black hat EUROPE 2020
DECEMBER 9–10
BRIEFINGS

# BinTyper: Type Confusion Detection for C++ Binaries

Dongju Kim @ School of Cybersecurity, Korea University
Seungjoo Kim* @ School of Cybersecurity, Korea University

* Corresponding author

# About us

- **Dongju Kim**
  - Graduate Student at SANE Lab., School of Cybersecurity, Korea University
  - Research Interests: Static Binary Analysis and Software Vulnerability Analysis

- **Seungjoo (Gabriel) Kim** (Corresponding Author)
  - Professor at School of Cybersecurity, Korea University
  - Member of Presidential Committee on the 4th Industrial Revolution
  - Black Hat Asia Review Board Member
  - Research Interests: Security·Privacy by Design, Threat-Risk Modeling, Formal Verification, Security Assessment & Authorization (such as Common Criteria, CMVP, SSE-CMM, RMF A&A), Blockchain & Crypto Engineering



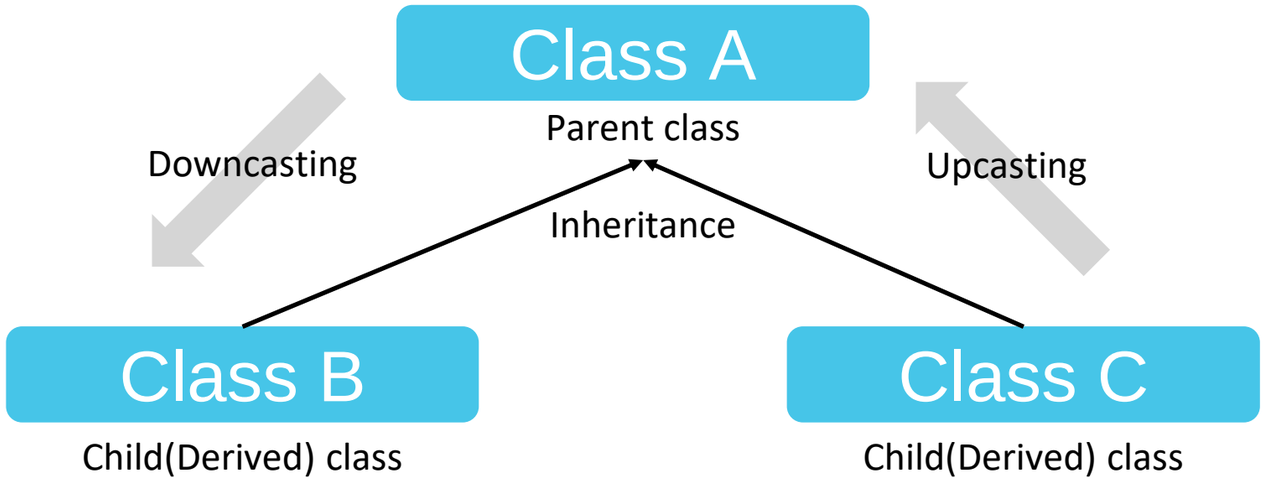
* This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government(MSIT) (No.2018-0-00532,Development of High-Assurance(≥EAL6) Secure Microkernel)

# Agenda

- Introduction & Motivation
- Source-level codes vs Binary-level codes
- Our goal
- Challenges & Ideas
- Our approach
- Demo
- Future work & Limitation
- Conclusion

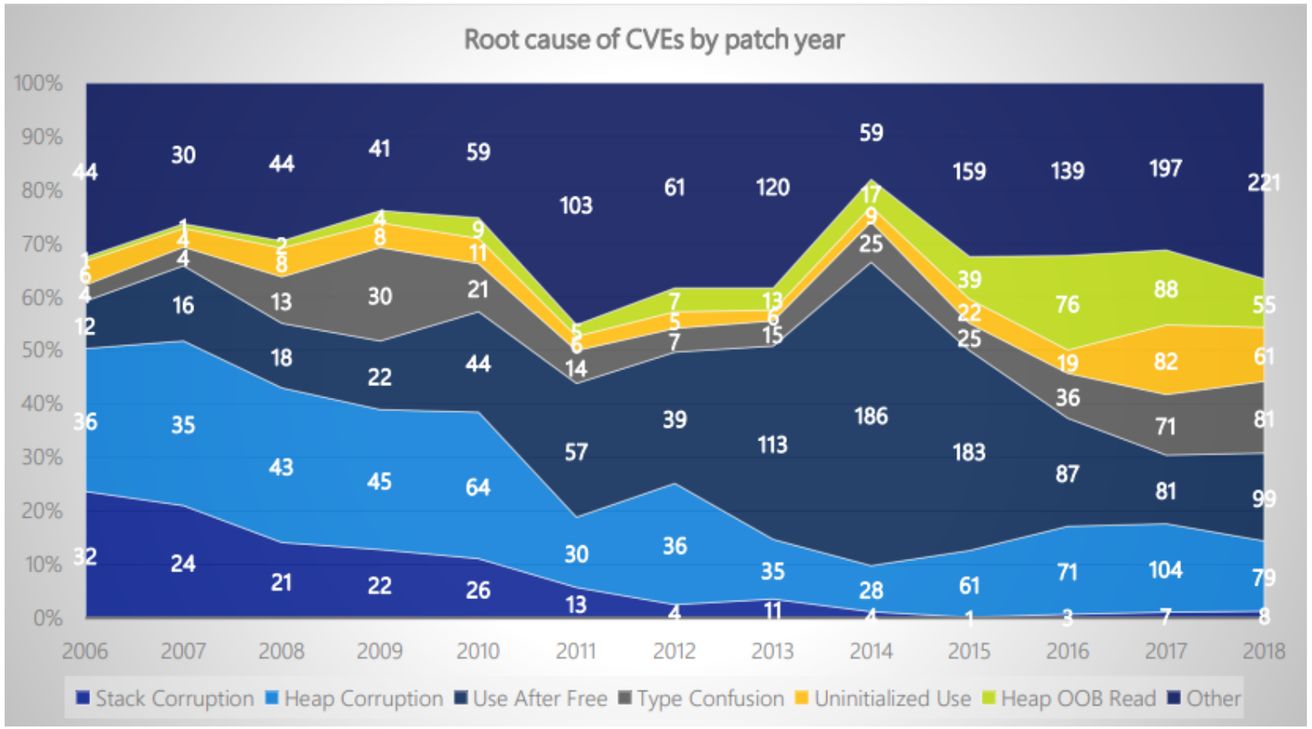
# Introduction & Motivation

- What is the type confusion bug?




```
17    B *b = new B;
18
19    // upcasting (safe)
20    A *a = static_cast<A*>(b);
21
22    // downcasting to incompatible type (unsafe)
23    C *c = static_cast<C*>(a);
24
25    // undefined-behavior
26    c->c = 0;
```

# Introduction & Motivation




*(Image reference: 'Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape' of BlueHat IL 2019)*

- Type confusion bug is one of the powerful bug types that is used for exploit development

# Introduction & Motivation

- Sanitizer supports:
  - Better detection of triggered software bug during runtime
  - Information about detected bugs

- Two areas of sanitizer: **source-level** and **binary-level**

- However, current binary-level sanitizers have not focused on detecting type confusion bugs

# Source-level codes vs Binary-level codes

- We'll talk about the difference between source-level codes and binary-level codes
- We'll cover the source-level codes first

# Source-level codes vs Binary-level codes

- High-level information is available
  - Class names, member variables, methods, hierarchy ...
  - Variable names, size, position ...
- Easy to edit code reliably

```
1   class ClassName {
2   public:
3       int member_name;
4       void hello() {};
5   };
6
7   int main() {
8       ClassName cls;
9       cls.hello();
10  }
```

```
7   int main() {
8       ClassName cls;
9
10      // ADDITIONAL CODE1
11      cls.hello();
12      // ADDITIONAL CODE1
13  }
```

# Source-level codes vs Binary-level codes

- Sanitizers ‘edit’ the codes to add a bug-checking routine
  - Also high-level information is available
- Bug-checking routine catches bugs more accurately and earlier
- Researches
  - *AddressSanitizer, ThreadSanitizer*
  - *UBSan, CAVER, TypeSan, HexType*
  - *...*

# Source-level codes vs Binary-level codes

- Sanitizers ‘edit’ the codes to add a bug-checking routine
  - Also high-level information is available
- Bug-checking routine catches bugs more accurately and earlier
- Researches
  - *AddressSanitizer, ThreadSanitizer*
  - *UBSan, CAVER, TypeSan, HexType*
  - *…*

Specially designed to detect type confusion bugs

# Source-level codes vs Binary-level codes

- UBSan
  - Detector for undefined behavior detector
  - Supports many checks
    - *-fsanitize=implicit-conversion*
    - *-fsanitize=integer*
    - *-fsanitize=vptr*
    - *…*
  - ***-fsanitize=vptr*** supports type confusion detection
    - Cannot handle non-polymorphic class types
    - Incompatible with *-fno-rtti*

```
streambuf_view.h:43:7: runtime error: downcast of address 0x7ffe4eeb26b8
 type '__gnu_cxx::stdio_filebuf<char, std::char_traits<char> >'
 00  00 08 0f 00 10 63 00 00  00 08 0f 00 10 63 00 00  00 08 0f 00
```

# Source-level codes vs Binary-level codes

- CAVER, TypeSan, HexType
  - Insert verification code around typecasting operation
  - Type compatibility verification
  - **'Actual object type' <-> 'Destination type of typecasting'**

**[Source code]**

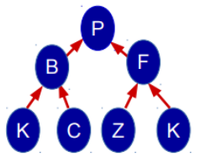
```
……..
B *pB = new B;
Update_objTypeMap(pB, B);

C *pC = static_cast<C*>(pB);
Verify_type_casting(pB, C);
```

| Object to Type Mapping Table | |
|---|---|
| 0x1232 | Type1 |
| 0x2312 | Type2 |
| 0x2333 | Type3 |

**[Runtime library]**

```
Verify_type_casting (Ptr *SrcPtr, TypeInfo Dst) {
  TypeInfo Src = getSrcType(SrcPtr);
  isTypeConfusionCast(Src, Dst);
}
```

**Type relation information**




9

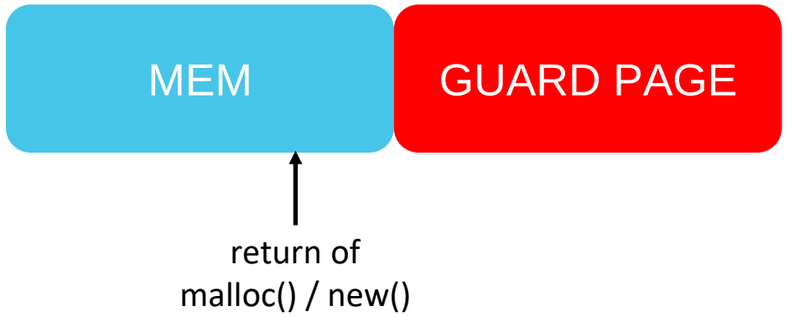
*(Image reference: presentation of HexType)*

# Source-level codes vs Binary-level codes

- High-level information is heavily used in source-level approaches
- How about binary-level?
  - There is no high-level information :(
  - Hard to modify the code
- Therefore, almost binary-level sanitizers do not rely on high-level information
  - It highly reduces the coverage/usability of sanitizers
  - Recent research, RetroWrite, try to infer and use high-level information

# Source-level codes vs Binary-level codes

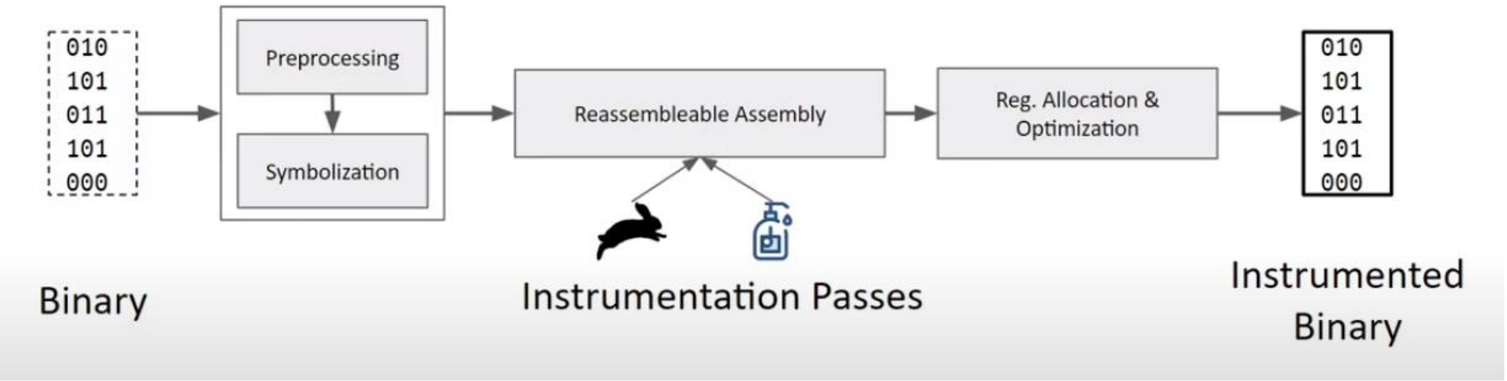
- Researches
  - **GFlags**
  - **ApplicationVerifier**
  - **Electric Fence**
  - RetroWrite
  - …




