function table and constructor/destructor. The Area Layout Analysis step of BinTyper is performed based on the recovered class hierarchy. Therefore, by combining these researches for BinTyper's class hierarchy recovery it is possible to increase the coverage and accuracy of BinTyper.

Virtual call integrity Previous researches to protect virtual function table hijacking attacks by protecting virtual calls at source-level and binary-level respectively have been proposed [20][27][25][36][43][41][37][44][22]. However, these researches are limited to protecting virtual calls, so only some type confusion bugs can be detected.

Data type reconstruction REWARDS [33], Howard [40], and TIE [32] infer and recover internal type information of binary. They focus on recovering primitive-level data types like *int* and *char**. dynStruct [34] and R. Rolles [2] recover structure-level data types from dynamic analysis. These researches can be used to improve the accuracy and coverage of class hierarchy recovery.

8 Conclusion

C++ programming language is used to develop widespread and performance-critical software such as web browsers. However, C++ is not type safety language. In C++, typecasting can be performed even though the source object is not compatible with the destination type. As a result, the object is treated as the wrong type, and undefined actions are performed and the attacker can exploit it.

We present BinTyper. Previous researches attempt to detect a type confusion bug at runtime based on the source code. This has a limitation that it is difficult to apply when Black-box testing, which only provides binary without source code. BinTyper performs static analysis and dynamic analysis on the target binary to analyze the type information necessary for the execution and performs type confusion detection at runtime. Our prototype results show that BinTyper can detect type confusion bugs in wide-spread and large-scale software.

References

[1] Application Verifier. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/application-verifier.

[2] Automation Techniques in C++ Reverse Engineering. https://cfp.recon.cx/reconmtl2019/talk/FGCZYU/.

[3] Chromium Issue 1043508. https://bugs.chromium.org/p/chromium/issues/detail?id=1043508.

[4] CVE-2017-8618. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-8618.

[5] CVE-2019-8221. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-8221.

[6] CVE-2020-1219. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-1219.

[7] Dr. Memory. https://drmemory.org/.

[8] Dyninst. https://www.dyninst.org.

[9] Electric Fence. https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/application-verifier.

[10] GFlags. https://docs.microsoft.com/en-us/windows-hardware/drivers/debugger/gflags.

[11] Google Chrome. https://www.google.com/chrome.

[12] Google PDFium. https://opensource.google/projects/pdfium.

[13] Hex-Rays IDA Disassembler. https://www.hex-rays.com/products/ida/.

[14] Itanium C++ ABI. https://refspecs.linuxbase.org/cxxabi-1.86.html.

[15] Miasm: Python reverse engineering framework. https://github.com/cea-sec/miasm.

[16] Mozilla Firefox. https://www.mozilla.org/en-US/firefox/products.

[17] Pin - A Dynamic Binary Instrumentation Tool. https://software.intel.com/content/www/us/en/develop/articles/pin-a-dynamic-binary-instrumentation-tool.html.

[18] UndefinedBehaviorSanitizer (UBSan). https://www.chromium.org/developers/testing/undefinedbehaviorsanitizer.

[19] Valgrind. https://valgrind.org/.

[20] DEWEY, D., AND GIFFIN, J. T. Static detection of c++ vtable escape vulnerabilities in binary code. In *NDSS* (2012).

[21] DINESH, S. *Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization*. PhD thesis, Purdue University Graduate School, 2019.

[22] ELSABAGH, M., FLECK, D., AND STAVROU, A. Strict virtual call integrity checking for c++ binaries. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* (2017), pp. 140–154.

[23] ERINFOLAMI, R. A., AND PRAKASH, A. Declassifier: Class-inheritance inference engine for optimized c++ binaries. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security* (New York, NY, USA, 2019), Asia CCS '19, Association for Computing Machinery, p. 28–40.

[24] FOKIN, A., TROSHINA, K., AND CHERNOV, A. Reconstruction of class hierarchies for decompilation of c++ programs. In *2010 14th European Conference on Software Maintenance and Reengineering* (2010), pp. 240–243.

[25] GAWLIK, R., AND HOLZ, T. Towards automated integrity protection of c++ virtual function tables in binary programs. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), pp. 396–405.

