NTT

Egg Hunting without Eggs:

Identifying Memory Locations of Objects with Structural Characteristics

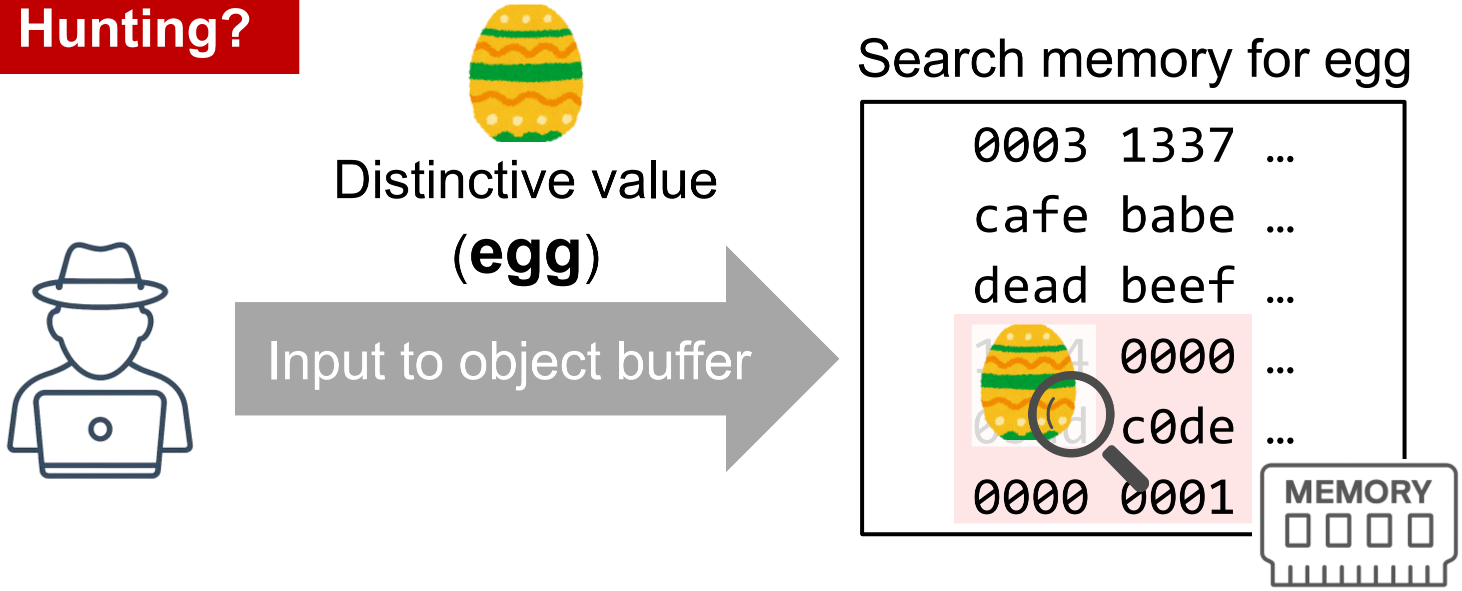
Toshinori Usui, Yuto Otsuki, Yuhei Kawakoya, Eitaro Shioji, Makoto Iwamura
NTT Social Informatics Laboratories

Toshinori Usui, Ph.D.

- Assoc. distinguished researcher, security principal @NTT
- Interested in: malware analysis, reverse engineering, and offensive security
- Speaks at: Black Hat USA, RAID, ACSAC etc.
- Loves: CTF, Brazilian Jiu-Jitsu

Q. What if you want to locate an object useful for exploitation?

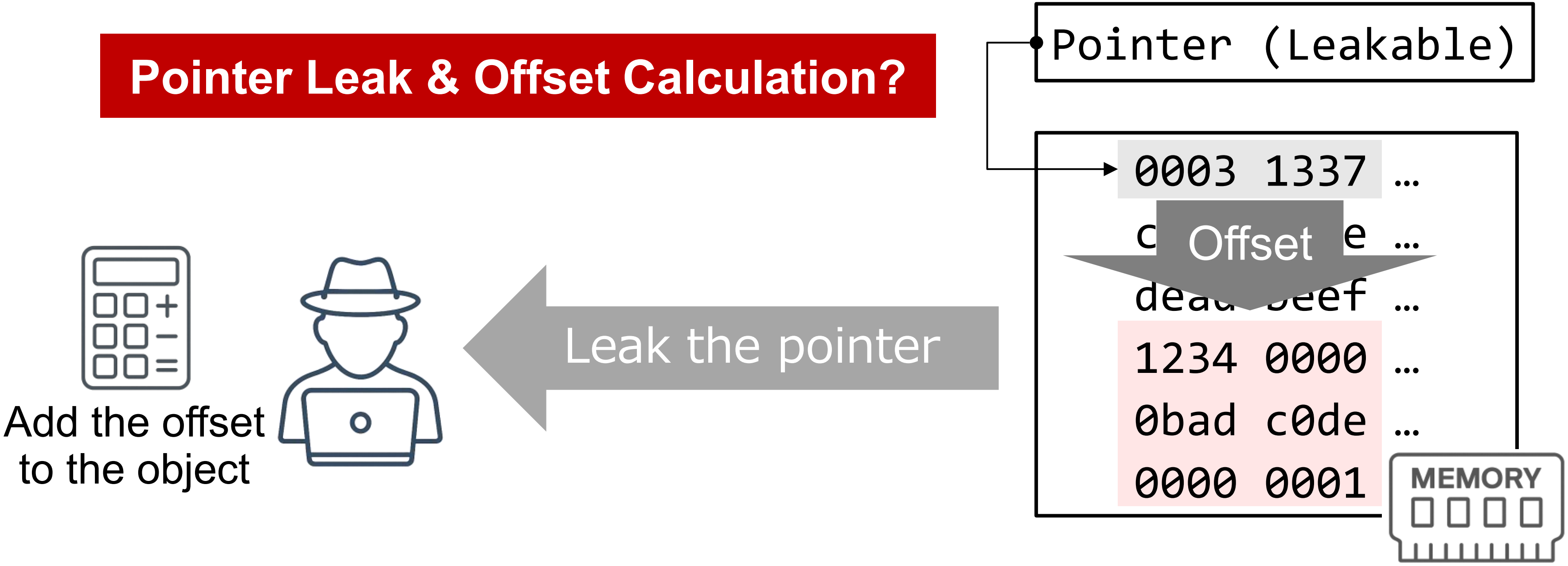
Egg Hunting?



No, it does not work when we don’t have controllable buffer for a distinctive value (i.e., egg).

Q. What if you want to locate an object useful for exploitation?

Pointer Leak & Offset Calculation?



No, it does not work when the pointer or the derived offset from it are unavailable.

Today's Talk

New Technique to Identify Memory Locations of Objects

No need to:

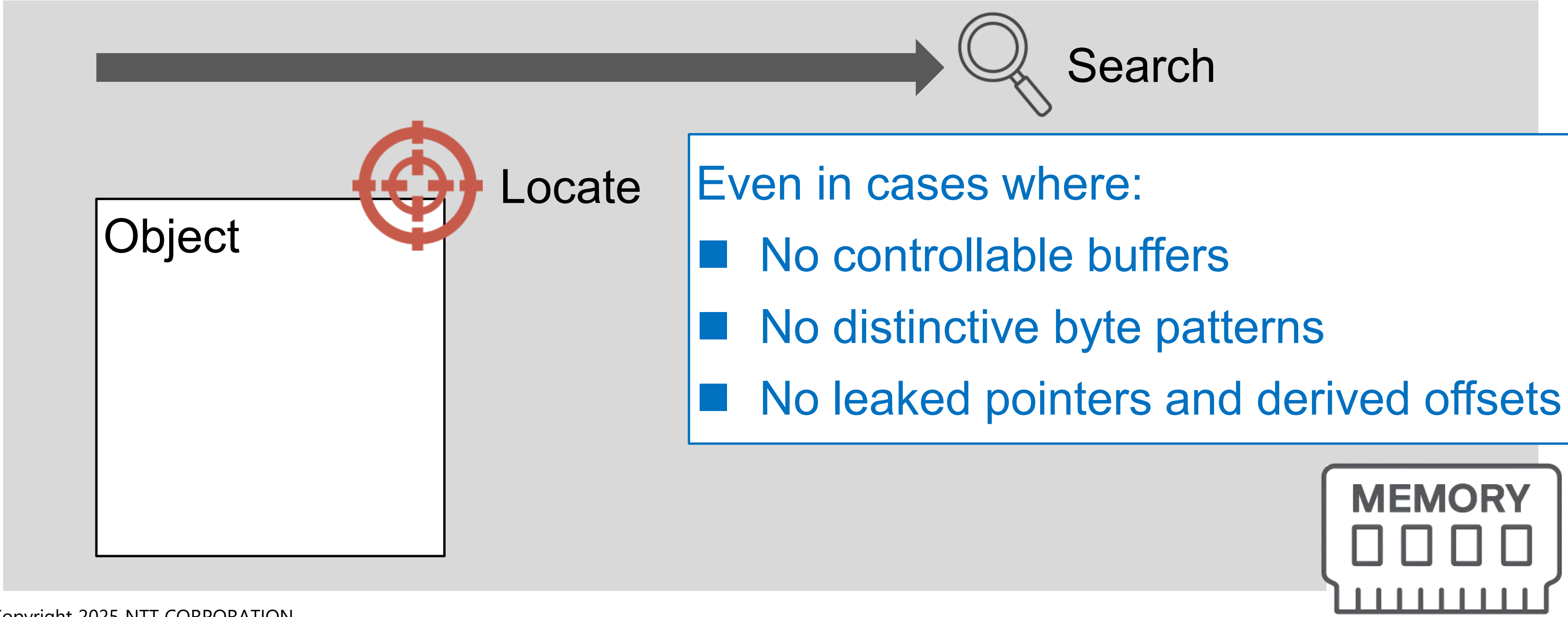
- ✅ use a controllable buffer
- ✅ find a distinctive value (egg)
- ✅ leak pointers
- ✅ calculate offsets to the object base

✅ Only needs **structural characteristics** of objects

Goal & Motivation

Goal

- To enable the identification of the memory locations of objects through memory space search



Even in cases where:

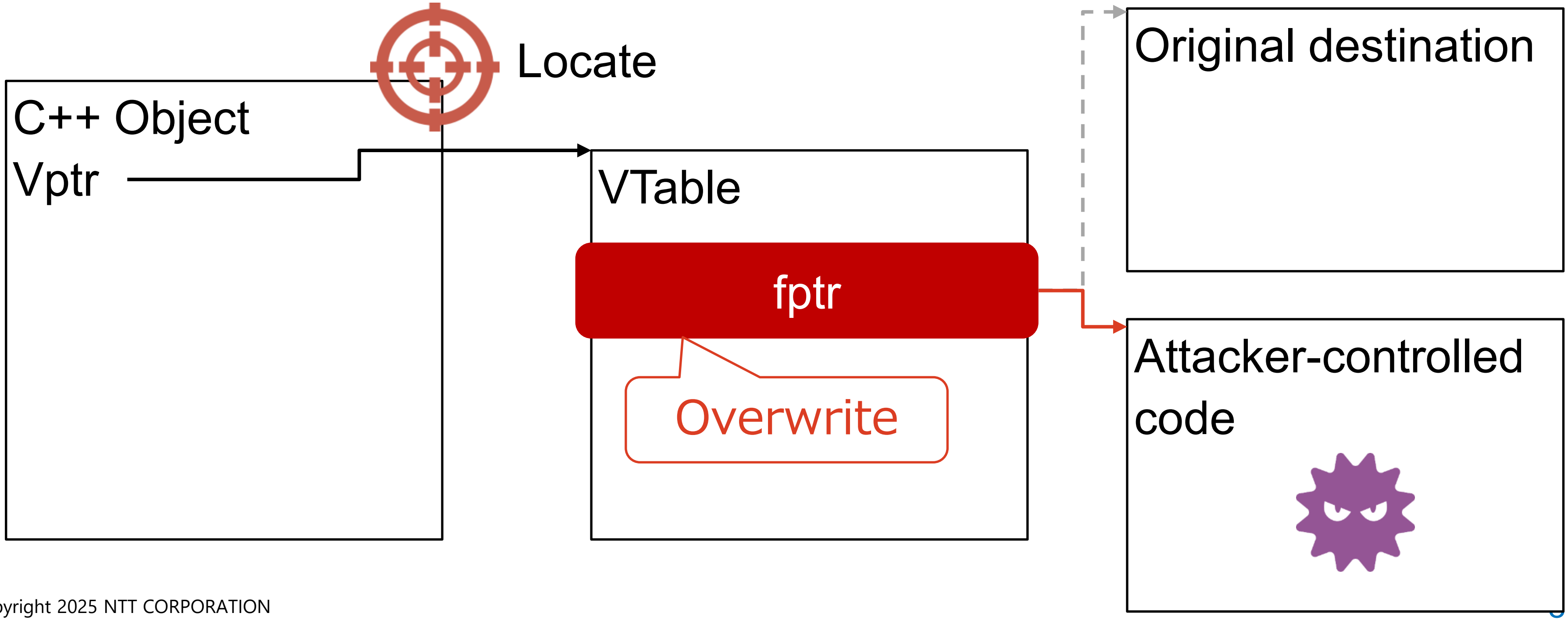
- No controllable buffers
- No distinctive byte patterns
- No leaked pointers and derived offsets