- General idea: Catch heap allocation/deallocation
- Detects heap out-of-bounds access and use-after-free

# Source-level codes vs Binary-level codes

- Researches
  - GFlags
  - ApplicationVerifier
  - Electric Fence
  - **RetroWrite**
  - …




*(Image reference: presentation of RetroWrite)*

- RetroWrite supports binary-level asan
- Reassembleable Assembly + Instrumentation

# Our goal

- Binary-level approach for detecting type confusion bugs
  - Detect type confusion bug at runtime
  - Provide information about the triggered bug
- We target Itanium C++ ABI
  - Generally used in Linux g++, clang++
- RTTI information may be helpful, but we do not use it

| Virtual function table | |
|---|---|
| -0x10 | Offset-to-top |
| -0x08 | Pointer to RTTI Information |
| 0x00 | Pointer to function 1 |
| 0x08 | Pointer to function 2 |
| 0x10 | Pointer to function 3 |

# Challenges

- Why we cannot easily detect type confusion bug at binary-level?
- Three major challenges
  - Vanished Type Casting Operators
  - Missing Class Information
  - Unknown Runtime Type Information

# Challenge 1 - Vanished Type Casting Operators

```
11   A *a = foo();
12   B *b;
13
14   // type casting
15   b = static_cast<B*>(a);
```

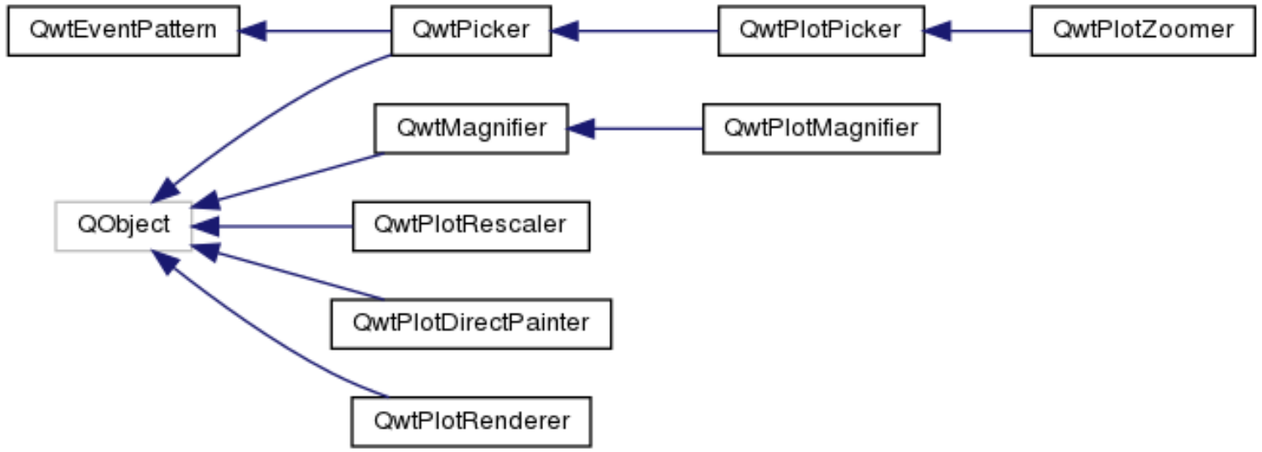
```
11          call    foo()
12          mov     qword ptr [rbp - 8], rax
13          mov     rax, qword ptr [rbp - 8]
14          mov     qword ptr [rbp - 16], rax
```

*Compiled by x86-64 clang 10.0.0*

- There is a typecasting operation in line 15
- But typecasting operation doesn't exist at binary
  - Except *dynamic_cast*<>
- Where should we perform type verification?

# Challenge 2 - Missing Class Information

- Verification code checks the type compatibility
  - ‘Actual object type’ <-> ‘Destination type of typecasting’
- For this, class relationship information(class hierarchy) is needed
  - it doesn’t exist on binary :(



# Challenge 3 - Unknown Runtime Type Information

```
1   class A {
2   public:
3       long counter;
4   };
5
6   class B: public A {
7   public:
8       long year;
9   };
10
11  class C {
12  public:
13      char *str;
14  };
```

```
15  void IncreaseCounter(A *a) {
16      a->counter++;
17  }
18
19  void NextChar(C* c) {
20      c->str++;
21  }
```

- What assembly codes are generated from *IncreaseCounter()* and *NextChar()?*
  - Are they the same or different?

# Challenge 3 - Unknown Runtime Type Information

```
1   class A {
2   public:
3       long counter;
4   };
5
6   class B: public A {
7   public:
8       long year;
9   };
10
11  class C {
12  public:
13      char *str;
14  };
```

```
15  void IncreaseCounter(A *a) {
16      a->counter++;
17  }
18
19  void NextChar(C* c) {
20      c->str++;
21  }
```

```
11  NextChar(C*):
12          push    rbp
13          mov     rbp, rsp
14          mov     qword ptr [rbp - 8], rdi
15          mov     rax, qword ptr [rbp - 8]
16          mov     rcx, qword ptr [rax]
17          add     rcx, 1
18          mov     qword ptr [rax], rcx
19          pop     rbp
20          ret
1   IncreaseCounter(A*):
2           push    rbp
3           mov     rbp, rsp
4           mov     qword ptr [rbp - 8], rdi
5           mov     rax, qword ptr [rbp - 8]
6           mov     rcx, qword ptr [rax]
7           add     rcx, 1
8           mov     qword ptr [rax], rcx
9           pop     rbp
10          ret
```

- What assembly codes are generated from *IncreaseCounter()* and *NextChar()*
  - => Both are same (Tested on clang++ x86-64 6 ~ 10, g++ 6 ~ 10)

# Challenge 3 - Unknown Runtime Type Information

- *IncreaseCounter()* and *NextChar()*
  - Different type of arguments
  - Different type of accessed member variable
  - … but produce the same result

- It means that we don’t know which object will be handled by specific assembly snippets in runtime

```
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
```

for A? or for B?

# Ideas

- Here are some ideas to overcome challenges and detect type confusion bugs
  - Class information is recoverable
  - Assembly representation doesn’t be shared
  - Class Object is composed of several areas
  - Type confusion bug is triggered once an area does not exist

# Idea 1 - Class information is recoverable

- Compiled C++ binaries have no high-level information
  - Class identifier(name), Class hierarchy ...
- But we can still infer class identifier and hierarchy
  - From indirect information
    - Constructor/Destructor chain, VFT …
- Previous researches
  - DeClassifier
  - MARX
  - VCI
  - OOAnalyzer
  - ...

```
12  B::B() [base object constructor]:
13          push    rbp
14          mov     rbp, rsp
15          sub     rsp, 16
16          mov     qword ptr [rbp - 8], rdi
17          mov     rax, qword ptr [rbp - 8]
18          mov     rcx, rax
19          mov     rdi, rcx
20          mov     qword ptr [rbp - 16], rax # 8-byte Spill
21          call    A::A() [base object constructor]
22          movabs  rax, offset vtable for B
23          add     rax, 16
24          mov     rcx, qword ptr [rbp - 16] # 8-byte Reload
25          mov     qword ptr [rcx], rax
```

# Idea 2 - Assembly representation doesn't be shared

- Different source codes don't share the same assembly code
  - ... even though both can be expressed in the same assembly code
- Certain assembly code will not operate on other types
  - It is not shared for several types

```
1   IncreaseCounter(A*):
2           push    rbp
3           mov     rbp, rsp
4           mov     qword ptr [rbp - 8], rdi
5           mov     rax, qword ptr [rbp - 8]
6           mov     rcx, qword ptr [rax]
7           add     rcx, 1
8           mov     qword ptr [rax], rcx
9           pop     rbp
10          ret
11  NextChar(C*):
12          push    rbp
13          mov     rbp, rsp
14          mov     qword ptr [rbp - 8], rdi
15          mov     rax, qword ptr [rbp - 8]
16          mov     rcx, qword ptr [rax]
17          add     rcx, 1
18          mov     qword ptr [rax], rcx
19          pop     rbp
20          ret
```

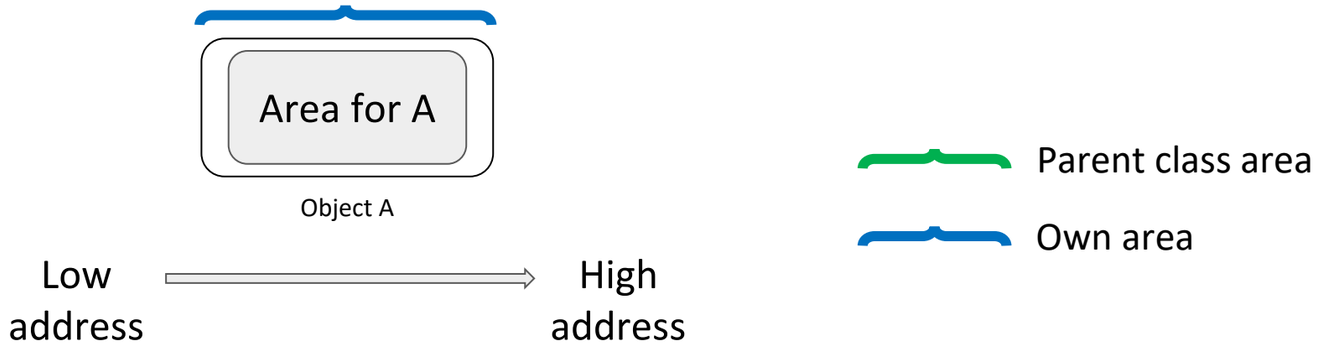
# Idea 3 - Class Object is composed of several areas

- Derived(Child) class inherit member fields from their parents
  - Parent classes inherit member fields from their parents too
- We'll use the ***'area'*** term to indicate *unique member fields of each class*

- Parent class area
  - Member fields which derived class inherits
- Own area
  - Member fields which derived class defines itself (doesn't exist in parents)
- Own area is located after the Parent class area
  - In Itanium ABI, but other ABIs may be the same

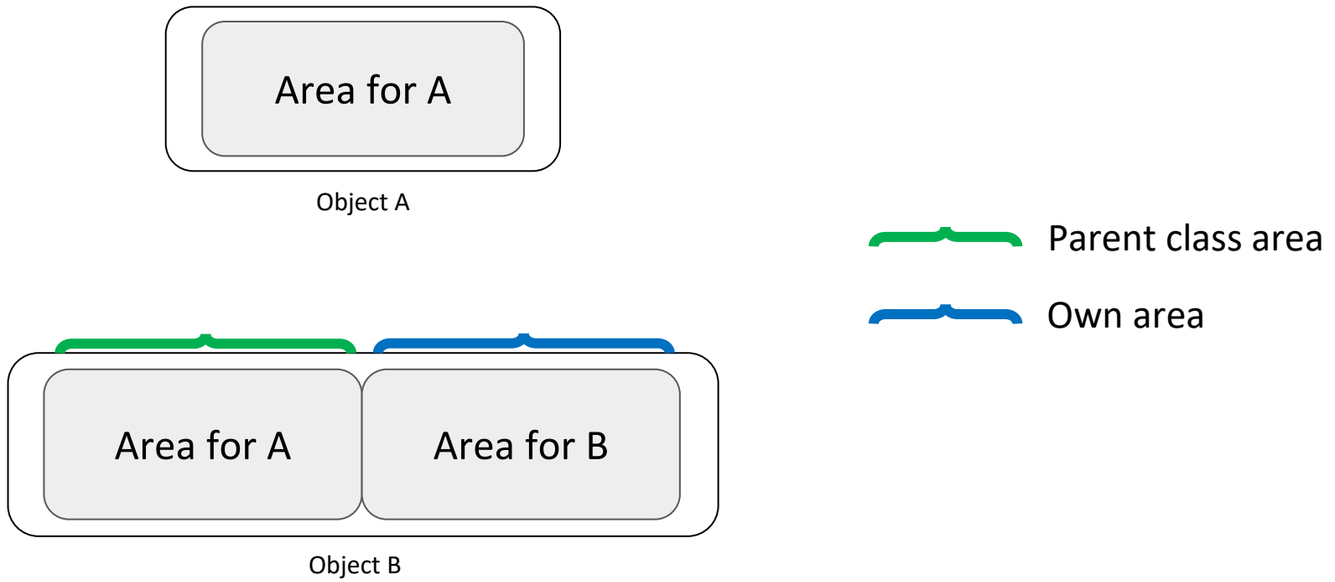
# Idea 3 - Class Object is composed of several areas

```
class A {
  int a;
}
```