[26] HALLER, I., JEON, Y., PENG, H., PAYER, M., GIUFFRIDA, C., BOS, H., AND VAN DER KOUWE, E. TypeSan: Practical type confusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security* (oct 2016), vol. 24-28-October-2016, Association for Computing Machinery, pp. 517–528.

[27] JANG, D., TATLOCK, Z., AND LERNER, S. Safedispatch: Securing c++ virtual calls from memory corruption attacks. In *NDSS* (2014).

[28] JEON, Y., BISWAS, P., CARR, S., LEE, B., AND PAYER, M. HexType: Efficient detection of type confusion errors for C++. In *Proceedings of the ACM Conference on Computer and Communications Security* (oct 2017), Association for Computing Machinery, pp. 2373–2387.

[29] JEON, Y., HAN, W., BUROW, N., AND PAYER, M. Fuzzan: Efficient sanitizer metadata design for fuzzing.

[30] JIN, W., COHEN, C., GENNARI, J., HINES, C., CHAKI, S., GURFINKEL, A., HAVRILLA, J., AND NARASIMHAN, P. Recovering c++ objects from binaries using inter-procedural dataflow analysis. In *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014* (New York, NY, USA, 2014), PPREW'14, Association for Computing Machinery.

[31] LEE, B., SONG, C., KIM, T., AND LEE, W. Type casting verification: Stopping an emerging attack vector. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Berkeley, CA, USA, 2015), SEC'15, USENIX Association, pp. 81–96.

[32] LEE, J., AVGERINOS, T., AND BRUMLEY, D. Tie: Principled reverse engineering of types in binary programs.

[33] LIN, Z., ZHANG, X., AND XU, D. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium* (2010), pp. 1–1.

[34] MERCIER, D., CHAWDHARY, A., AND JONES, R. dynstruct: An automatic reverse engineering tool for structure recovery and memory use analysis. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (2017), IEEE, pp. 497–501.

[35] PAWLOWSKI, A., CONTAG, M., VAN DER VEEN, V., OUWEHAND, C., HOLZ, T., BOS, H., ATHANASOPOULOS, E., AND GIUFFRIDA, C. Marx: Uncovering class hierarchies in c++ programs. In *NDSS* (2017).

[36] PRAKASH, A., HU, X., AND YIN, H. vfguard: Strict protection for virtual function calls in cots c++ binaries. In *NDSS* (2015).

[37] SARBINOWSKI, P., KEMERLIS, V. P., GIUFFRIDA, C., AND ATHANASOPOULOS, E. Vtpin: practical vtable hijacking protection for binaries. In *Proceedings of the 32nd Annual Conference on Computer Security Applications* (2016), pp. 448–459.

[38] SCHWARTZ, E. J., COHEN, C. F., DUGGAN, M., GENNARI, J., HAVRILLA, J. S., AND HINES, C. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (2018), pp. 426–441.

[39] SEREBRYANY, K., BRUENING, D., POTAPENKO, A., AND VYUKOV, D. Addresssanitizer: A fast address sanity checker. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)* (2012), pp. 309–318.

[40] SLOWINSKA, A., STANCESCU, T., AND BOS, H. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS* (2011).

[41] VAN DER VEEN, V., GÖKTAS, E., CONTAG, M., PAWOLOSKI, A., CHEN, X., RAWAT, S., BOS, H., HOLZ, T., ATHANASOPOULOS, E., AND GIUFFRIDA, C. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *2016 IEEE Symposium on Security and Privacy (SP)* (2016), IEEE, pp. 934–953.

[42] YOO, K., AND BARUA, R. Recovery of object oriented features from c++ binaries. In *2014 21st Asia-Pacific Software Engineering Conference* (2014), vol. 1, pp. 231–238.

[43] ZHANG, C., SONG, C., CHEN, K. Z., CHEN, Z., AND SONG, D. Vtint: Defending virtual function tables' integrity. In *Symposium on Network and Distributed System Security (NDSS)* (2015), vol. 160, pp. 173–176.

[44] ZHANG, C., SONG, D., CARR, S. A., PAYER, M., LI, T., DING, Y., AND SONG, C. Vtrust: Regaining trust on virtual calls. In *NDSS* (2016).