Motivations:
Applicability to Cybersecurity

- Exploit development
  - Control flow hijacking
  - Security policy modification and privilege escalation
- Memory forensics
- Malware-based injections

Example: Exploit Development

■ Control flow hijacking
- Locate and overwrite objects that contains function pointers
- e.g., VTable overwrite, File Stream Oriented Programming (FSOP)



Context for This Presentation

Context	**Exploit Development**
Goal	Arbitrary code execution (ACE)
Target binary	Locally available and freely analyzable
	Contains usable object for ACE
Assumed primitives	■ Arbitrary address read (AAR) ■ Arbitrary address write (AAW)
Approach	■ Locate object with AAR ■ Overwrite it with AAW → ACE

Existing Techniques & Limitations

Existing Techniques

- Scan-based approach
- Pointer-leak-and-offset-calculation-based approach
- ~~Symbol-based approach~~

This time,
we do not assume symbols

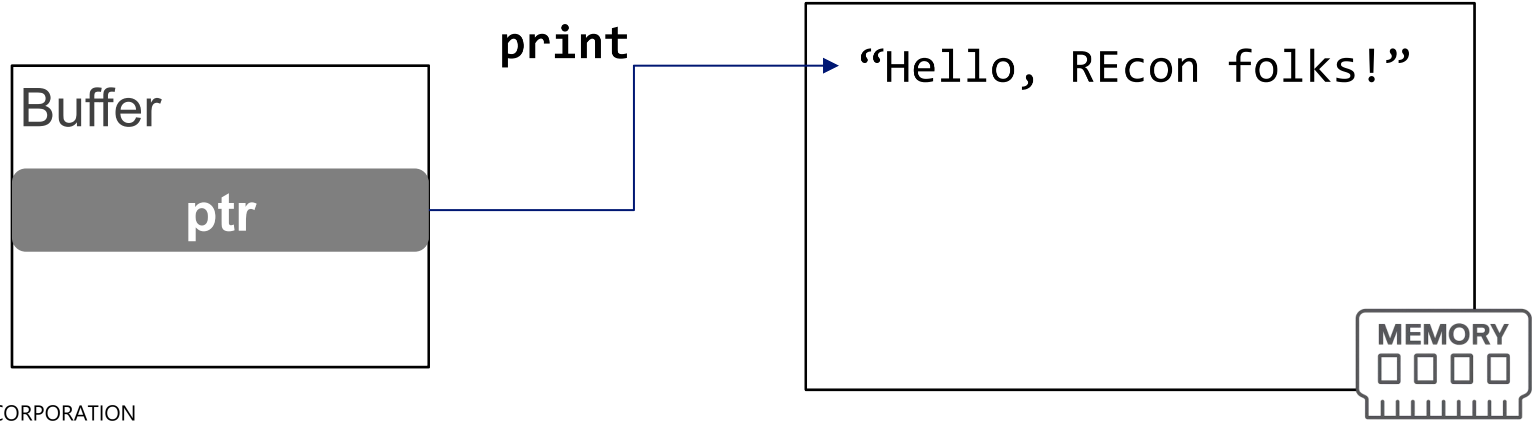
Underlying Technique:
Memory Disclosure Vulnerability

■ Vulnerabilities that allow unauthorized reading of memory contents

E.g., Use-After-Free (UAF)

```c
typedef struct {
    char *ptr;
    size_t val;
} Buffer;
```

```c
void print(Buffer *b) {
    printf("value: 0x%02x¥n", (unsigned char)b->ptr[0]);
}
```



Underlying Technique:
Memory Disclosure Vulnerability

NTT

■ Vulnerabilities that allow unauthorized reading of memory contents

E.g., Use-After-Free (UAF)

```c
typedef struct {
    char *ptr;
    size_t val;
} Buffer;
```

```c
void print(Buffer *b) {
    printf("value: 0x%02x¥n", (unsigned char)b->ptr[0]);
}
```

Freed

“Hello, REcon folks!”

MEMORY

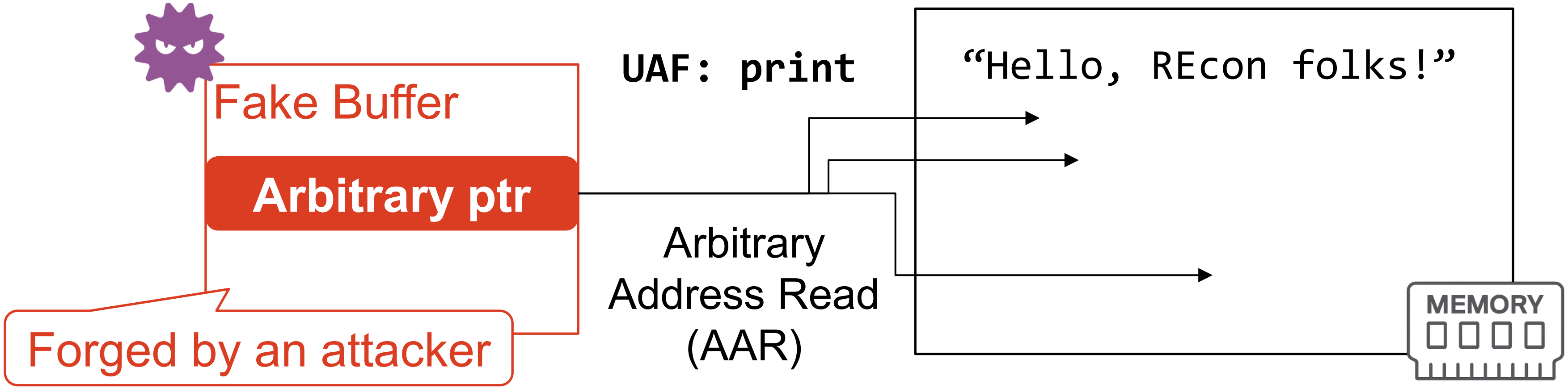
Underlying Technique:
Memory Disclosure Vulnerability

■ Vulnerabilities that allow unauthorized reading of memory contents

E.g., Use-After-Free (UAF)

```c
typedef struct {
    char *ptr;
    size_t val;
} Buffer;
```

```c
void print(Buffer *b) {
    printf("value: 0x%02x¥n", (unsigned char)b->ptr[0]);
}
```



Scan-based Approach

Enumerates memory and **scans it for distinctive byte patterns** to locate target variables

E.g., **Egg hunting** to locate embedded shellcode

```c
char target_buf[1024];
gets(target_buf);
```

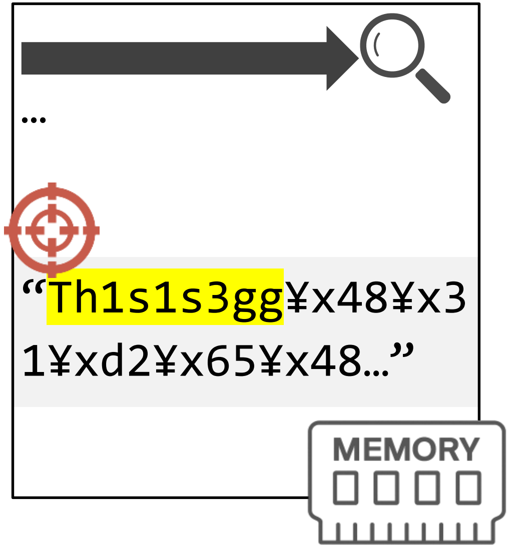
Controllable buffer

Embed distinctive byte patterns (egg)

target_buf:

"Th1s1s3gg¥x48¥x31¥xd2¥x65¥x48…"

Locate the buffer by scanning memory (e.g., with AAR)



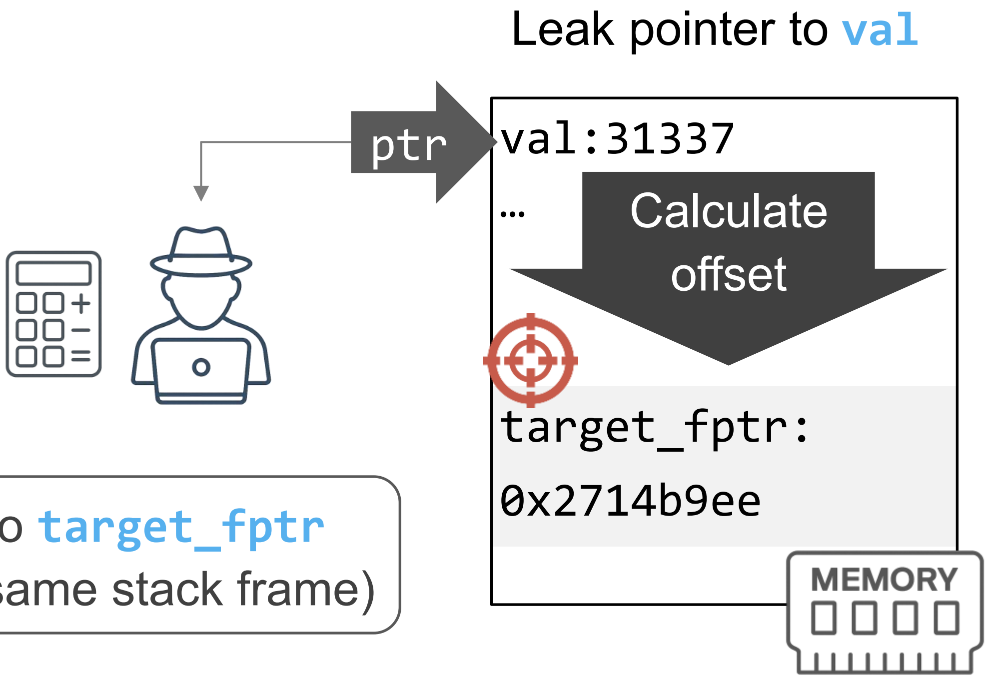
Offset-Calculation-based Approach

Leaking pointers from memory and **adding/subtracting offsets** to derive the target variable

E.g., **Info leak** to defeat ASLR

```c
void f() {
    void *target_fptr;
    int val;
    int *ptr = &val;
    …
```