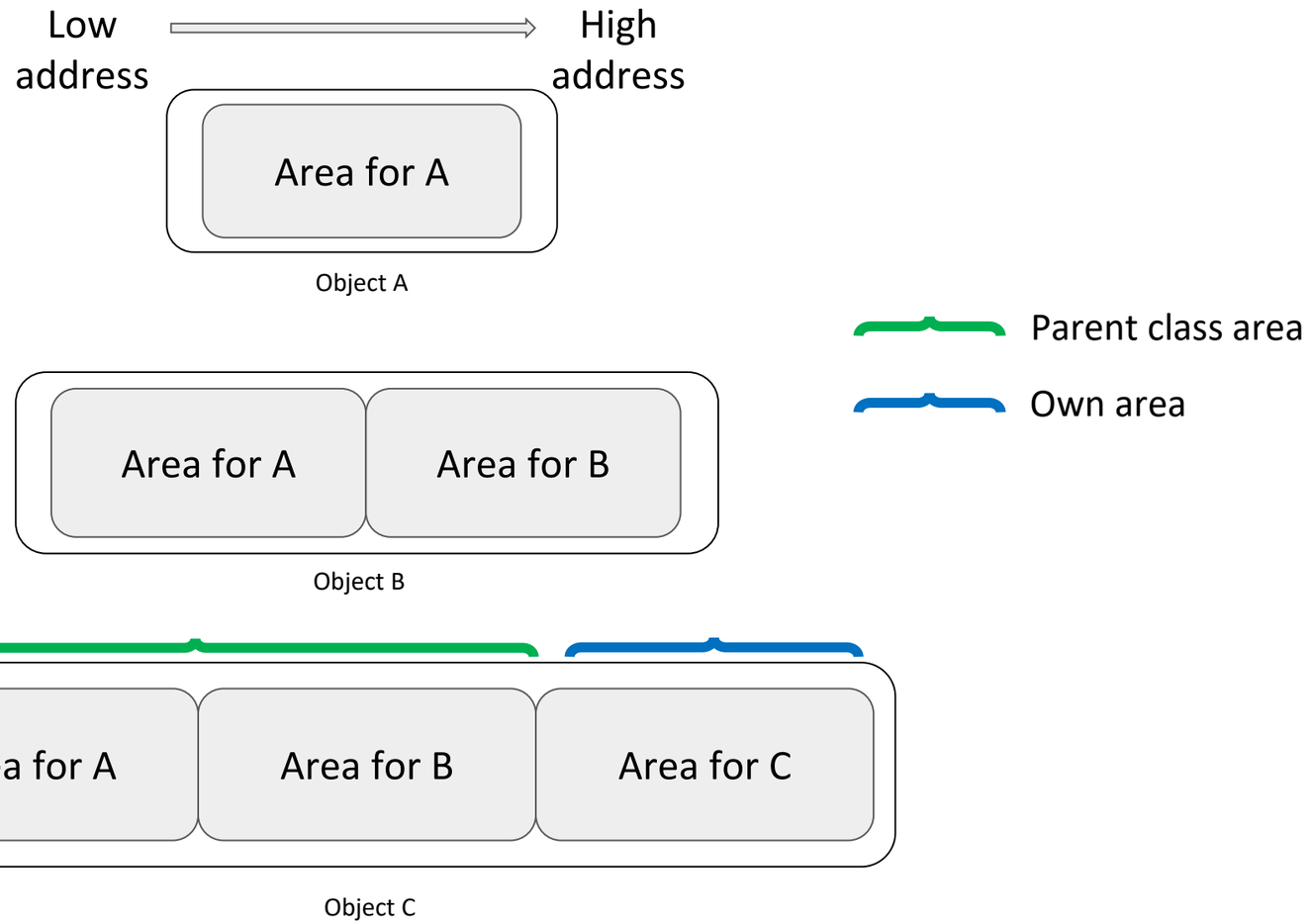
# Idea 3 - Class Object is composed of several areas

```
class A {
  int a;
}

Class B: A {
  int b;
}
```



# Idea 3 - Class Object is composed of several areas

```
class A {
  int a;
}

class B: A {
  int b;
}

class C: B {
  int c;
}
```



# Idea 4 - Type confusion bug is triggered once an area does not exist

```
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  };
```

```
16  void func_a(B* b) {
17      // Access an area of class A
18      b->var_a = 0xaaaaaaaa;
19  }
20
21  void func_b(B* b) {
22      // Access an area of class B
23      b->var_b = 0xbbbbbbbb;
24  }
```

- Both *func_a()* and *func_b()* receive *B** type argument
  - But access different area

# Idea 4 - Type confusion bug is triggered once an area does not exist

```
16  void func_a(B* b) {
17      // Access an area of class A
18      b->var_a = 0xaaaaaaaa;
19  }
20
21  void func_b(B* b) {
22      // Access an area of class B
23      b->var_b = 0xbbbbbbbb;
24  }
```

```
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  };
```

```
26  int main() {
27      A a;
28      B b;
29      C c;
30      func_a((B*)(&a)); // No problem
31      func_b((B*)(&a)); // Problem
32      func_a((B*)(&b)); // No problem
33      func_b((B*)(&b)); // No problem
34      func_a((B*)(&c)); // No problem
35      func_b((B*)(&c)); // Problem
36      return 0;
37  }
```

- Line 30 and 34, Typecasting (A/C to B) occurred
- but it isn't actually a problem
  - access to existing area(A)

Figure 1

# Idea 4 - Type confusion bug is triggered once an area does not exist

```
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  };
```

```
16  void func_a(B* b) {
17      // Access an area of class A
18      b->var_a = 0xaaaaaaaa;
19  }
20
21  void func_b(B* b) {
22      // Access an area of class B
23      b->var_b = 0xbbbbbbbb;
24  }
```

```
26  int main() {
27      A a;
28      B b;
29      C c;
30      func_a((B*)(&a)); // No problem
31      func_b((B*)(&a)); // Problem
32      func_a((B*)(&b)); // No problem
33      func_b((B*)(&b)); // No problem
34      func_a((B*)(&c)); // No problem
35      func_b((B*)(&c)); // Problem
36      return 0;
37  }
```

- Line 31 and 35, Typecasting (A/C to B) occurred
- It is a problem
  - access to the non-existing area(B)

# Idea 4 - Type confusion bug is triggered once an area does not exist

- Area-based approach
- Assembly codes require valid(matched) areas to operate properly
- Type confusion bug occurs once the required area doesn't exist in the actual object
  - we're able to detect type confusion bug by checking the existence of the required area

- Why do we use the concept of the *area*?
  - *Area* can tell the condition that an interacted object must have
    - Compatibility: itself and its child types
      - *Area* is accumulated during inheritance

```
16  void func_a(B* b) {
17      // Access an area of class A
18      b->var_a = 0xaaaaaaaa;
19  }
20
21  void func_b(B* b) {
22      // Access an area of class B
23      b->var_b = 0xbbbbbbbb;
24  }
```

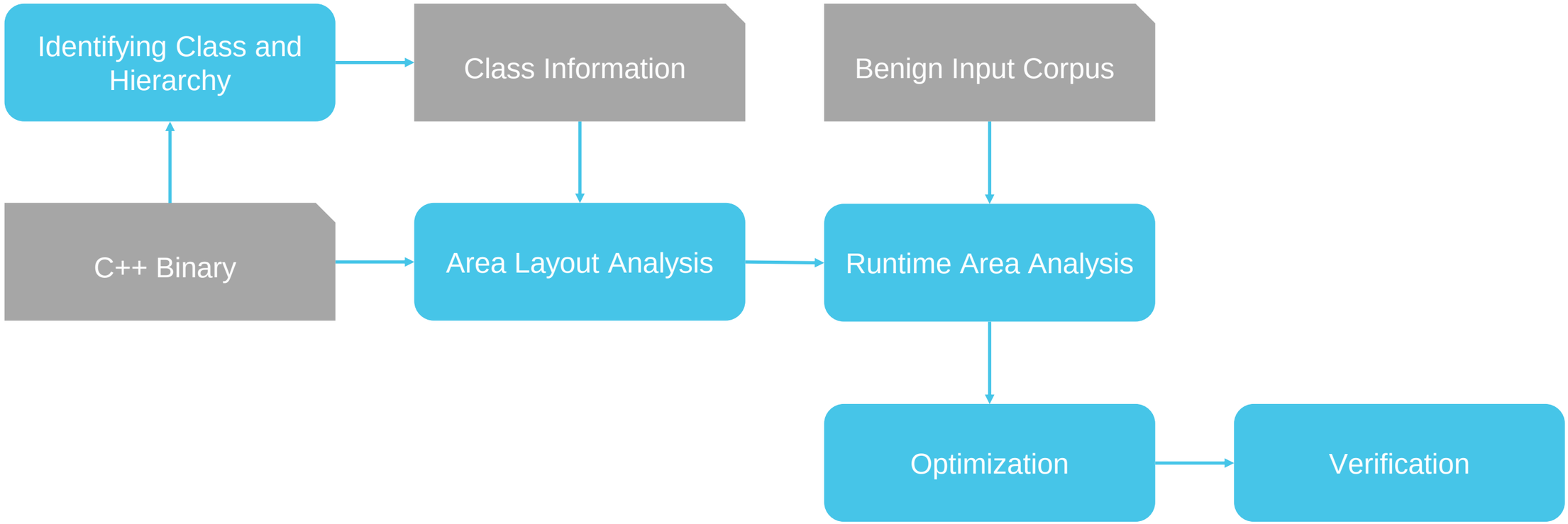
# Idea 4 - Type confusion bug is triggered once an area does not exist

- Area-based approach
- Assembly codes require valid(matched) areas to operate properly
- Type confusion bug occurs once the required area doesn't exist in the actual object
  - we're able to detect type confusion bug by checking the existence of the required area

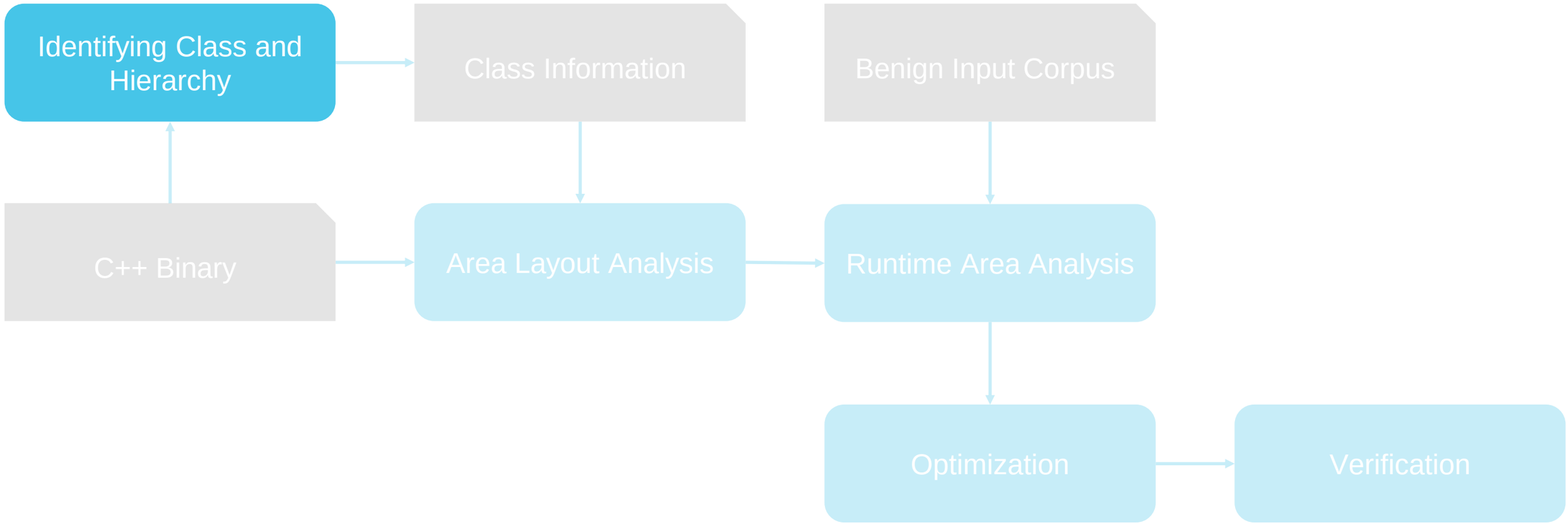
- Important things we have to know to detect bug:
  - **Which area does the assembly code needs(accesses)?**
  - **Does the actual object have that area?**

= required area

# Our approach



# Our approach



# Identifying Class and Hierarchy

- This step is to inference/recover class information from binary
  - Class identifier
  - Class hierarchy
  - ...

- It’s another huge research area itself
  - There are many existing researches for restoring class information
  - VCI, MARX, DeClassifier, OOAnalyzer...