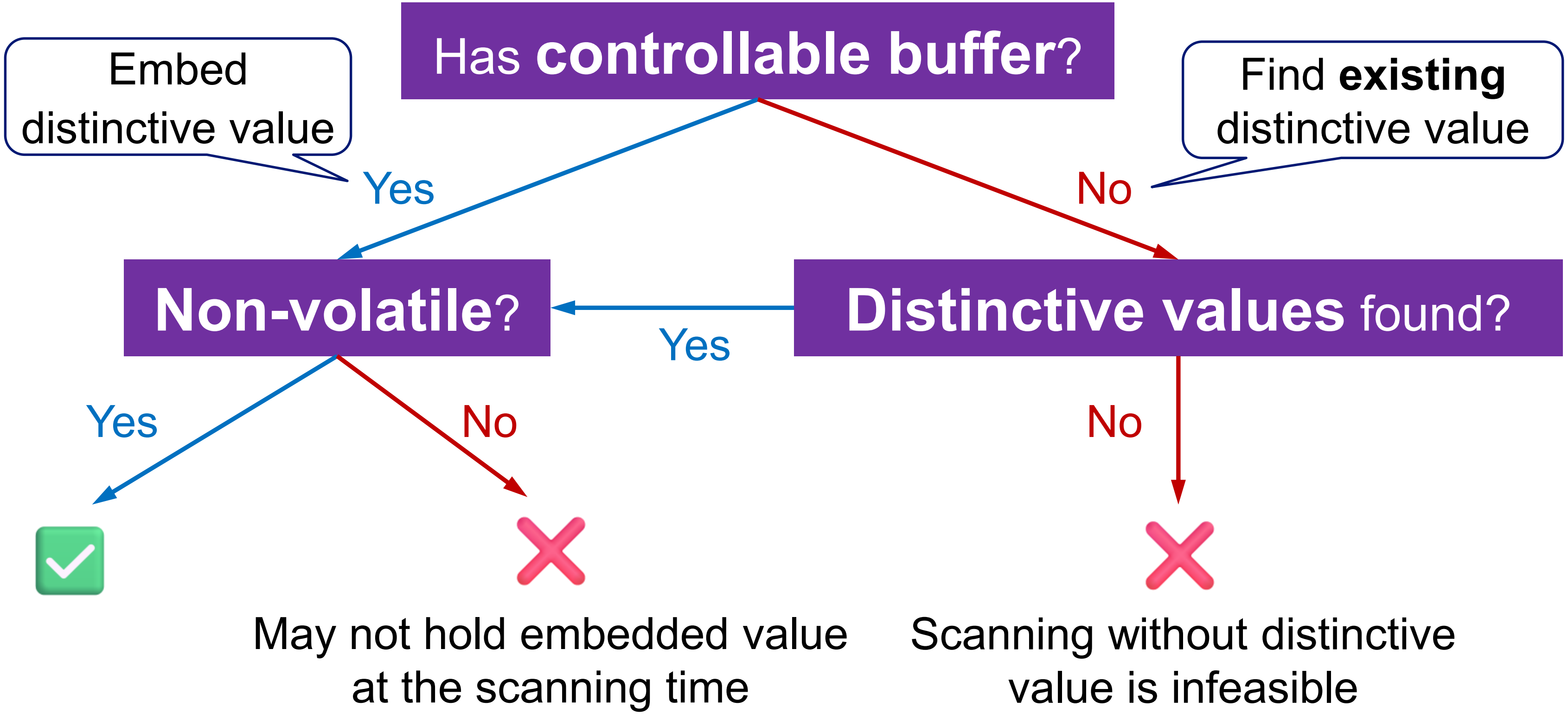
Offset from `val` to `target_fptr` calculable (due to same stack frame)



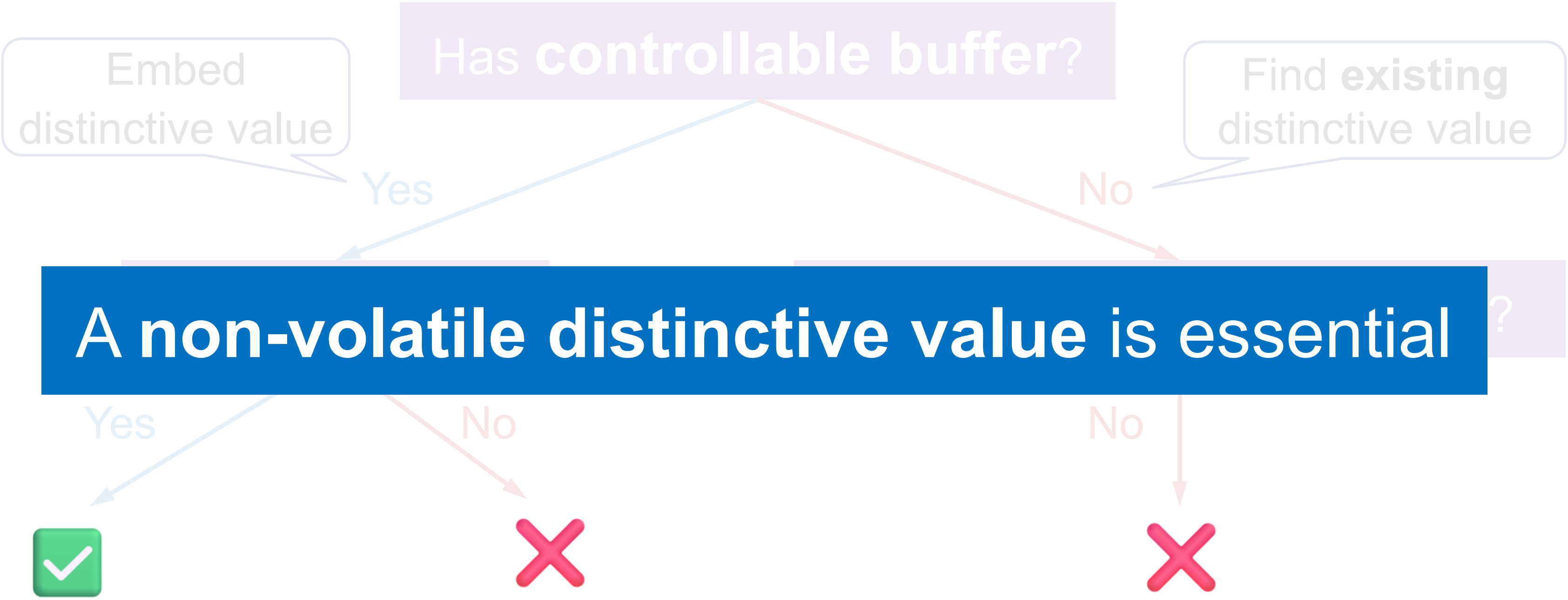
Question

Do **existing techniques** provide **sufficient information** to meet their needs?

Problems of Scan-based Approach



Problems of Scan-based Approach



A non-volatile distinctive value is essential

May not hold embedded value at the scanning time

Scanning without distinctive value is infeasible

Unscannable Values

Variable	Reason	
Pointers	❌ Volatile	Changes due to ASLR* across execution
Variables with transitive runtime values		May not hold expected values at the scanning time
Booleans	❌ Indistinctive	Too many identical byte patterns exist in memory
Small integers		
Variables ≤ 2 bytes		

* High-order bits are often invariant but insufficiently distinctive

Problem of Offset-Calculation-based Approach

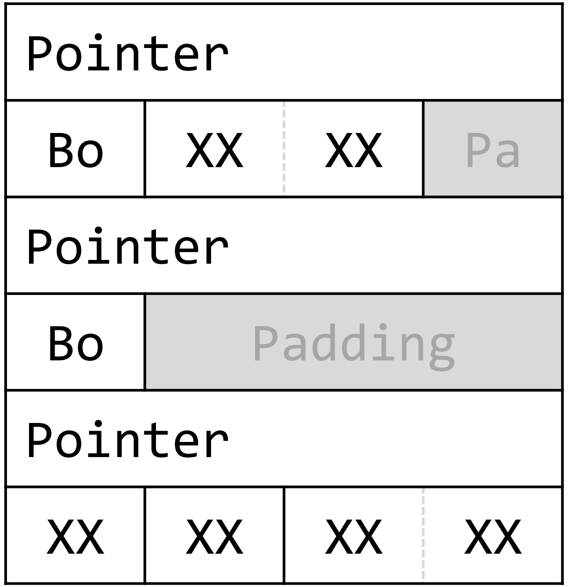
Scope	Pointer leakability	Offset computability	Locatability
Static	✅ Image base is leakable via scanning	✅ Offset from image base: invariant	✅
Stack	Partially ✅ Saved stack pointers are sometimes leakable	✅ Offsets in stack frames: invariant	Partially ✅
Heap	❌ Leaking the exact heap block of the target object is not likely	❌ Heap layouts vary across executions	❌

Problem Summary and Our Goal

Approach	Target Variable Scope			Distinctive values not required?
	Static	Stack	Heap	
Scan-based	✅	✅	✅	❌
Offset-calculation-based	✅	Partially ✅	❌	✅
Our goal	✅	✅	✅	✅

Key Idea: Structural Characteristics

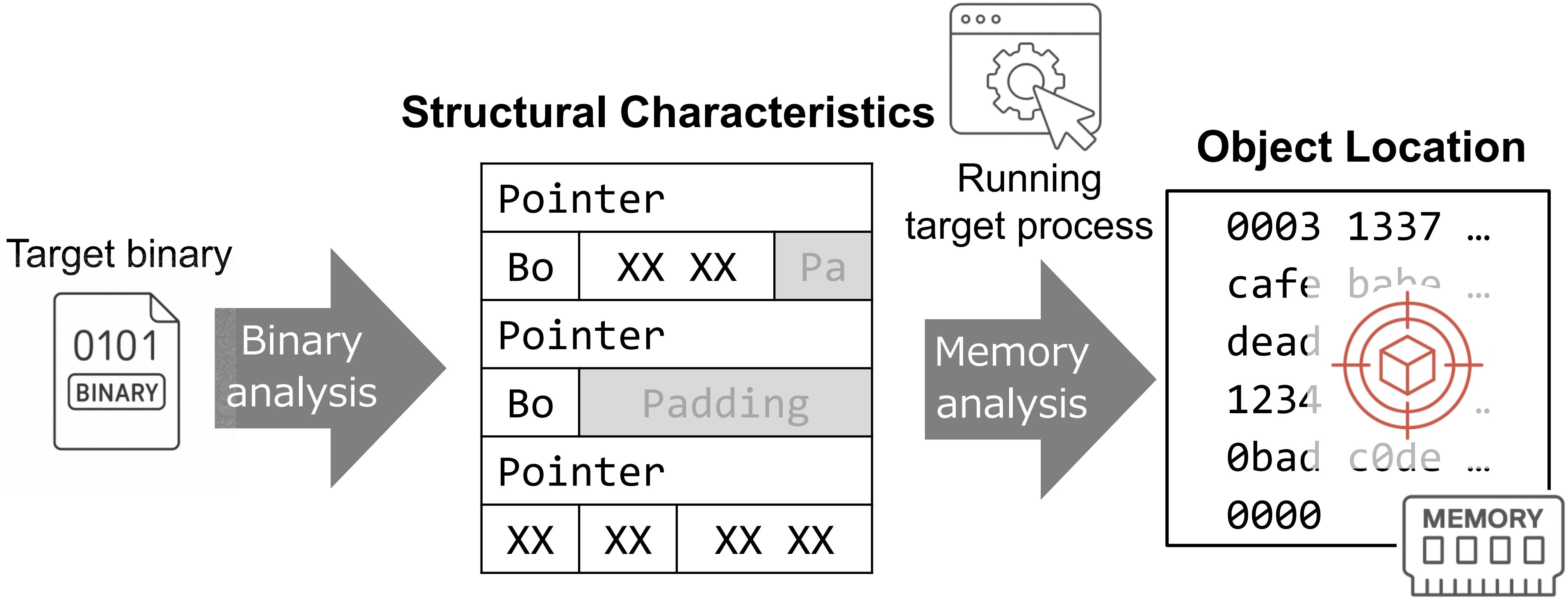
Indistinctive values not suitable for search can be made searchable by leveraging *structural characteristics*: **offsets**, **types**, and **sizes**



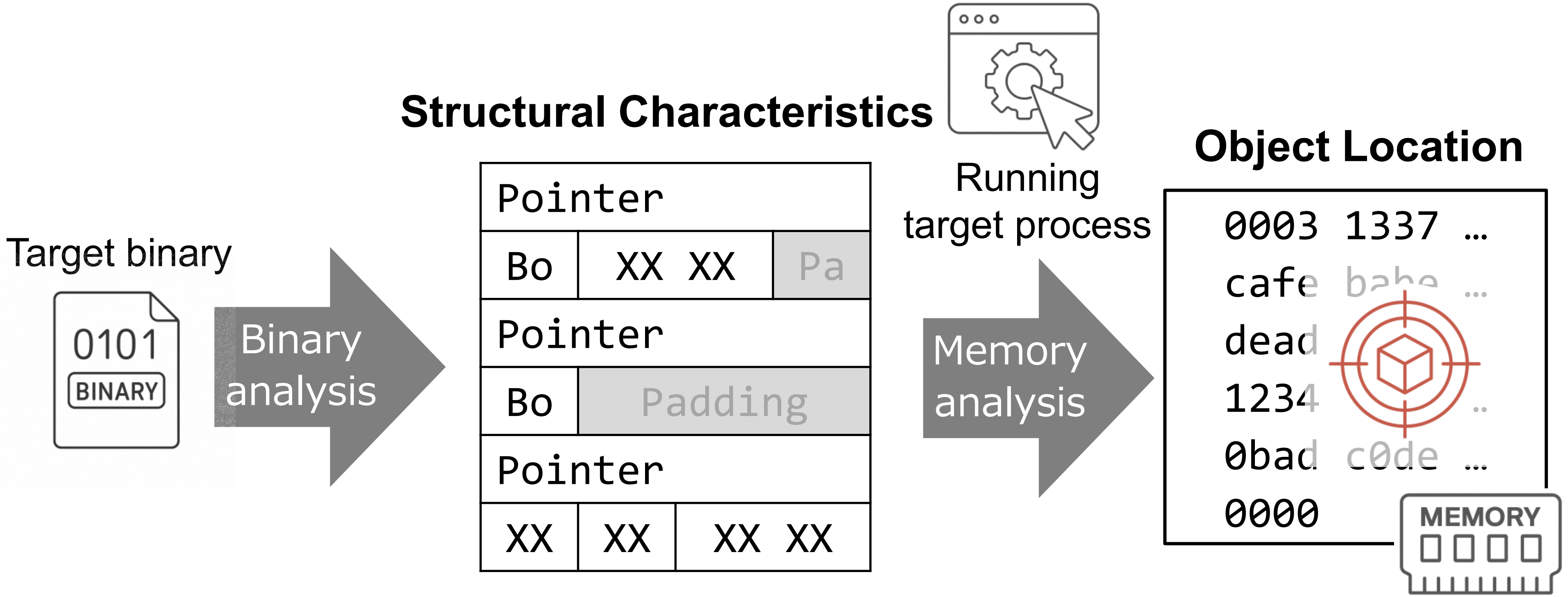
Example: Structural characteristics of an object

Type	Size	Offset
Pointer	4	0x0, 0x8, 0x10
Boolean	1	0x4, 0xC
Value	1	0x14, 0x15
	2	0x5, 0x16
Padding	-	0x7, 0xD

Key Idea: Structural Characteristics



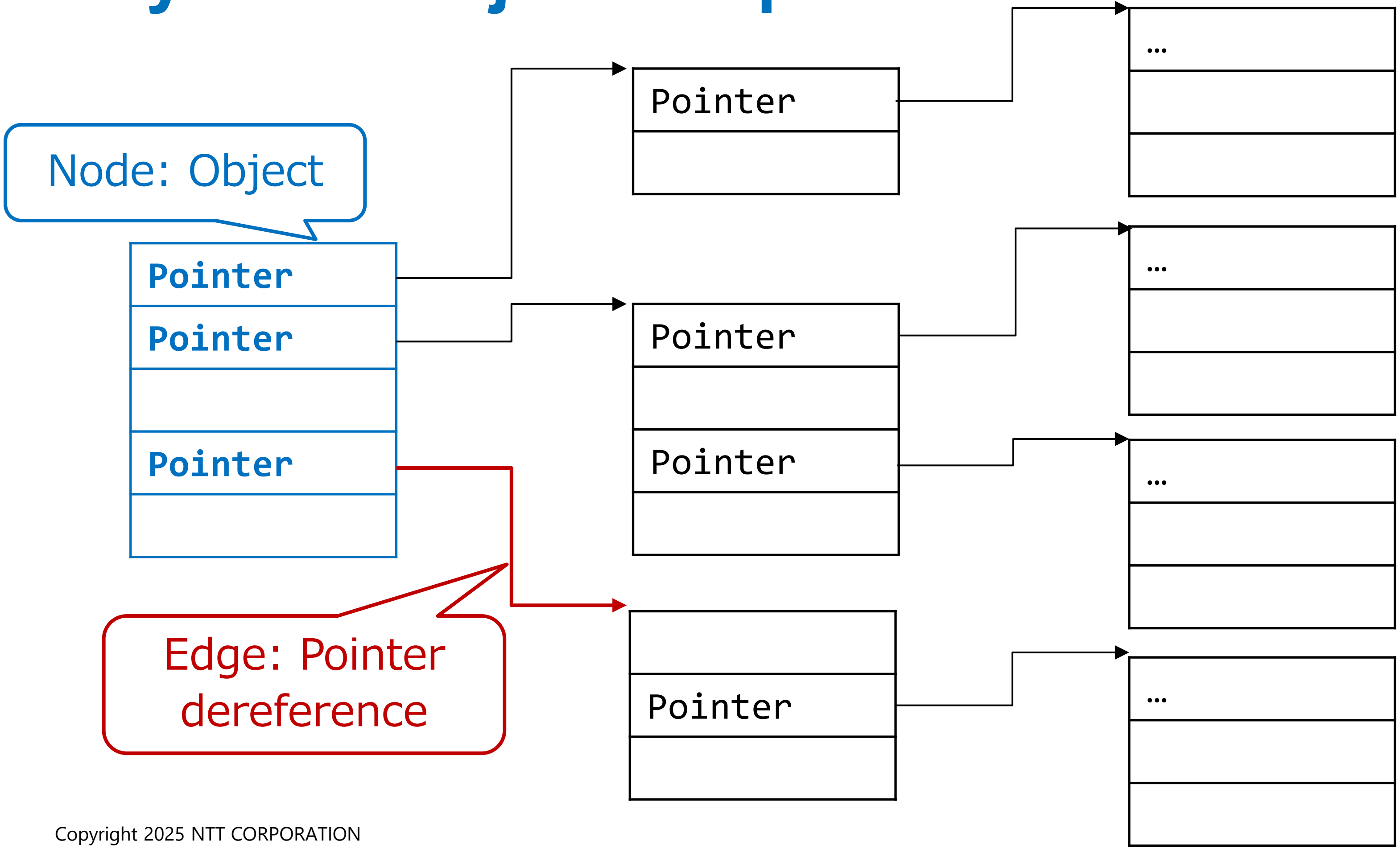
Key Idea: Structural Characteristics



But, is one object enough to locate?