- Best situation: Use published tools and BinTyper just uses recovered class information

# Identifying Class and Hierarchy

- Oops

*allow_non_pe: boolean*

The OOAnalyzer tool only really supports Windows 32-bit executables produced by Microsoft Visual Studio. Whil
requirement for these input executables to be in the Portable Executable (PE) file format, there is in practice a str
the supported files and the PE file format. Due to confusion about OOAnalyzer not really supporting executables

OOAnalyzer

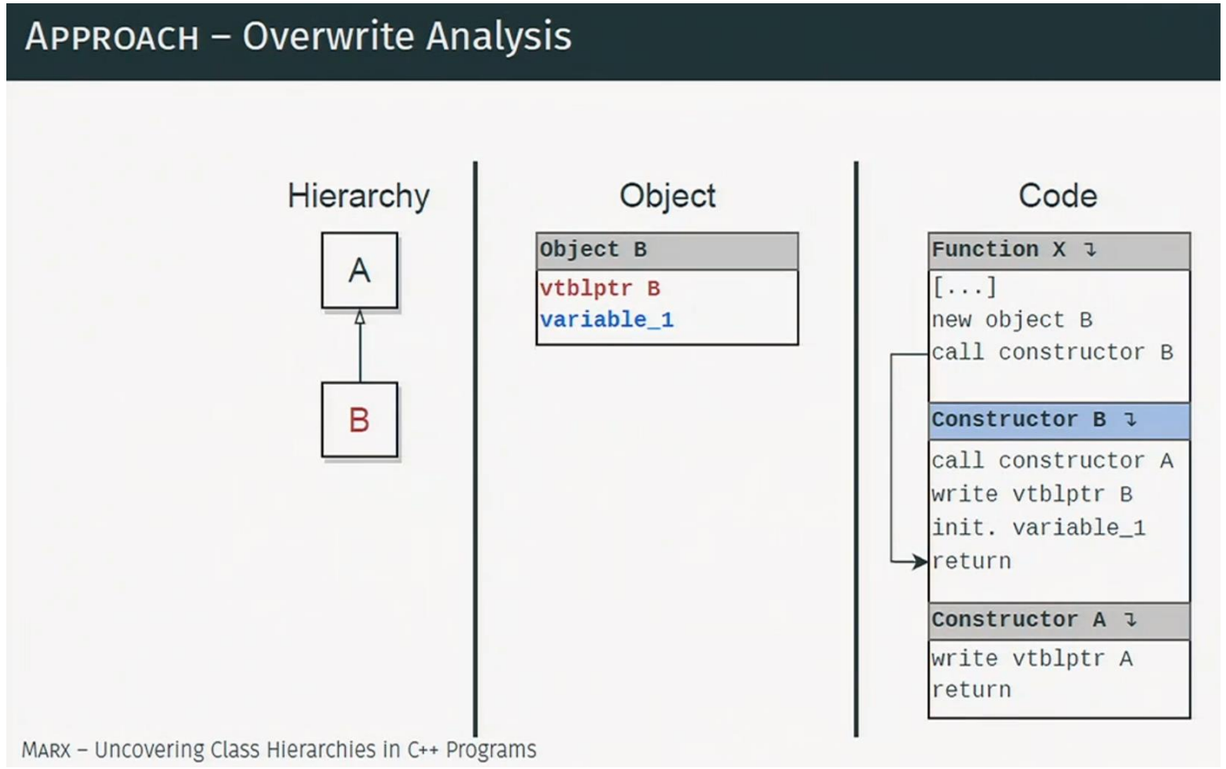
```
user@ubuntu:~/marx_pdfium_test$ cat pdfium_test.hierarchy | wc -l
3
```

MARX

# Identifying Class and Hierarchy

- Manually implements some of its ideas
  - Use VFT as a unique class identifier
  - VFT detection
  - Constructor/Destructor analysis
  - VFT Overwrite analysis

    * VFT: Virtual function table
- Not a perfect implementation but it works
  - It's enough to prove our ideas




*(Image reference: presentation of MARX)*

# Identifying Class and Hierarchy

- Sample of recovered class identifiers and hierarchies

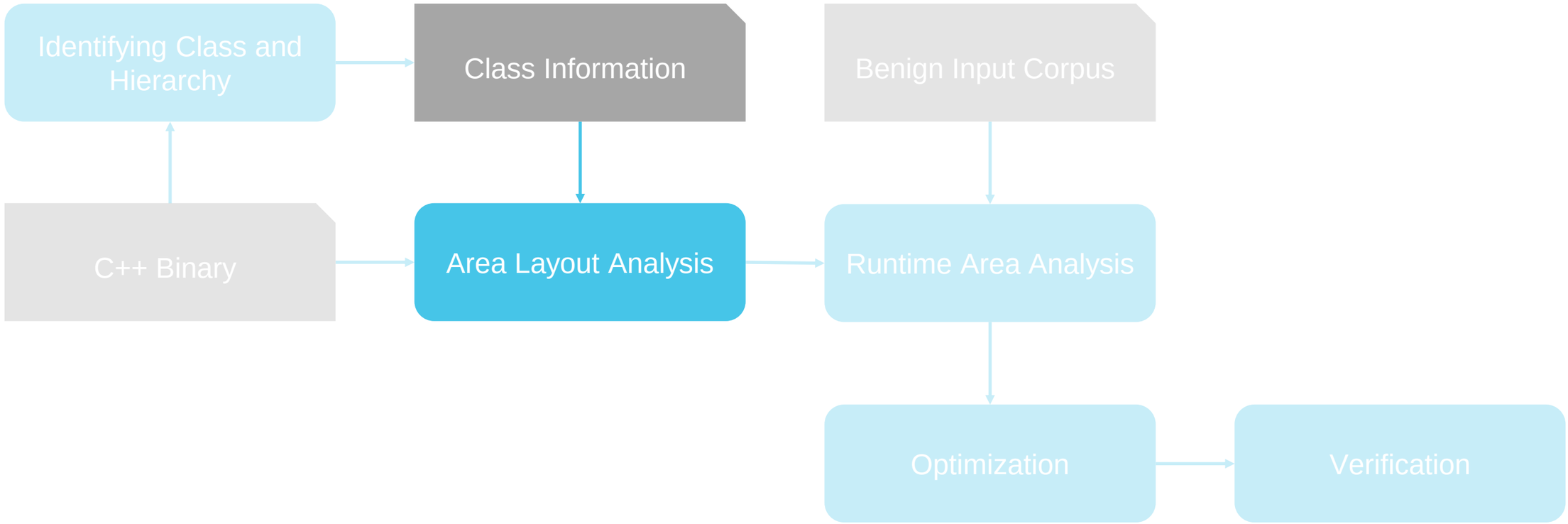
```
149799120, 151023368, 149358192, 153556584, 153547152, 154696112, 154801992, 153000496,
154857600, 152211448, 149620496, 149816976, 150307096, 149494696, 152159056, 149915288,
151301064, 149256672, 154031616, 149327392, 149456568, 154332768, 153791248, 149416136,
150576008, 149185328, 149587672, 149332120, 150564848, 149718752, 150039504, 149287384,
151025360, 150584656, 149451344, 149325560, 149290224, 149800216, 150308608, 152950400,
149686040, 150338144, 149540528, 150764776, 152288264, 149915424, 149799216, 149464656,
149632184, 151445072, 150008624, 149358400, 149662688, 149804896, 149807696, 149194568,
```

Figure 1

```
7899        "152164016": {
7900            "0": 152105576,
7901            "368": 149444536
7902        },
7903        "150630176": {
7904            "0": 149268656
7905        },
7906        "152123712": {
7907            "0": 152124328,
7908            "32": 152117104
7909        },
7910        "152014560": {
7911            "0": 149290944
7912        },
7913        "149210616": {
7914            "0": 149210528
7915        },
```

Figure 2

# Our approach



# Area Layout Analysis

- ***'area'*** : *unique member fields of each class*
- The ***Area Layout*** provides internal area structure
  - Where each area of the object starts/ends
- BinTyper performs static analysis to obtain Area Layout of classes
  - Based on recovered information about the class identifier and hierarchy

```
Area Layout of C
        [ {identifier: 0xAAAA,
           offset: 0x00,
           size:0x08},
          {identifier: 0xBBBB,
           offset: 0x08,
           size:0x08},
          {identifier: 0xCCCC,
           offset: 0x10,
           size:0x08} ]
```

# Area Layout Analysis

- Example of Area Layout

```
1   class A {
2   public:
3       long a;
4   };
5
6   class B: public A {
7   public:
8       long b;
9   };
10
11  class C: public B {
12  public:
13      long c;
14  };
```

Figure 1

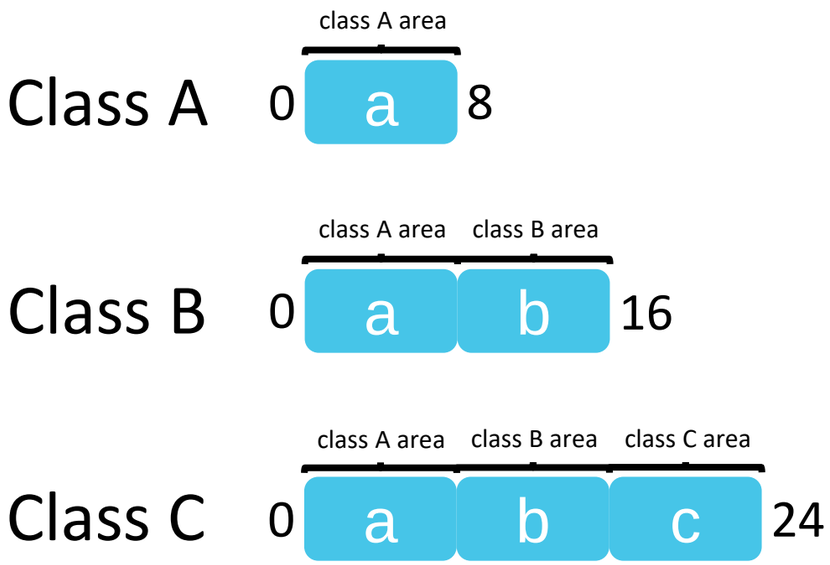



Figure 2

```
Area Layout of C
        [ {identifier: 0xAAAA,
           offset: 0x00,
           size:0x08},
          {identifier: 0xBBBB,
           offset: 0x08,
           size:0x08},
          {identifier: 0xCCCC,
           offset: 0x10,
           size:0x08} ]
```

Figure 3

# Area Layout Analysis

- The Area Layout is the list of the following pairs: *[identifier, (own) area starts, area ends]*
  - *area ends = area starts + area size*

- The size of class gradually increases through inheritance
  - *[Derived class size] = [Parent area size] + [Own area size]*
- **Area size**: calculating the difference
  - *[Own area size] = [Derived class size] - [Parent area size]*
- **Area starts**:
  - Own area(area of derived class) is located behind the parent area
  - Area starts := [Parent area size]

```
Area Layout of C
        [ {identifier: 0xAAAA,
           offset: 0x00,
           size:0x08},
          {identifier: 0xBBBB,
           offset: 0x08,
           size:0x08},
          {identifier: 0xCCCC,
           offset: 0x10,
           size:0x08} ]
```

# Area Layout Analysis

- The next question is: ***How to get the (derived) class size by static analysis?***

- In the case of an object located in the heap:
  - The new() operator directly indicates the size of the object
- In the case of an object located in the stack/global:
  - There is no direct clue indicating the size of the object

# Area Layout Analysis

- Assembly representation of code accessing class member variable
  - *QWORD / DWORD / WORD / BYTE PTR [class-ptr + offset-to-member]*
    - *class-ptr* indicates start address of an object
    - *offset-to-member* indicates offset that mapped for target member variable

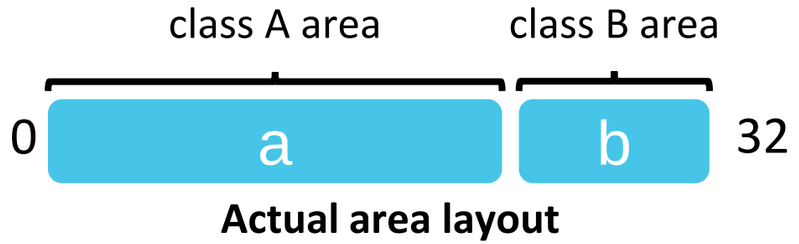
- Analyze the maximum possible *offset-to-member* value for each class identifier
  - From the virtual-methods and constructor/destructors of identifier

- BinTyper uses that value as object size
  - Maximum *offset-to-member* + Access unit size

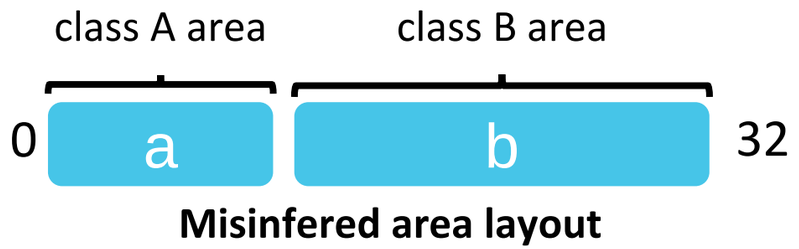
# Area Layout Analysis

- A similar approach was performed previously in DeClassifier
  - They analyze object size to infer class hierarchy