Key Idea: Object Graph

Key Idea: Object Graph

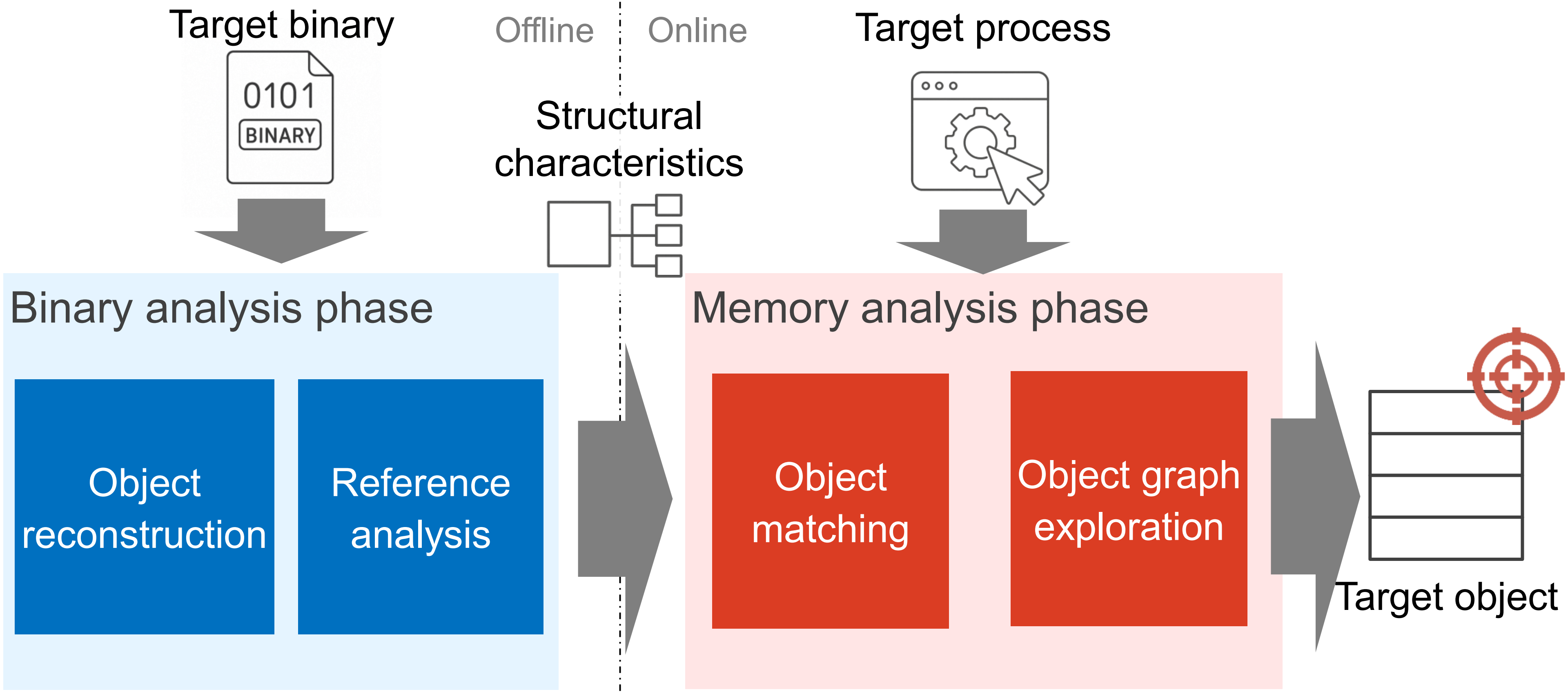


Requirements to Realize the Key Ideas

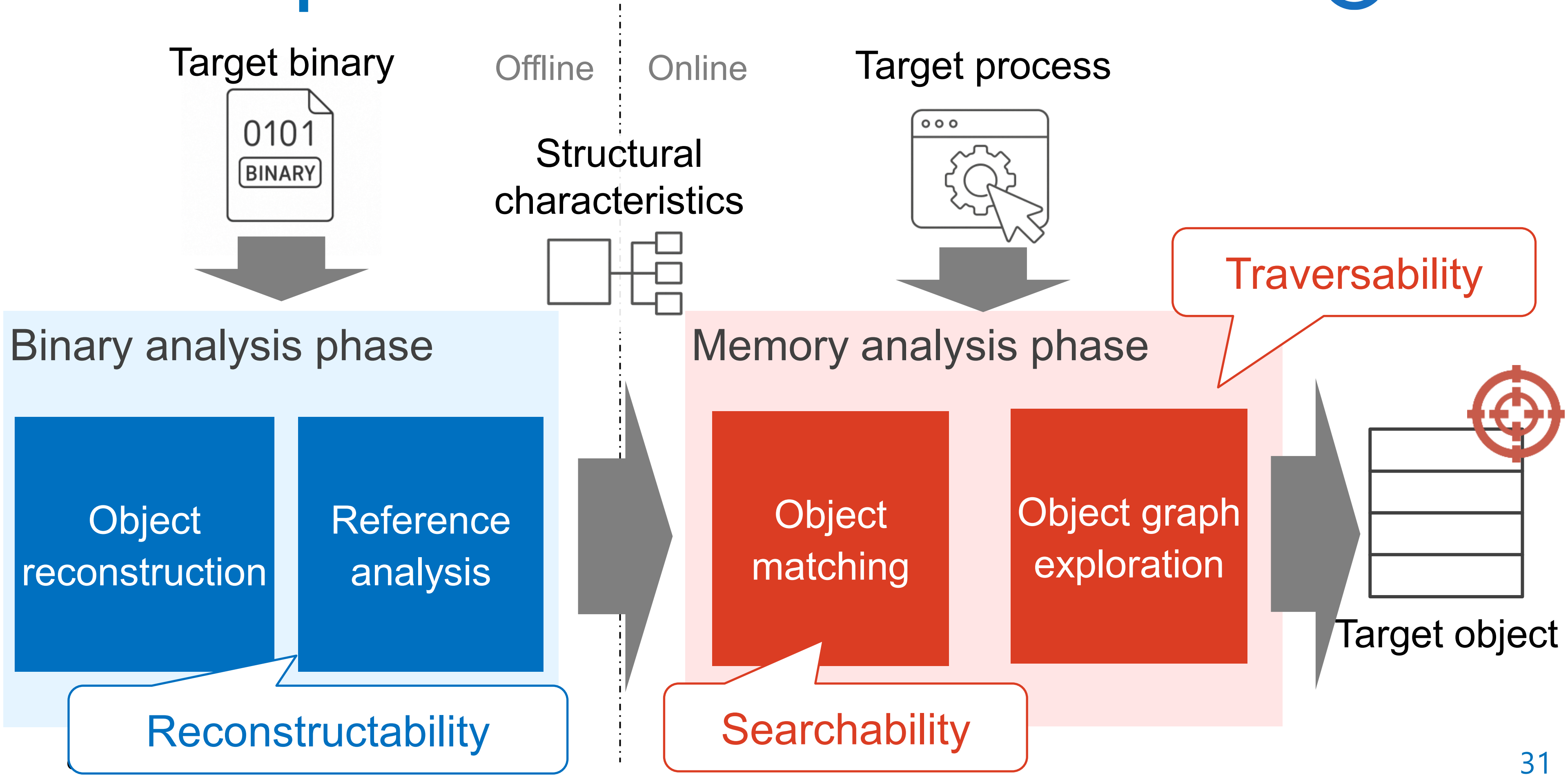
1. **Reconstructability**
   The structure of objects and object graphs must be reconstructible from the binary.
2. **Searchability**
   It must be possible to search memory for an object or object graph based on structure.
3. **Traversability**
   The object graph must be freely traversable to find the target object or target member variable.

Technique Overview

Technique Overview



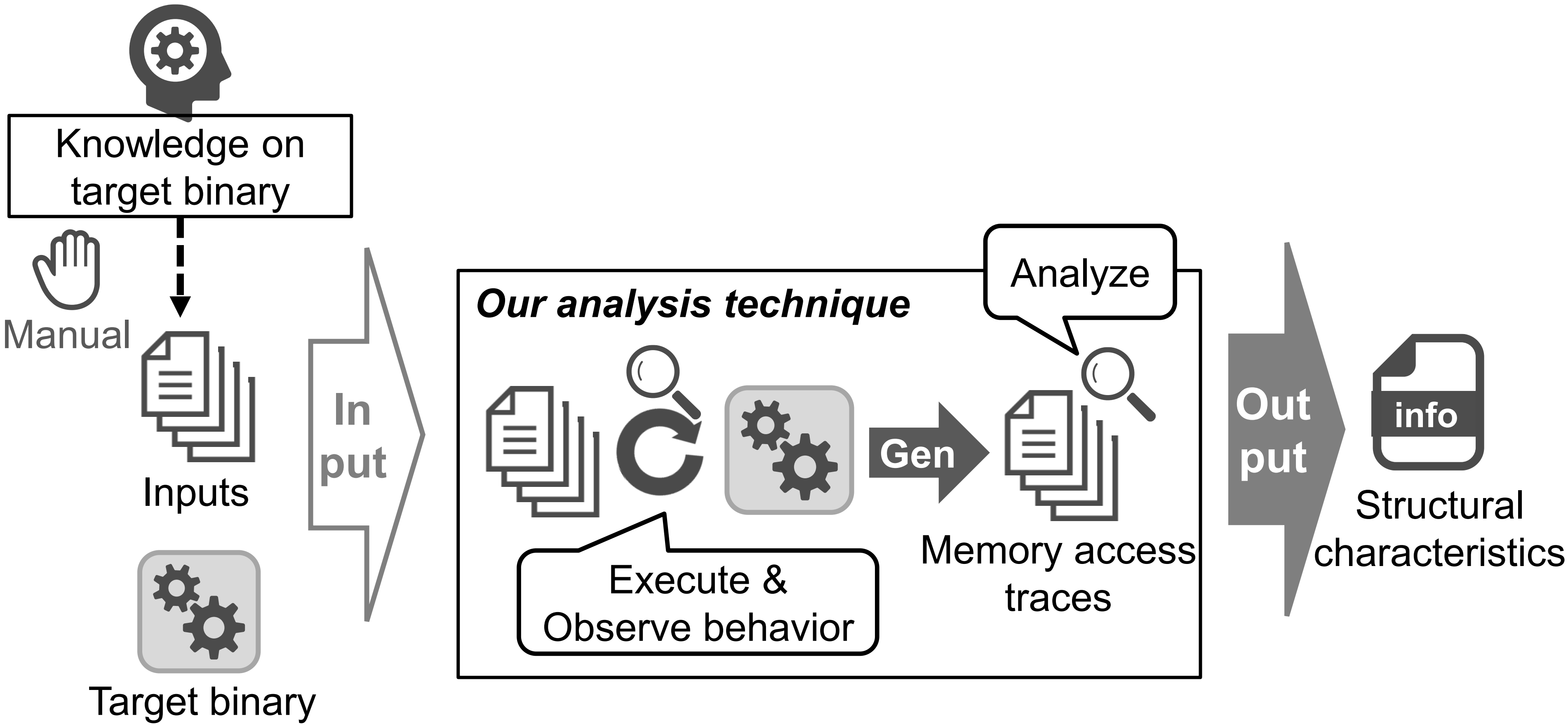
Technique Overview



Proposed Technique:

Binary Analysis Phase

Overview of Binary Analysis



Memory Access Instruction Monitoring

Executed memory access instruction

Index register

Displacement

```
0x7ffb0c1ccad2:  mov rax, [ rbx + rsi * 8 + 0x10 ]
```

Base register

Logging via instruction monitoring

Corresponding log record

```
type: read, ip: 0x7ffb0c1ccad2, target: 0x15e5ea7b5a8,
base: 0x15e5ea7b588, index: 2, disp: 0x10,
size: 8, value: 0x0000015e5ea7c010
```

Memory Access Instruction Monitoring

Executed memory access instruction

Index register

Displacement

```
0x7ffb0c1ccad2:  mov rax, [ rbx + rsi * 8 + 0x10 ]
```

Dynamic Binary Instrumentation (DBI)
is your friend!

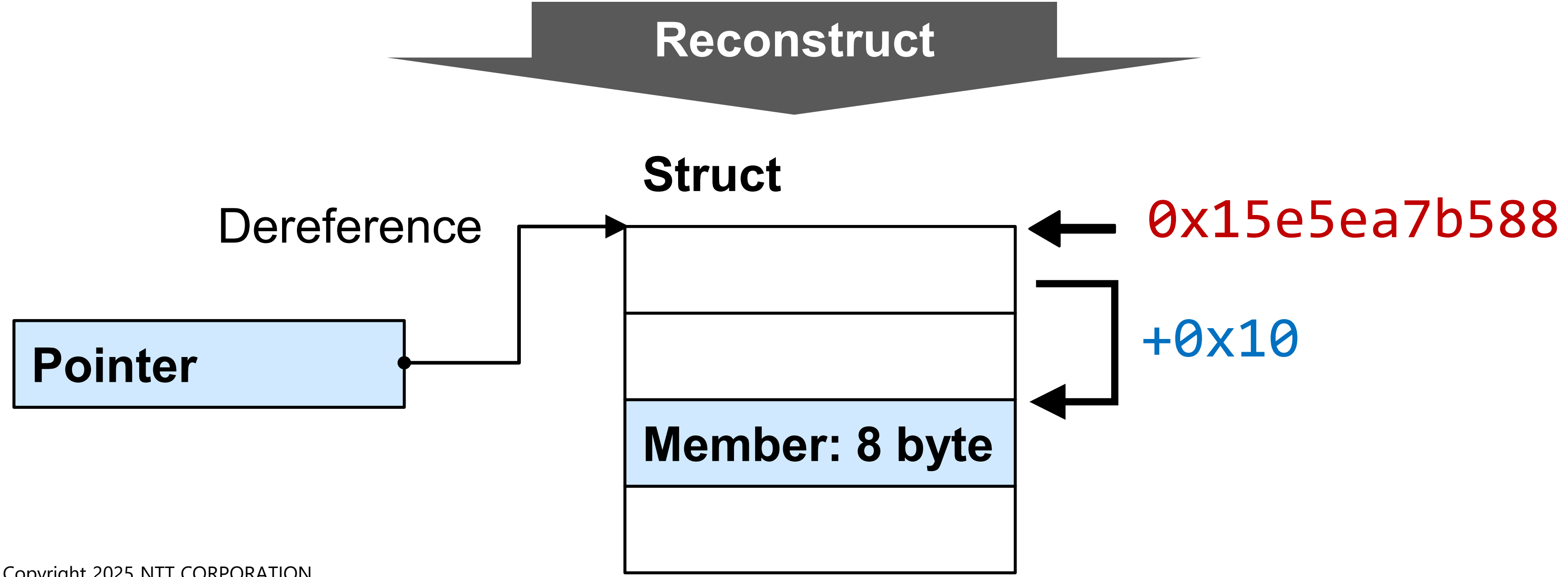
Corresponding log record

```
type: read, ip: 0x7ffb0c1ccad2, target: 0x15e5ea7b5a8,
base: 0x15e5ea7b588, index: 2, disp: 0x10,
size: 8, value: 0x0000015e5ea7c010
```

Structure Reconstruction

A memory access record to a single struct member

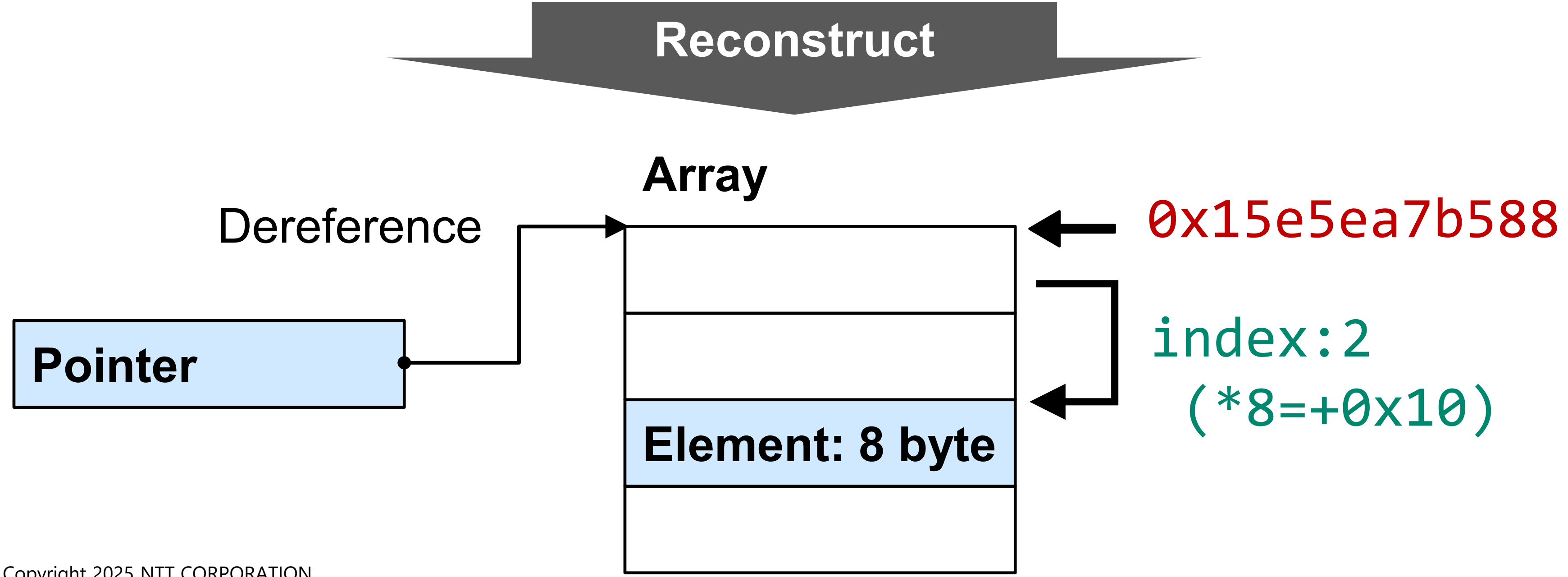
```
…, base: 0x15e5ea7b588, disp: 0x10, size: 8, …
```



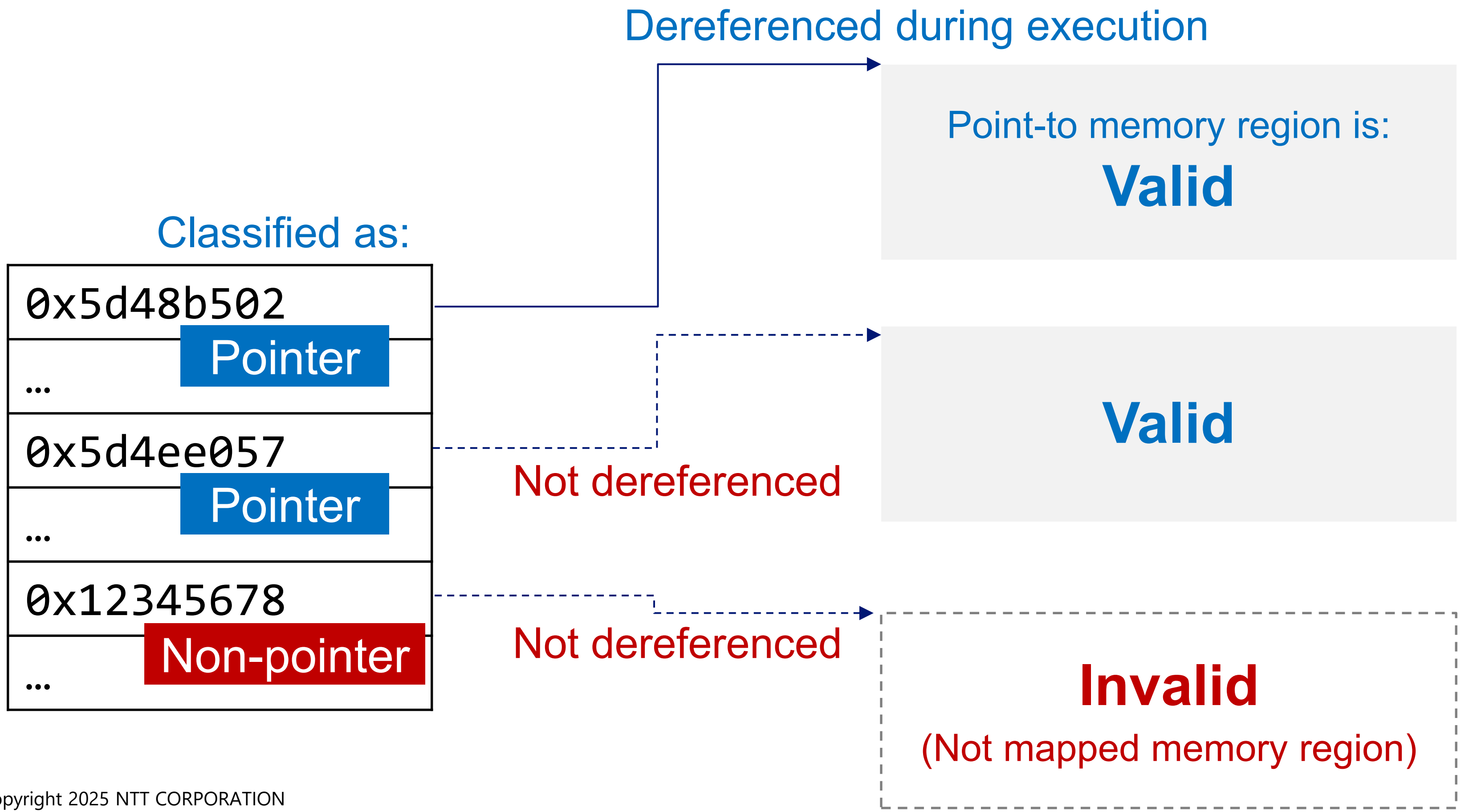
Array Reconstruction

A memory access record to a single array element

```
…, base: 0x15e5ea7b588, index: 2, size: 8, …
```



Pointer Inference



Boolean Inference

Used as a Boolean at runtime

```
mov BYTE PTR bx, [rax]
cmp bx, cx
jnz …
```

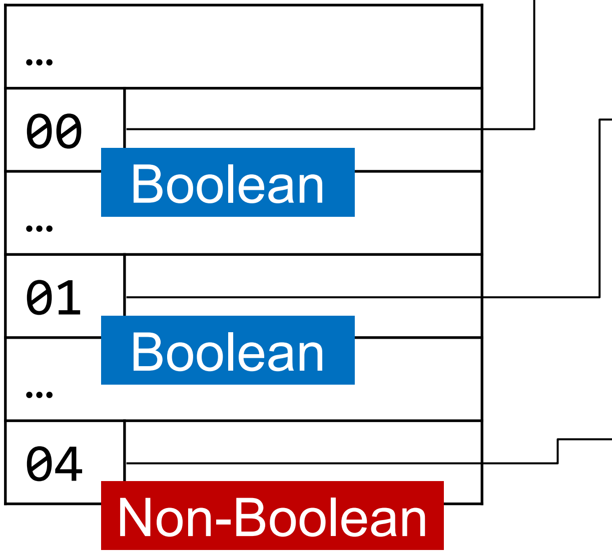
Values valid for a Boolean type are taken during execution

```
…, size: 1, value: 0x0
…, size: 1, value: 0xff
…, size: 1, value: 0x0
```

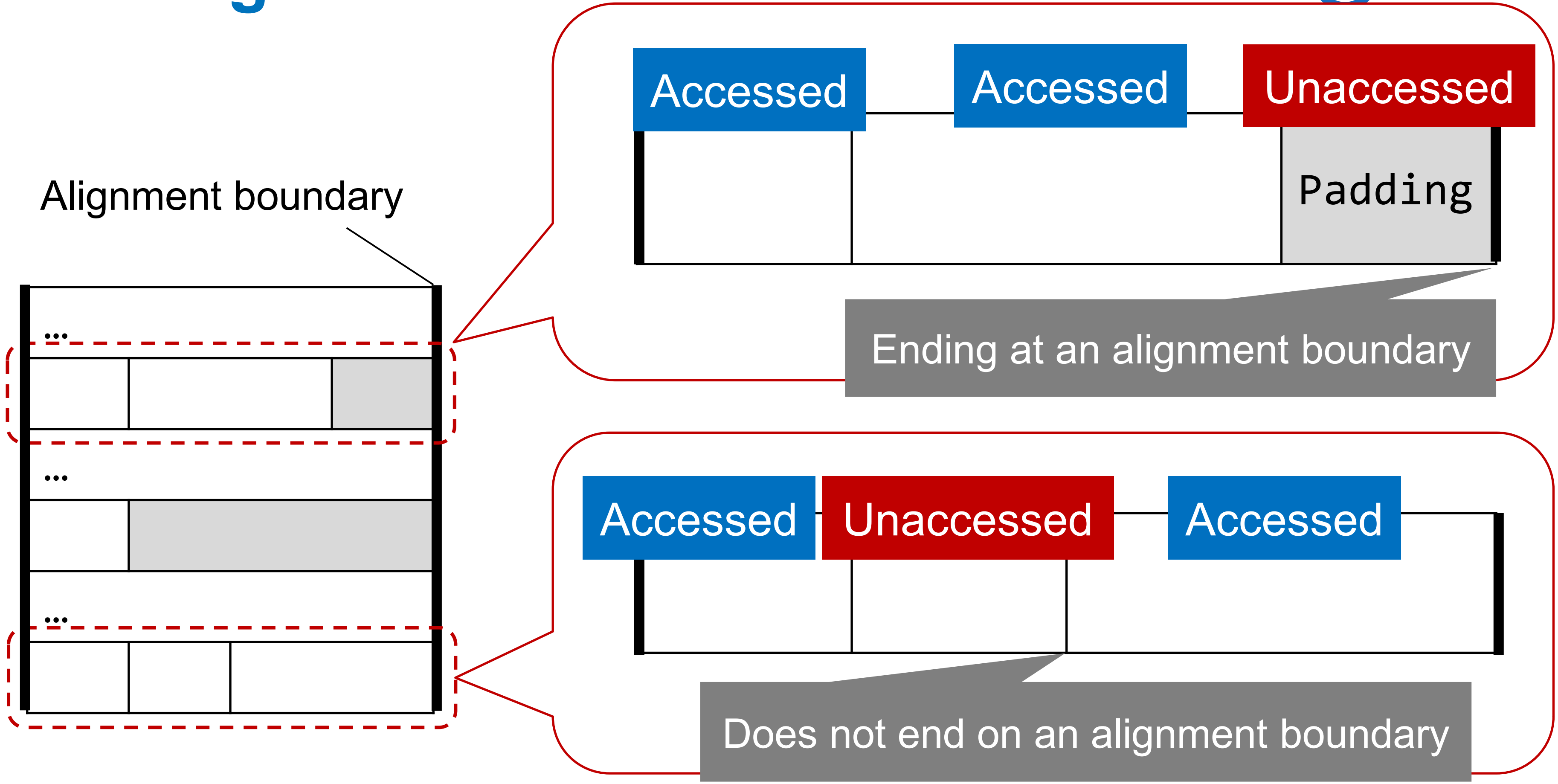
Values invalid for a Boolean type are taken

```
…, size: 1, value: 0x4
…, size: 1, value: 0x5
…, size: 1, value: 0x8
```

Classified as:



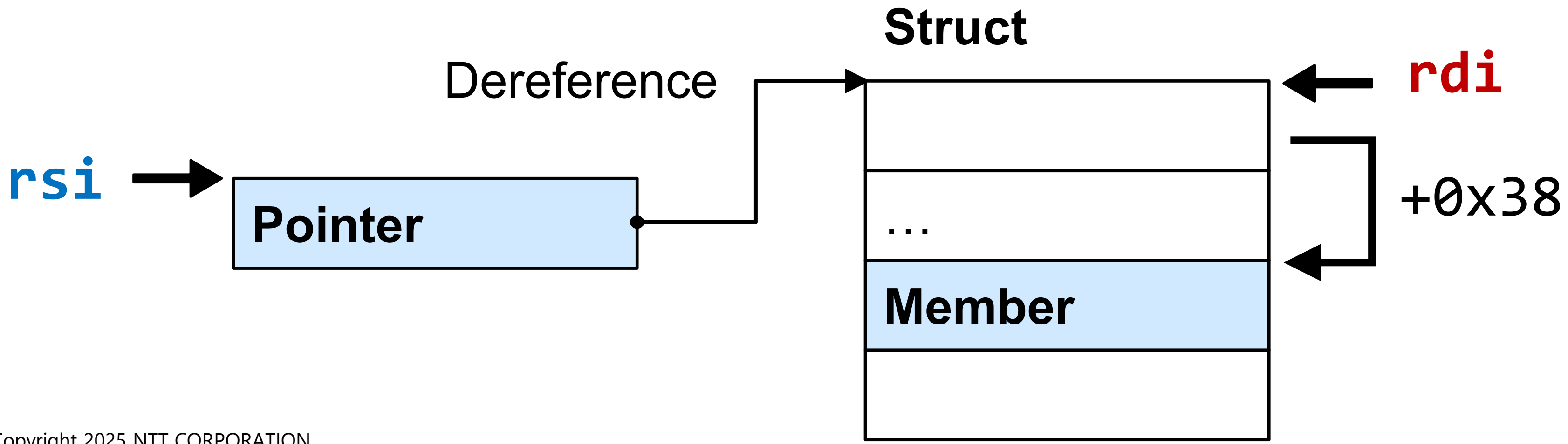
Padding Inference



Pointer Dereference Tracking

A memory access instruction pattern to a single struct member

```
mov rdi, [ rsi ]  // Read from a pointer
…
mov rax, [ rdi + 0x38 ]  // Used as base
```



Pointer Dereference Tracking

```
mov rdi, [ rsi ] // Read from a pointer to RDI
…
mov rax, [ rdi + 0x38 ]  // Used as base
```

Generate

```
type: read, …, target: 0x15e5eb837e0, value: 0x15e5ea7b588
…
type: read, …, target: 0x15e5ea7b5c0, base: 0x15e5ea7b588,
disp: 0x38, …
```

Pointer Dereference Tracking

```
mov rdi, [ rsi ] // Read from a pointer to RDI
…
mov rax, [ rdi + 0x38 ]  // Used as base
```

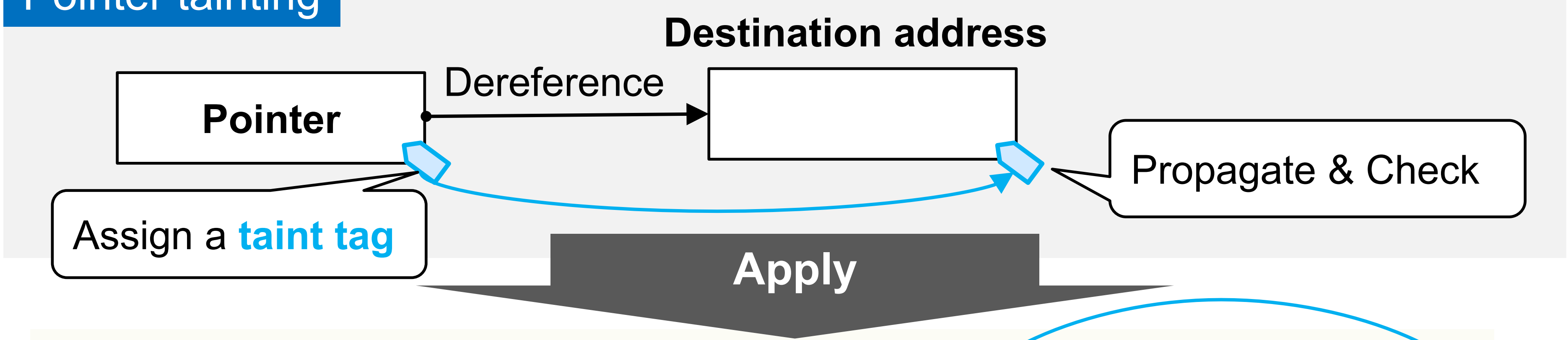
Generate

```
type: read, …, target: 0x15e5eb837e0, value: 0x15e5ea7b588
…
type: read, …, target: 0x15e5ea7b5c0, base: 0x15e5ea7b588,
disp: 0x38, …
```

Can we prove the **correlation** between these two log entries?

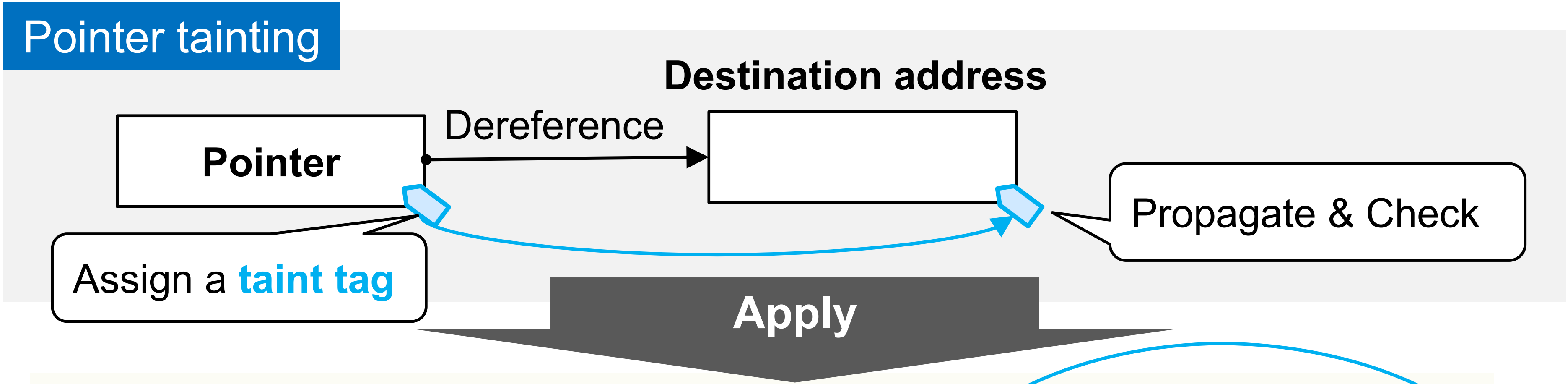
Pointer Dereference Tracking

Pointer tainting




```
type: read, …, target: 0x15e5eb837e0, value: 0x15e5ea7b588
…
type: read, …, target: 0x15e5ea7b5c0, base: 0x15e5ea7b588,
disp: 0x38, …
```

Pointer Dereference Tracking

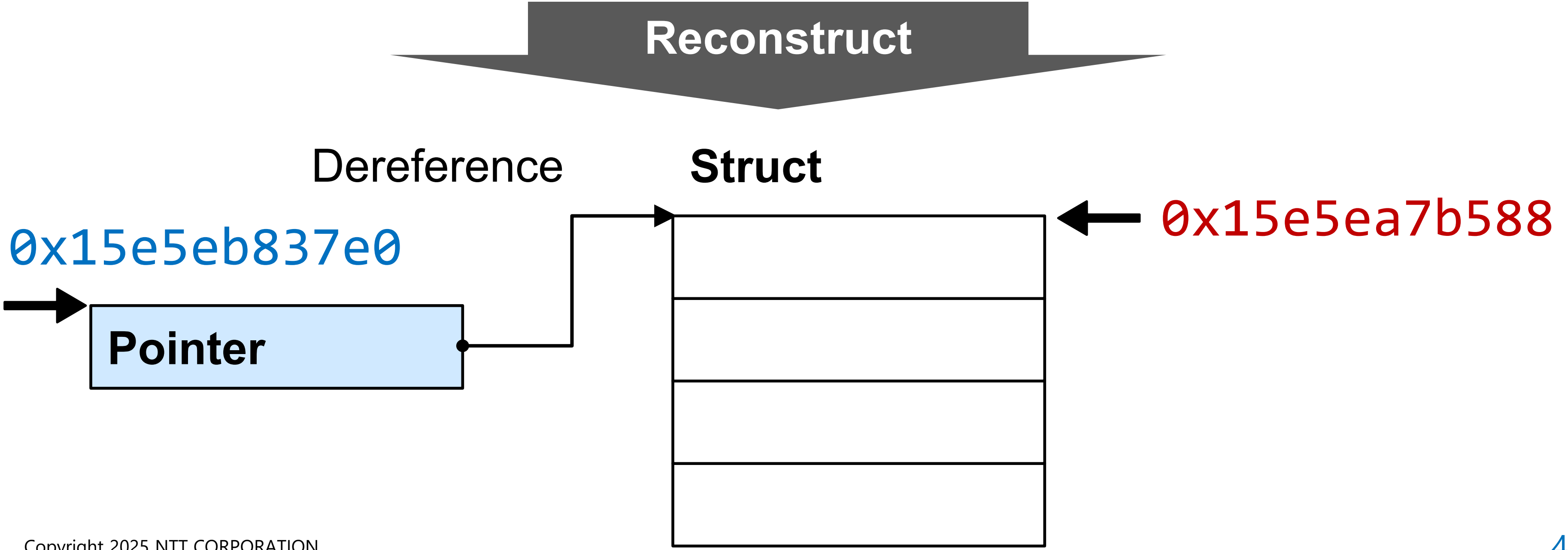



```
type: read, …, target: 0x15e5eb837e0, value: 0x15e5ea7b588
…
type: read, …, target: 0x15e5ea7b5c0, base: 0x15e5ea7b588,
disp: 0x38, …
```