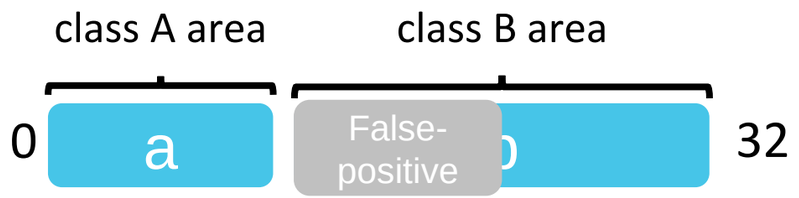
- Limitation: The analyzed object size may be incorrect
  - It may be smaller than the actual object size
  - This can lead to false positives in analysis



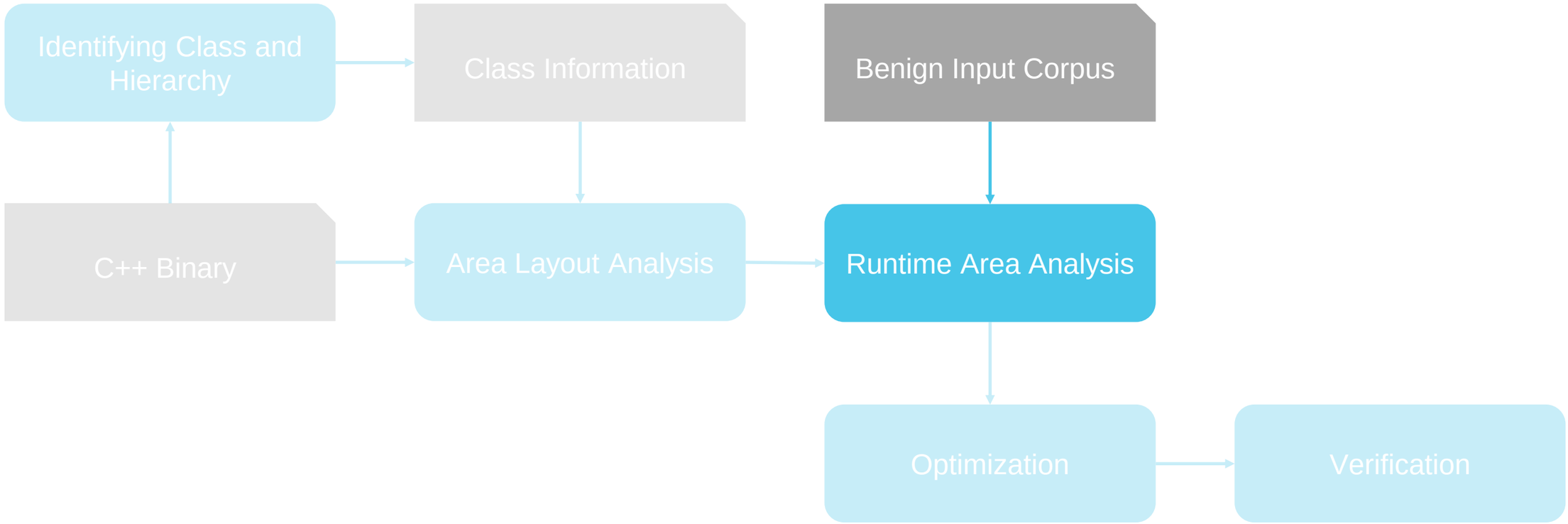
**Actual area layout**



**Misinfered area layout**



# Our approach



# Runtime Area Analysis

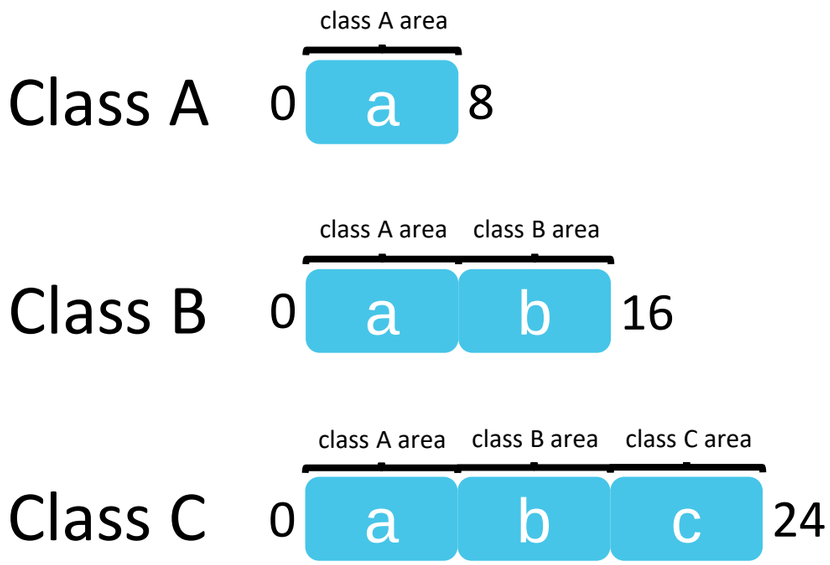
- The Runtime type information is unknown
  - Challenge 3 - Unknown Runtime Type Information
  - It’s hard to statically determine whether type(area) will be passed

- Approach – *Applying the dynamic analysis to obtain area information*
- Obtained(Recorded) type information is used for verification later

```
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
```

# Runtime Area Analysis

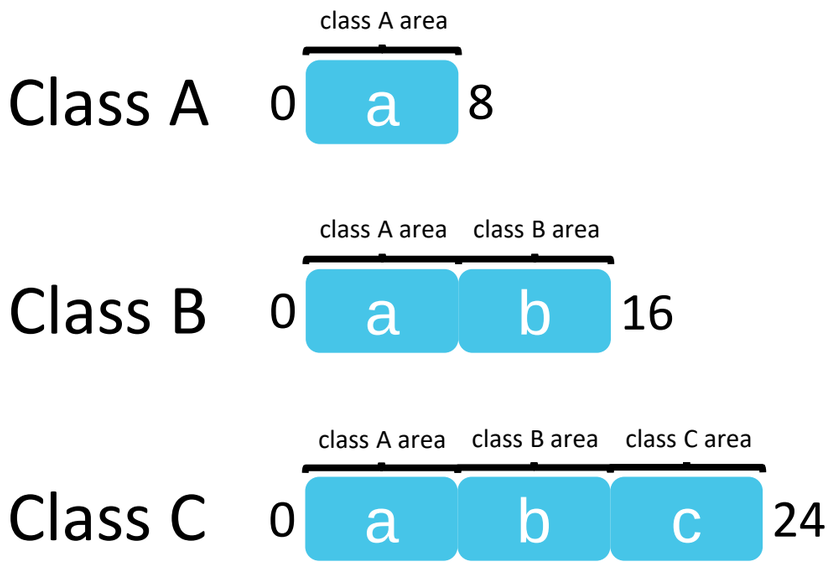
## Area Layout Information




```
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
```

# Runtime Area Analysis

## Area Layout Information

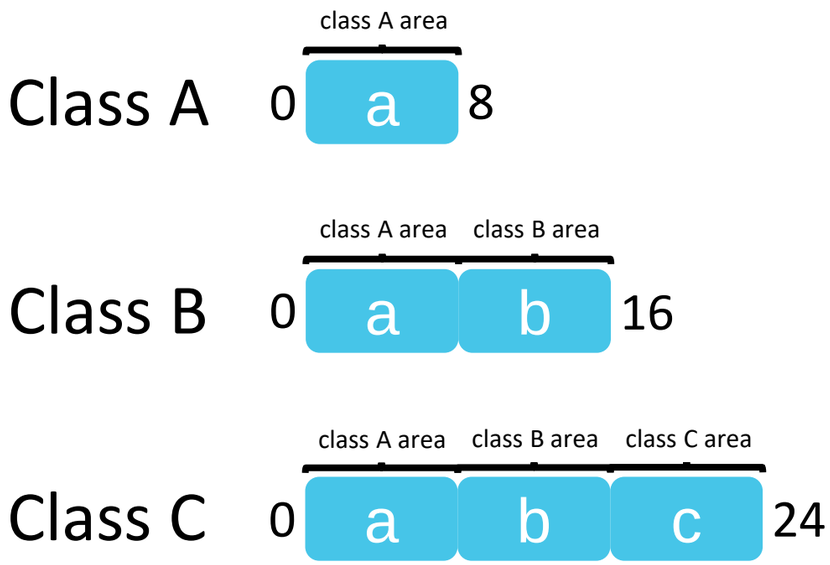



## RDI := Object of class C

```
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
```

# Runtime Area Analysis

## Area Layout Information

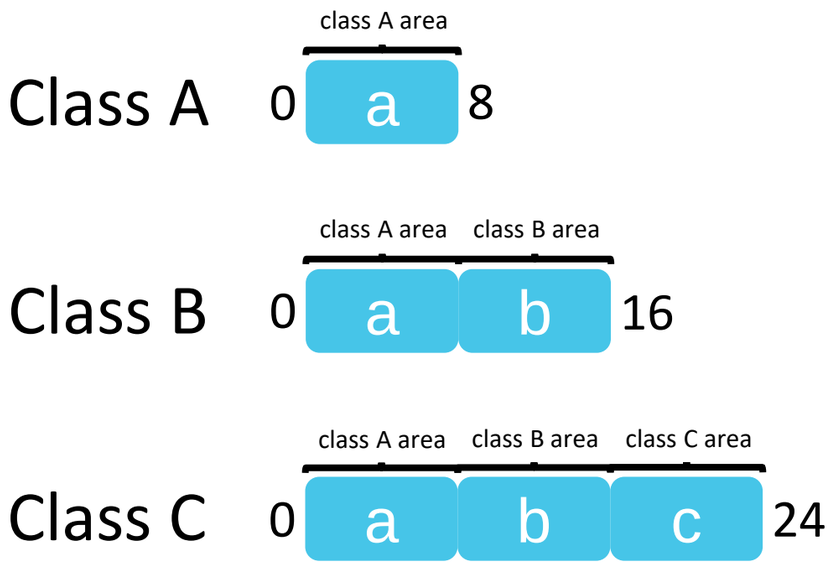



## RDI := Object of class C

```
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
```

# Runtime Area Analysis

## Area Layout Information

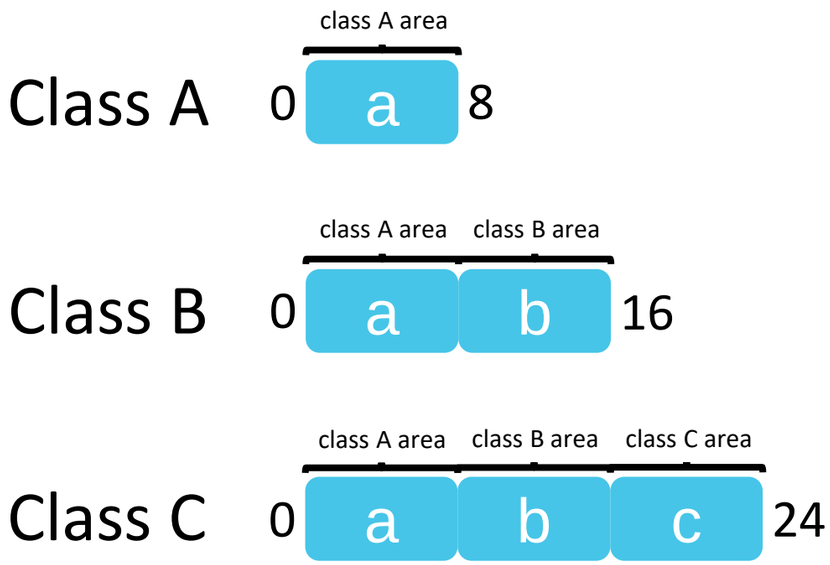



## RDI := Object of class C

```
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
```

# Runtime Area Analysis

## Area Layout Information

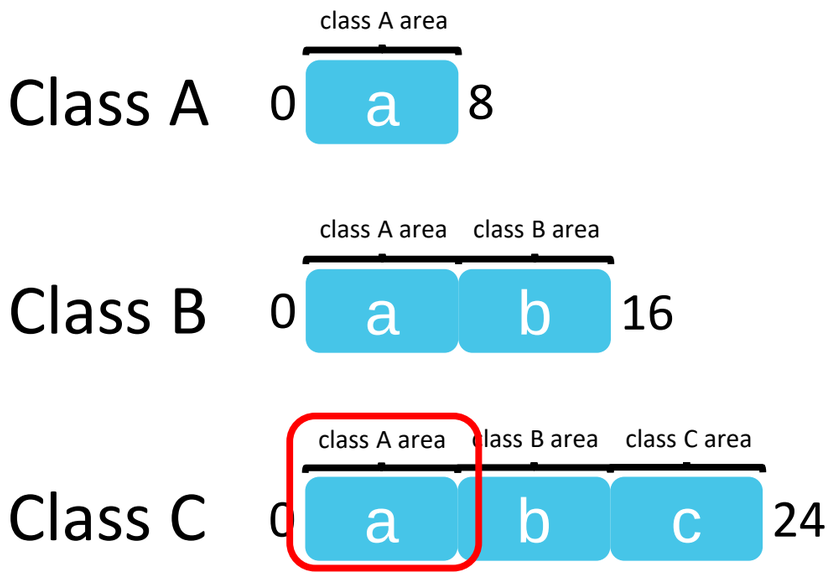



## RDI := Object of class C

```
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
```

# Runtime Area Analysis

## Area Layout Information

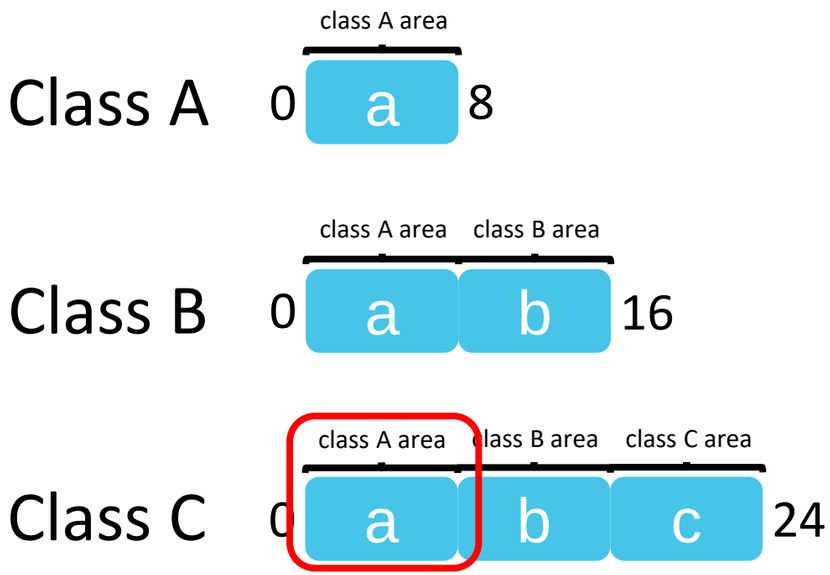



## RDI := Object of class C

```
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
```

# Runtime Area Analysis

## Area Layout Information




Class A area is accessed

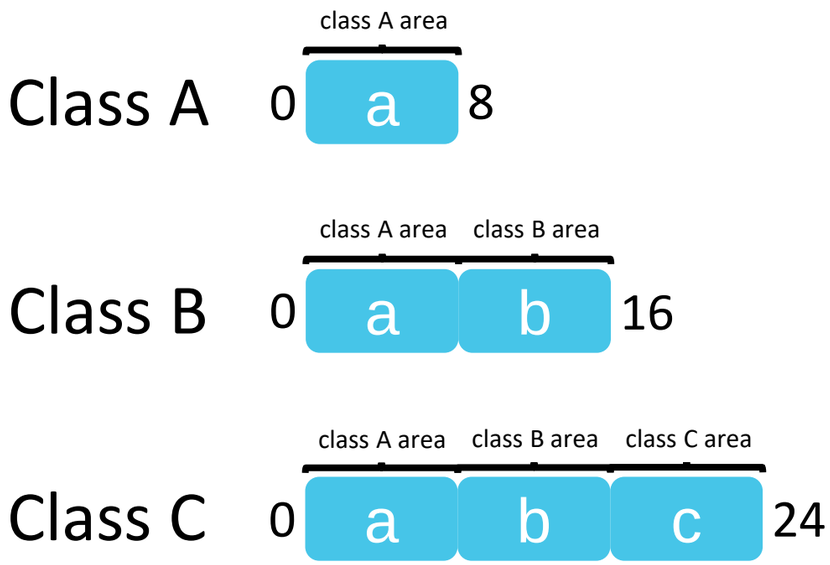
## RDI := Object of class C

```
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
```

Runtime information:
Area A

# Runtime Area Analysis

## Area Layout Information



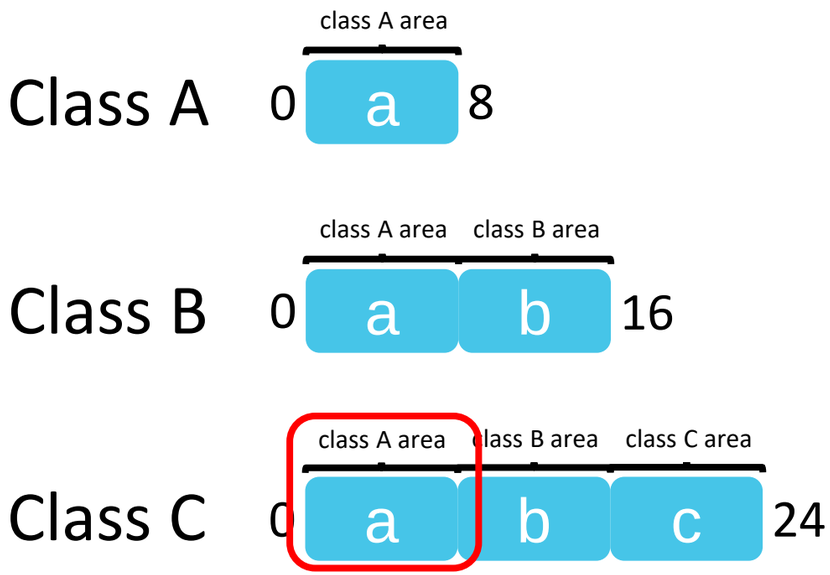

## RDI := Object of class C

```
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
```

Runtime information:
Area A

# Runtime Area Analysis

## Area Layout Information




Class A area is accessed

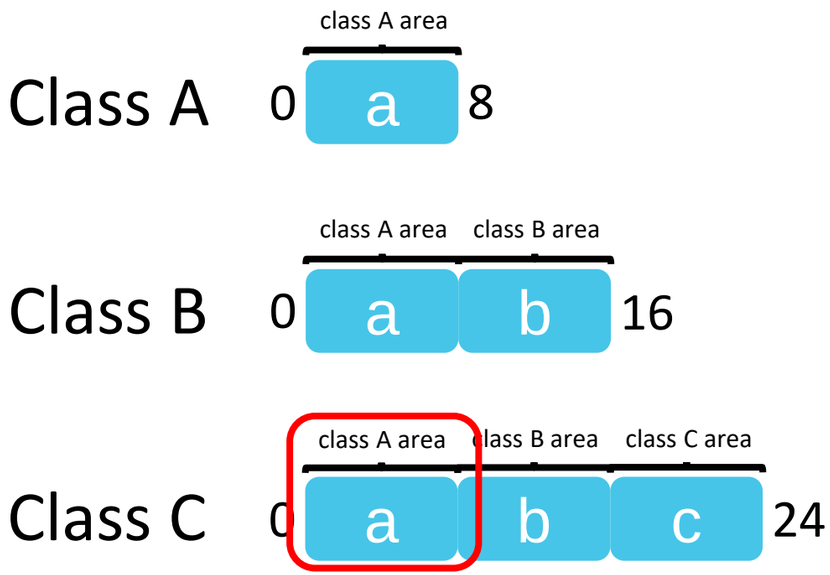
## RDI := Object of class C

```
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
```

Runtime information: Area A

# Runtime Area Analysis

## Area Layout Information




Class A area is accessed

## RDI := Object of class C

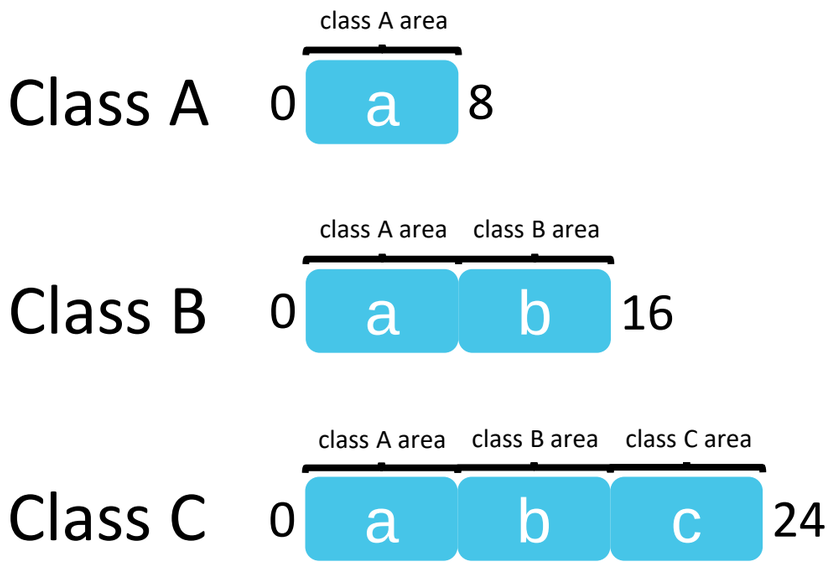
```
push    rbp
mov     rbp, rsp
mov     qword ptr [rbp - 8], rdi
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 1
mov     qword ptr [rax], rcx
pop     rbp
ret
```

Runtime information: Area A

Runtime information: Area A

# Runtime Area Analysis

## Area Layout Information




## RDI := Object of class C

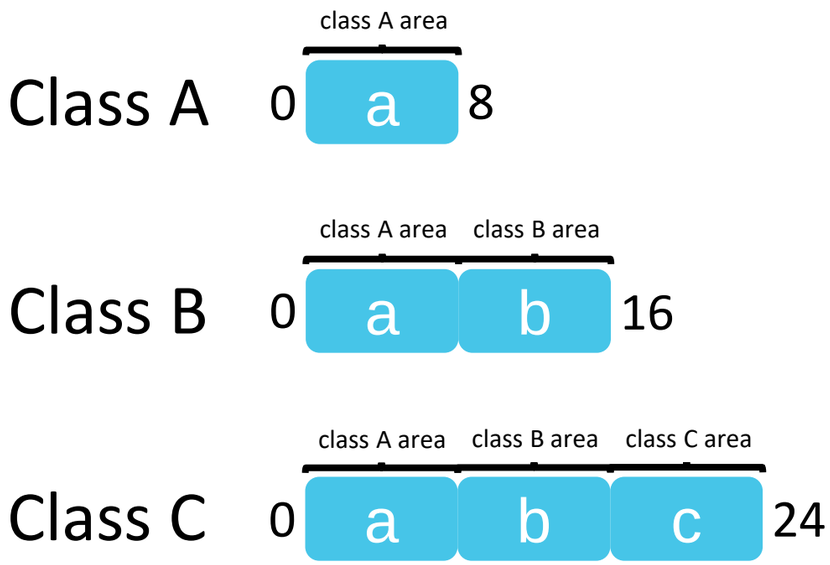
```
push    rbp
mov     rbp, rsp
mov     qword ptr [rbp - 8], rdi
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 1
mov     qword ptr [rax], rcx
pop     rbp
ret
```

Runtime information: Area A

Runtime information: Area A

# Runtime Area Analysis

## Area Layout Information




## RDI := Object of class C

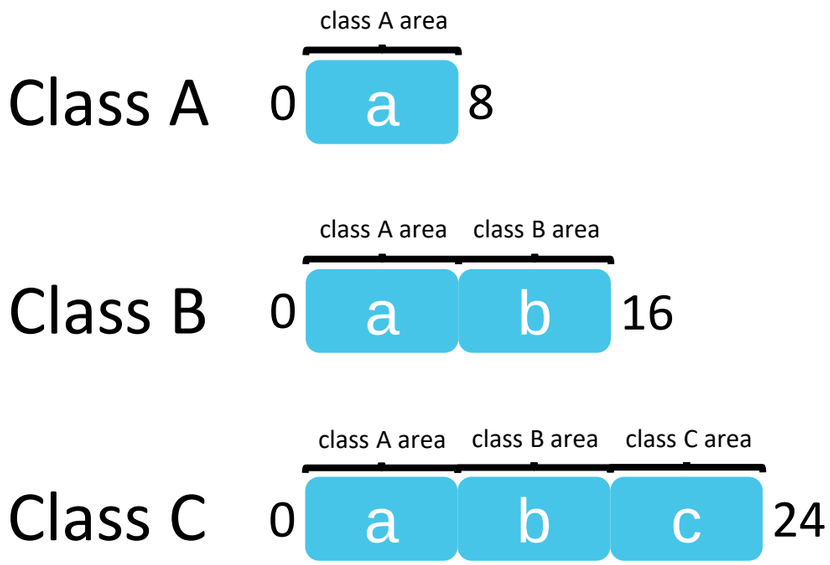
```
push    rbp
mov     rbp, rsp
mov     qword ptr [rbp - 8], rdi
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax]
add     rcx, 1
mov     qword ptr [rax], rcx
pop     rbp
ret
```

Runtime information: Area A

Runtime information: Area A

# Runtime Area Analysis

## Area Layout Information




## RDI := Object of class 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
```

Runtime information: Area A

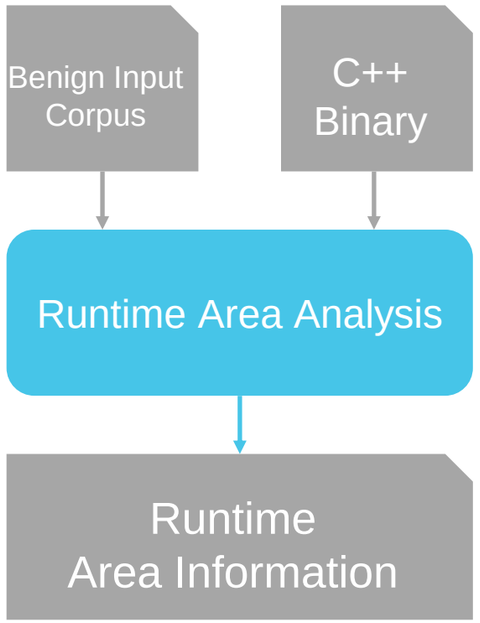
Runtime information: Area A

Idea 2 - Assembly representation doesn't be shared

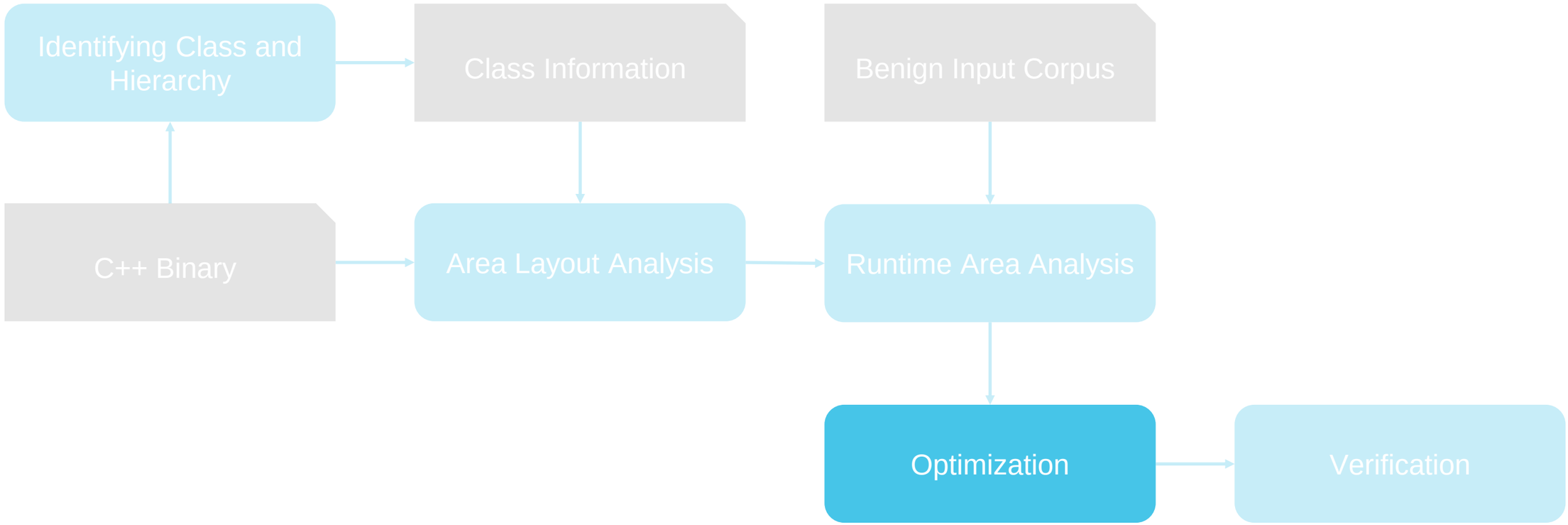
# Runtime Area Analysis

- The coverage is a challenge
  - This step is performed by dynamic analysis
  - Only covered(executed) instruction is analyzed

- Benign seed corpus
  - A set of input files which the binary processes
    - Which doesn't trigger any type confusion bugs
  - Used to increase the coverage of Runtime Area Analysis
  - User should maintain benign seed corpus and re-use it



# Our approach



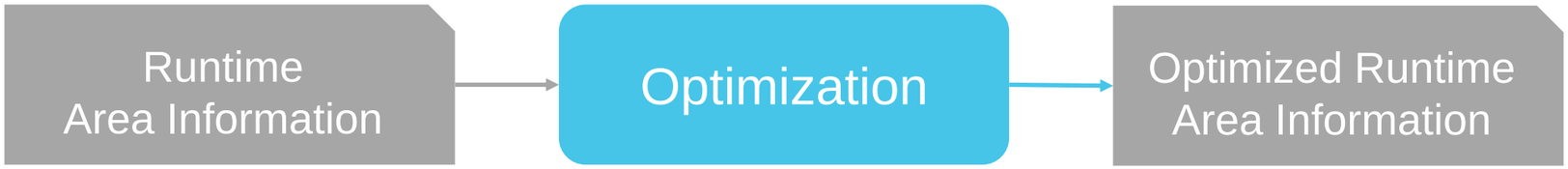
# Optimization

- Recorded information contains many duplicates
  - ... for verification usage
- Example

```
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
```

Runtime information: Area A

Runtime information: Area A



# Optimization

```
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
```

Runtime information:
Area A

Runtime information:
Area A

# Optimization

```
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
```

Runtime information:
Area A

**Duplicated checks**

# Optimization

```
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
```

Runtime information:
Area A

It is enough for verification

# Optimization

- Simple optimization rules:
  - Same basic block
  - Same area
  - Same register
  - ... reduce 30% of duplicates in our test

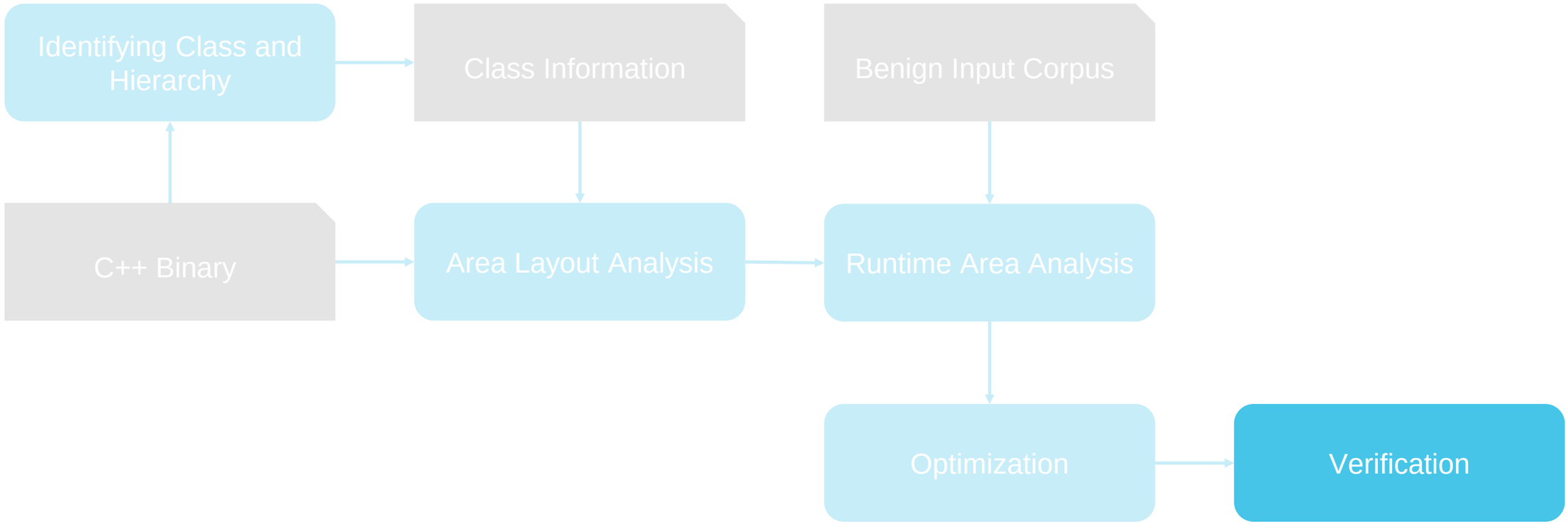
- More advanced optimization may be possible
  - Symbolic execution
  - (Post)-dominator analysis
  - ...

```
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
```

Runtime information: Area A

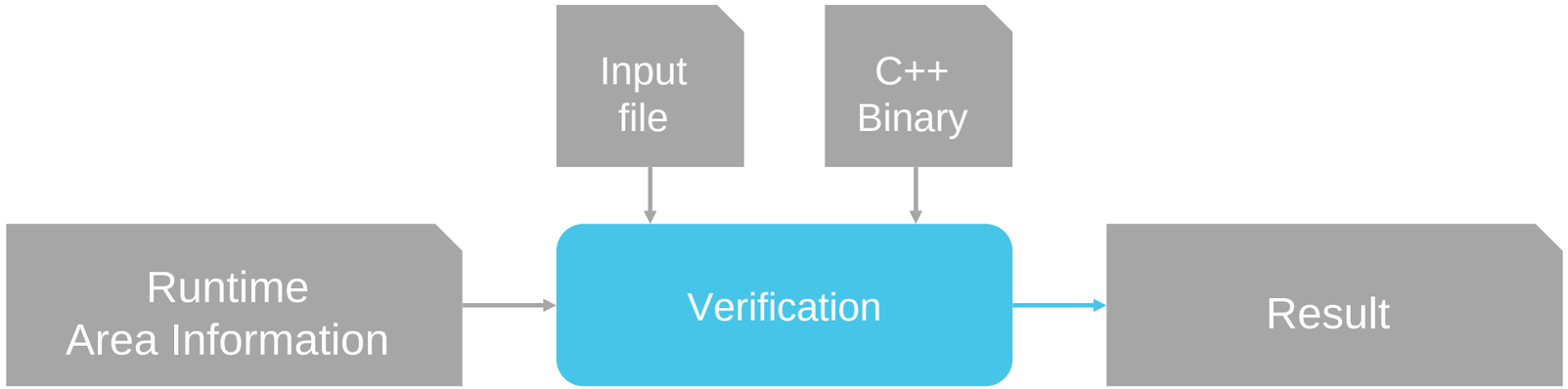
Runtime information: Area A

# Our approach



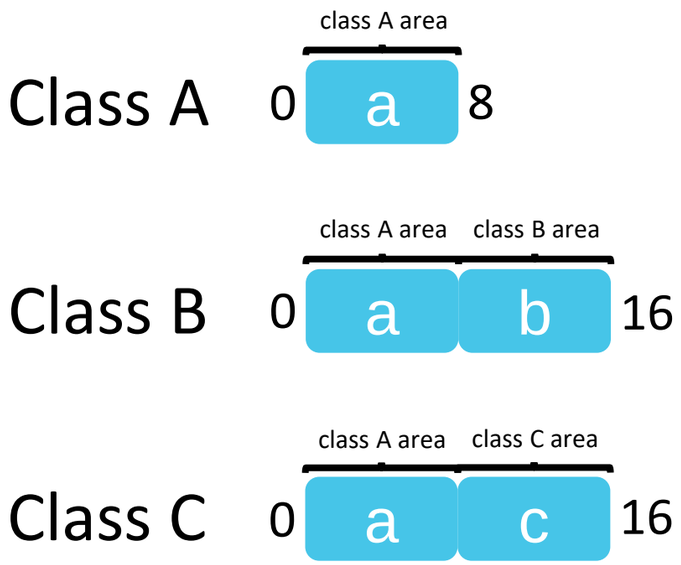
# Verification

- Execute the binary with verification
- Compare accessed area information with recorded runtime area information



# Verification

## Area Layout Information

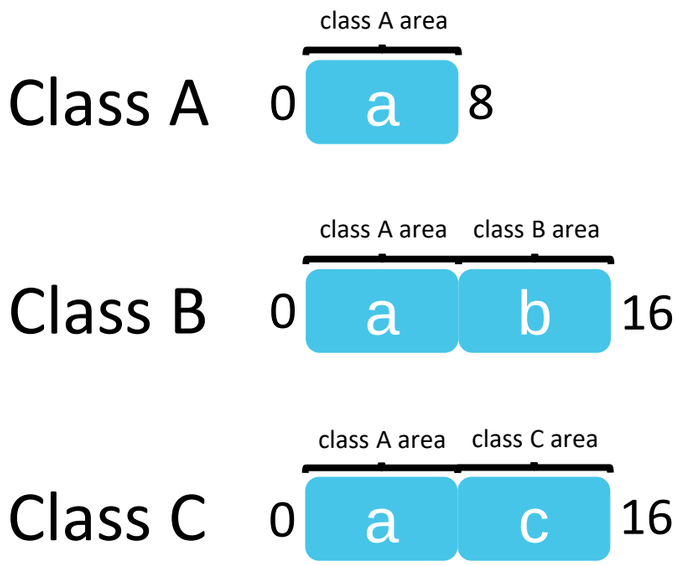



```
push    rbp
mov     rbp, rsp
mov     qword ptr [rbp - 8], rdi
mov     rax, qword ptr [rbp - 8]
mov     rcx, qword ptr [rax + 8]
add     rcx, 1
mov     qword ptr [rax + 8], rcx
pop     rbp
ret
```

Recorded Runtime information:
Area B

# Verification

## Area Layout Information



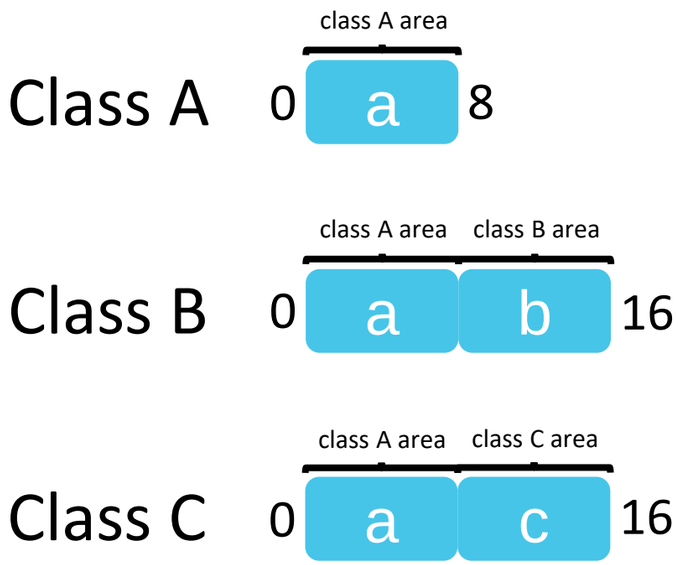

RDI := Object of class C

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

Recorded Runtime information:
Area B

# Verification

## Area Layout Information



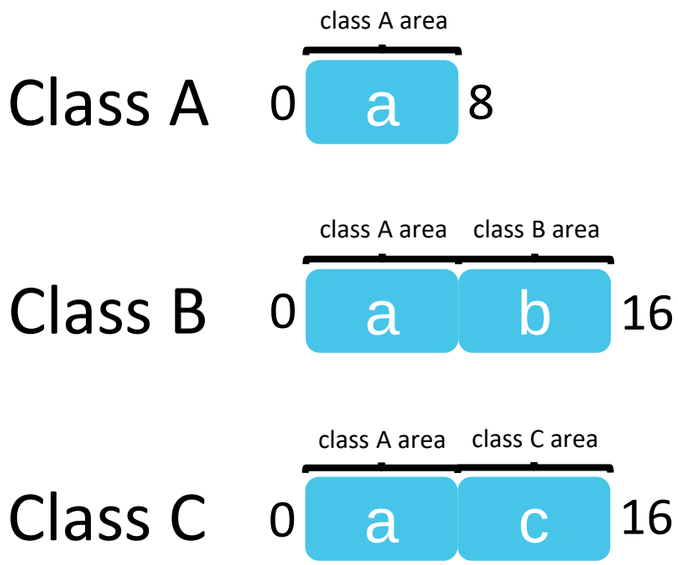

RDI := Object of class C

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

Recorded Runtime information:
Area B

# Verification

## Area Layout Information



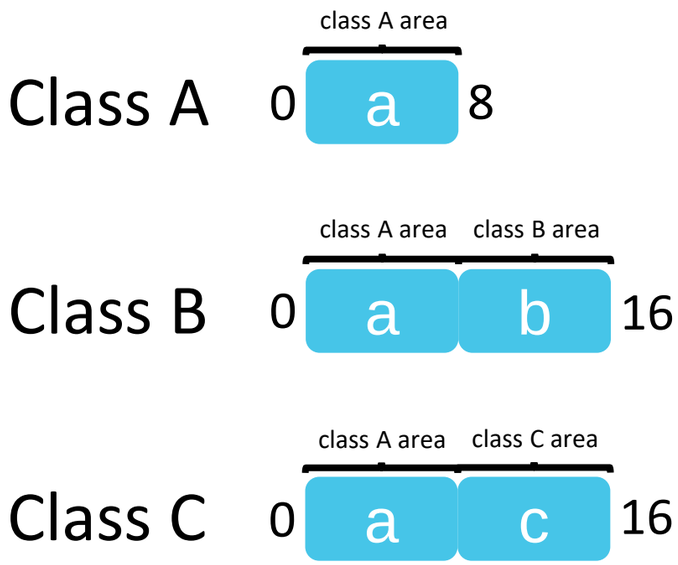

RDI := Object of class C

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

Recorded Runtime information:
Area B

# Verification

## Area Layout Information



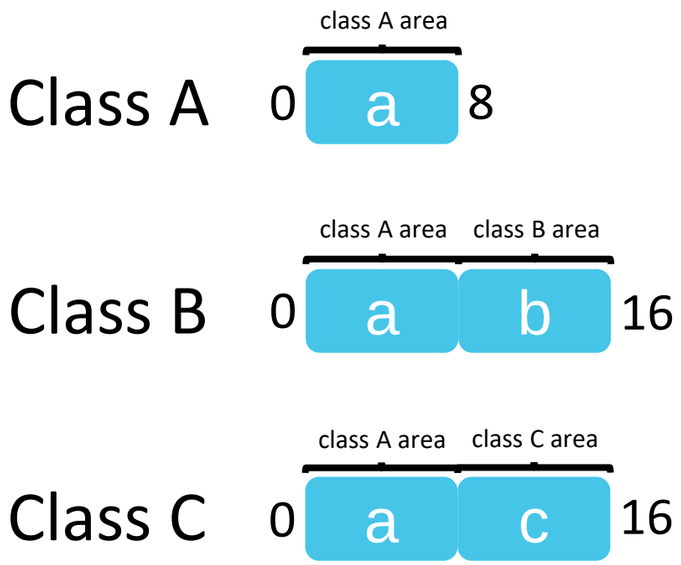

RDI := Object of class C

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

Recorded Runtime information:
Area B

# Verification

## Area Layout Information




RDI := Object of class C

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

Actual Area Information:
Area C

Recorded Runtime information:
Area B

Area Information Mismatched (B vs C)
= Type confusion bug has occurred

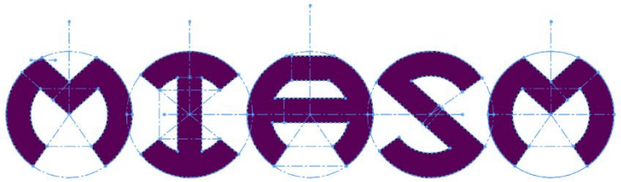
# Verification

- Once a type confusion bug detected, the following information is logged
  - RVA of fault instruction
  - Actual area information
  - Recorded(Required) area information

- Analyst can use this information to perform further analysis

# About the implementation

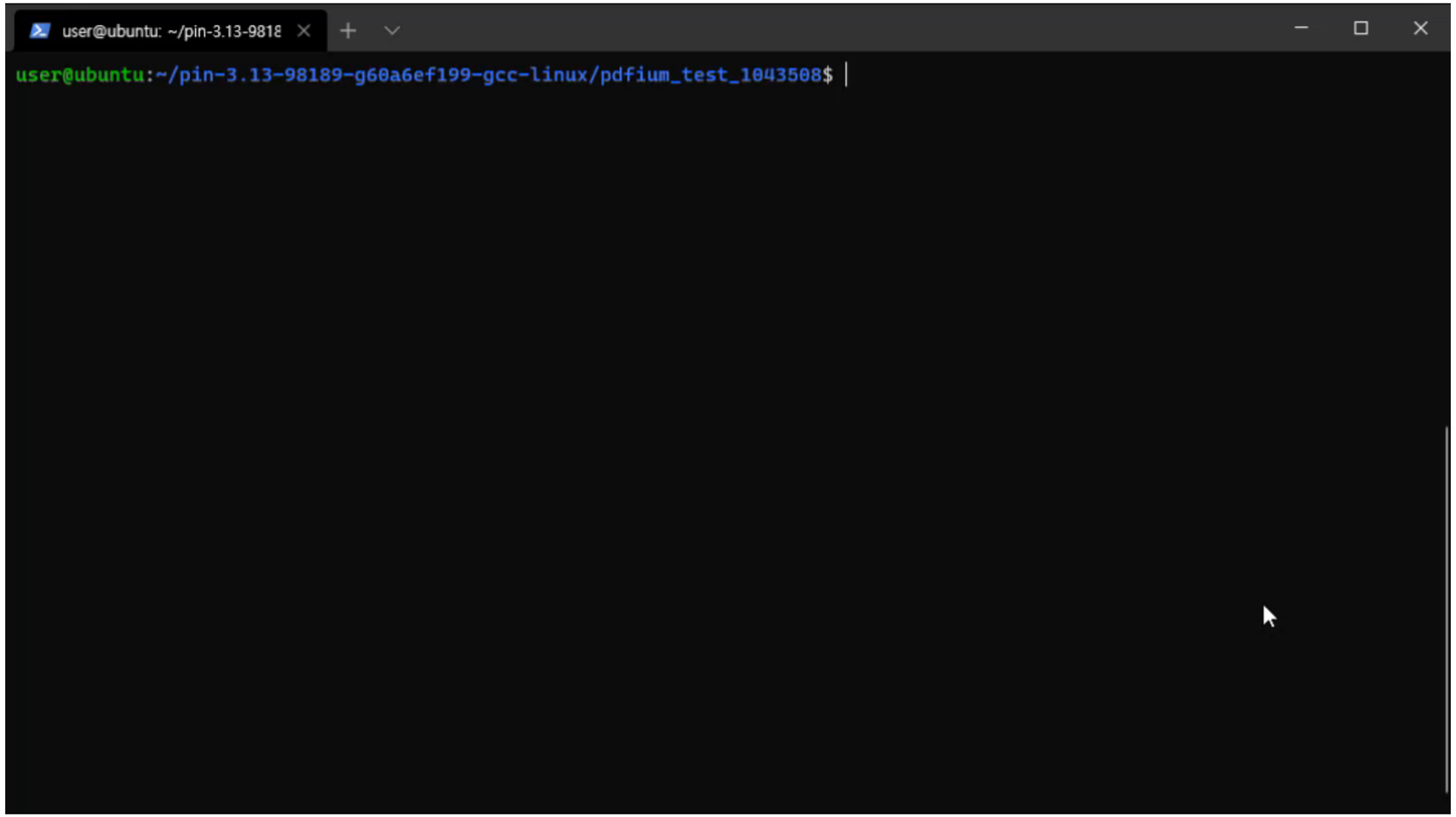
- Static analysis
  - IDA + IDAPython
  - Miasm
- Dynamic analysis
  - Intel Pin


# Demo

- Detection demo for Chromium bug 1043508
  - Type confusion bug in PDFium, the PDF rendering library of Chrome

# Demo


user@ubuntu: ~/pin-3.13-9818
user@ubuntu:~/pin-3.13-98189-g60a6ef199-gcc-linux/pdfium_test_1043508$

# Demo

Comment 4 by bugdroid on Fri, Jan 24, 2020, 3:15 AM GMT+9 **Project Member**

The following revision refers to this bug:
https://pdfium.googlesource.com/pdfium/+/02b6176bd77acf5672f56f1091ce5e495e5687fc

commit 02b6176bd77acf5672f56f1091ce5e495e5687fc
Author: Tom Sepez <tsepez@chromium.org>
Date: Thu Jan 23 18:15:08 2020

**Avoid casting CXFA_FFListBox to CXFA_FFComboBox.**

They are different subclasses of CXFA_FFDropDown.

```
[+] Type confusion detected
RVA : 0x000000000108be84
Actual accessed area : 0x0000000001337a38
Required areas : 0x0000000001337468
```

```
off_1337A38     dq offset n             ; DATA XREF: CFWL_ListBox::CFWL_ListBox
                                        ; CFWL_ListBox::~CFWL_ListBox()+E↑o
                dq offset n
                dq offset n

off_1337468     dq offset n             ; DATA XREF: CFWL_ComboBox::CFWL_ComboBox(
                                        ; CFWL_ComboBox::~CFWL_ComboBox()+9↑o ...
                dq offset n
```

# Future work & Limitation

- Our current implementation is based on DBI (Intel Pin)
  - For proof-of-concept: Easy to implement but low performance
    - 10x~. It’s impractical for the fuzzing purpose
  - Performs binary-rewriting for performance and usability
    - Retrowrite: Statically Instrumenting COTS Binaries for Fuzzing and Sanitization" (in IEEE S&P’20)
- Advanced static analysis for class information recovery / area layout analysis
- Propagating Runtime Type Information
- Highly-optimized binaries
  - Optimization make static analysis inaccurate for the inference of class information
  - e.g. Function inlining

# Conclusion

- The compiled binaries lack high-level information. It makes sanitization difficult
  - Especially for type confusion bug detection

- Class information can be recovered
  - Based on OOP-related characteristic

- By combining static analysis and dynamic analysis, Runtime type confusion detection is possible

# Thanks

- Q&A





* 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)