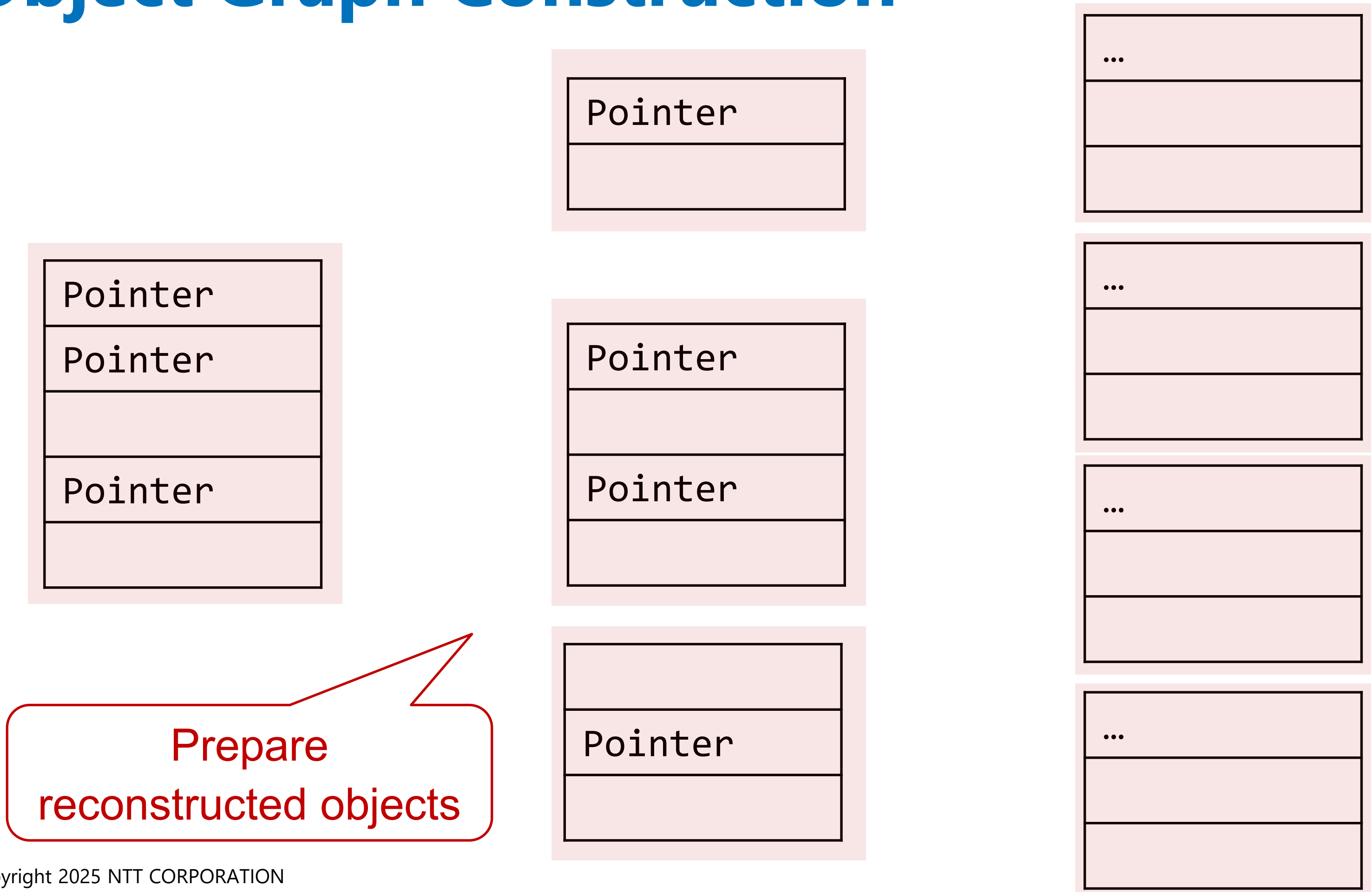
Reference Analysis

A pointer dereference record to a single struct

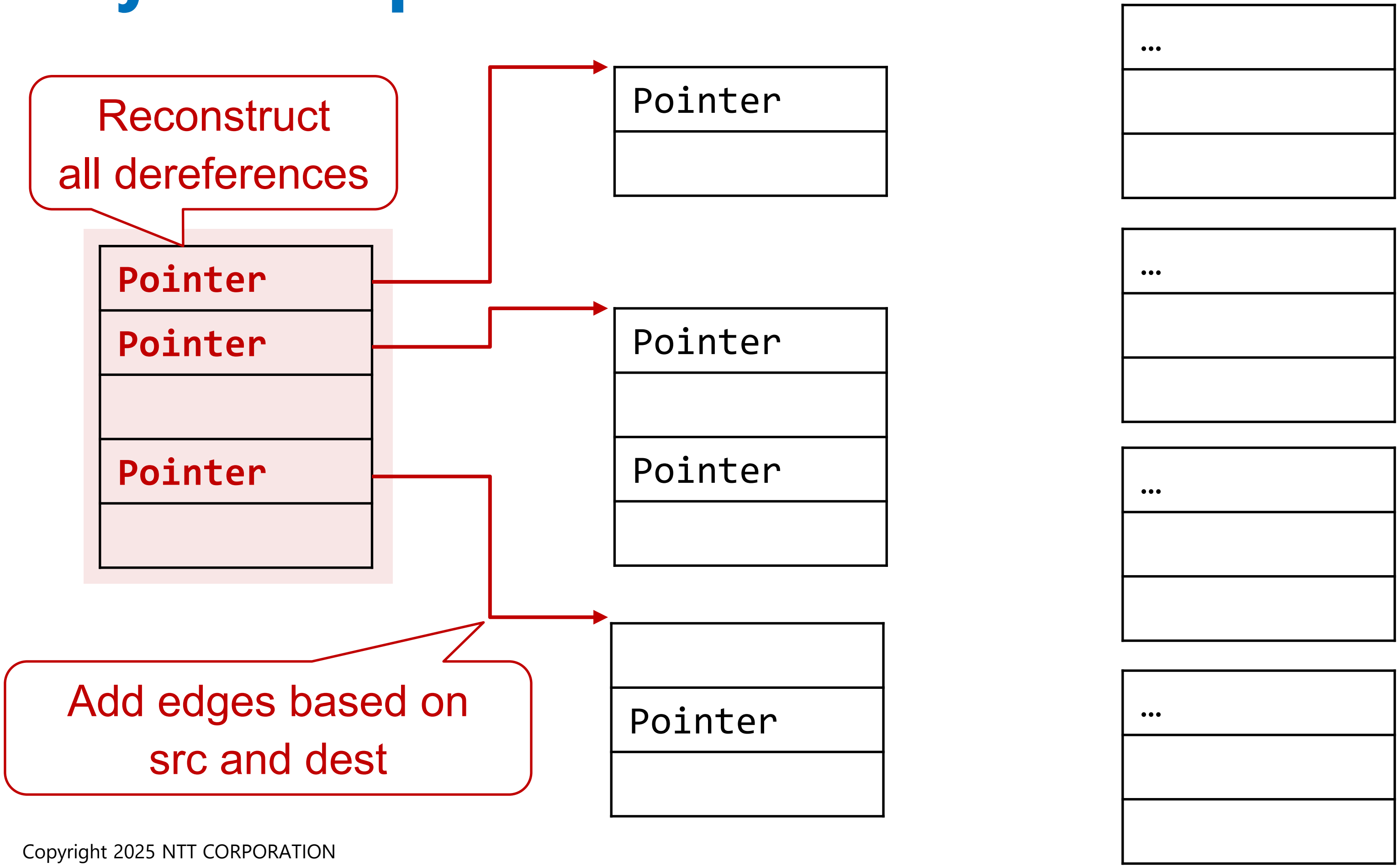
```
src: 0x15e5eb837e0, dst: 0x15e5ea7b588
```



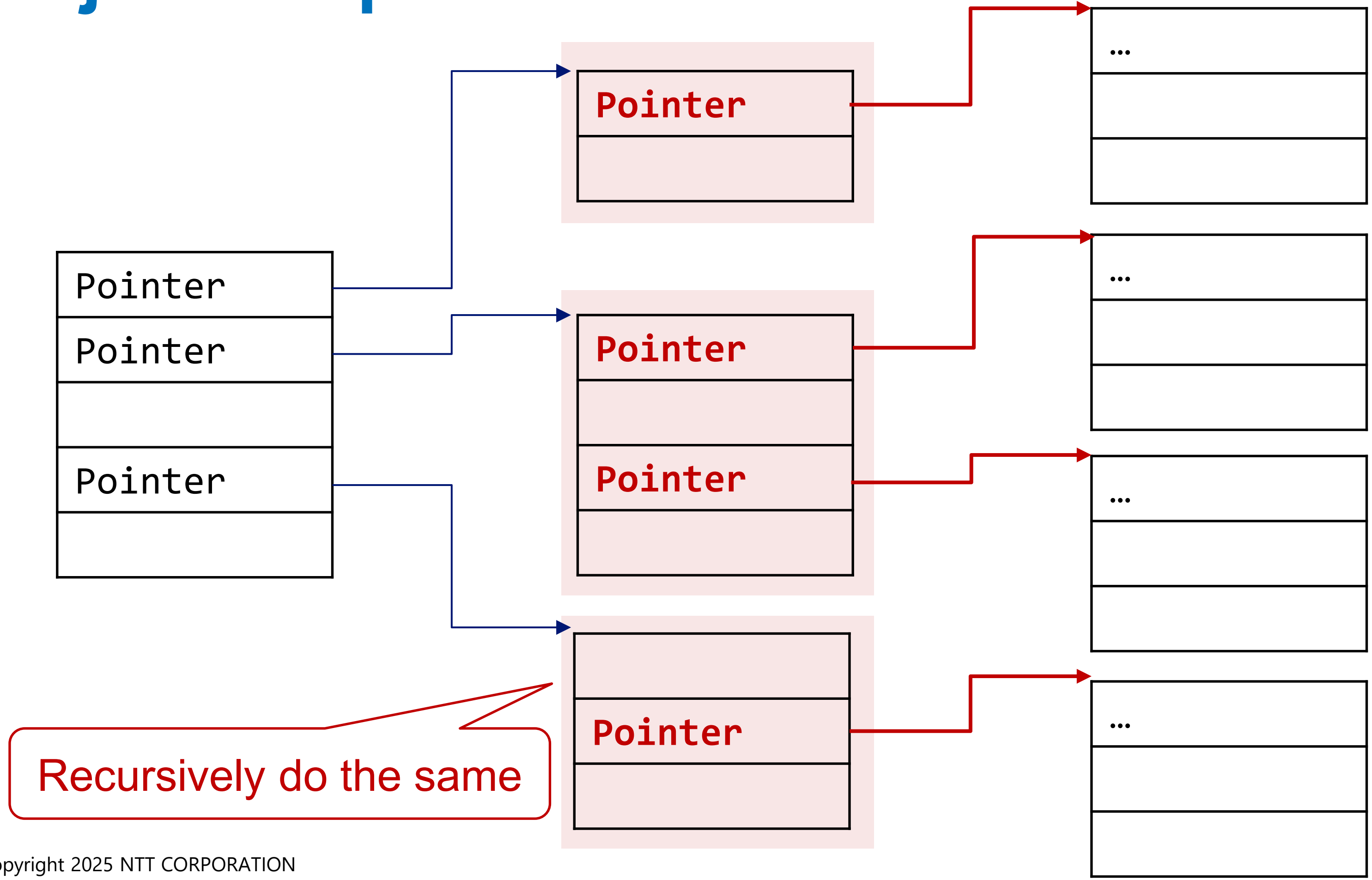
Object Graph Construction



Object Graph Construction

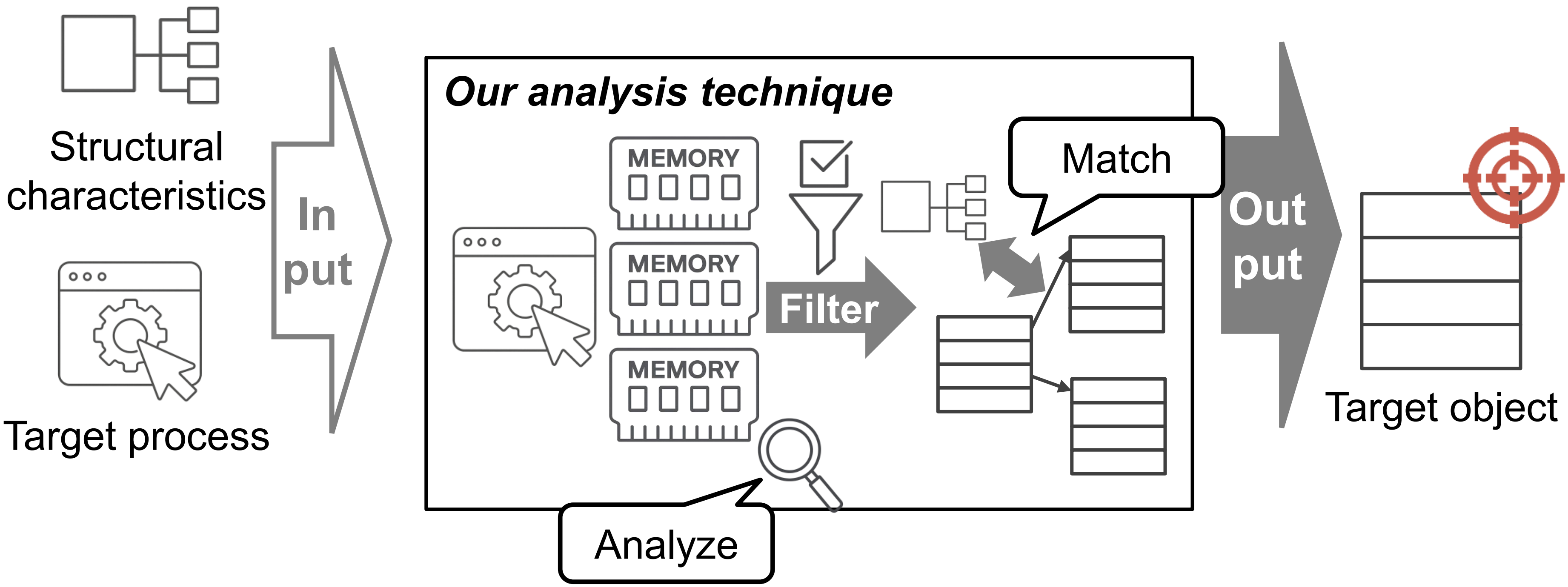


Object Graph Verification



Proposed Technique:
Memory Analysis Phase

Overview of Memory Analysis



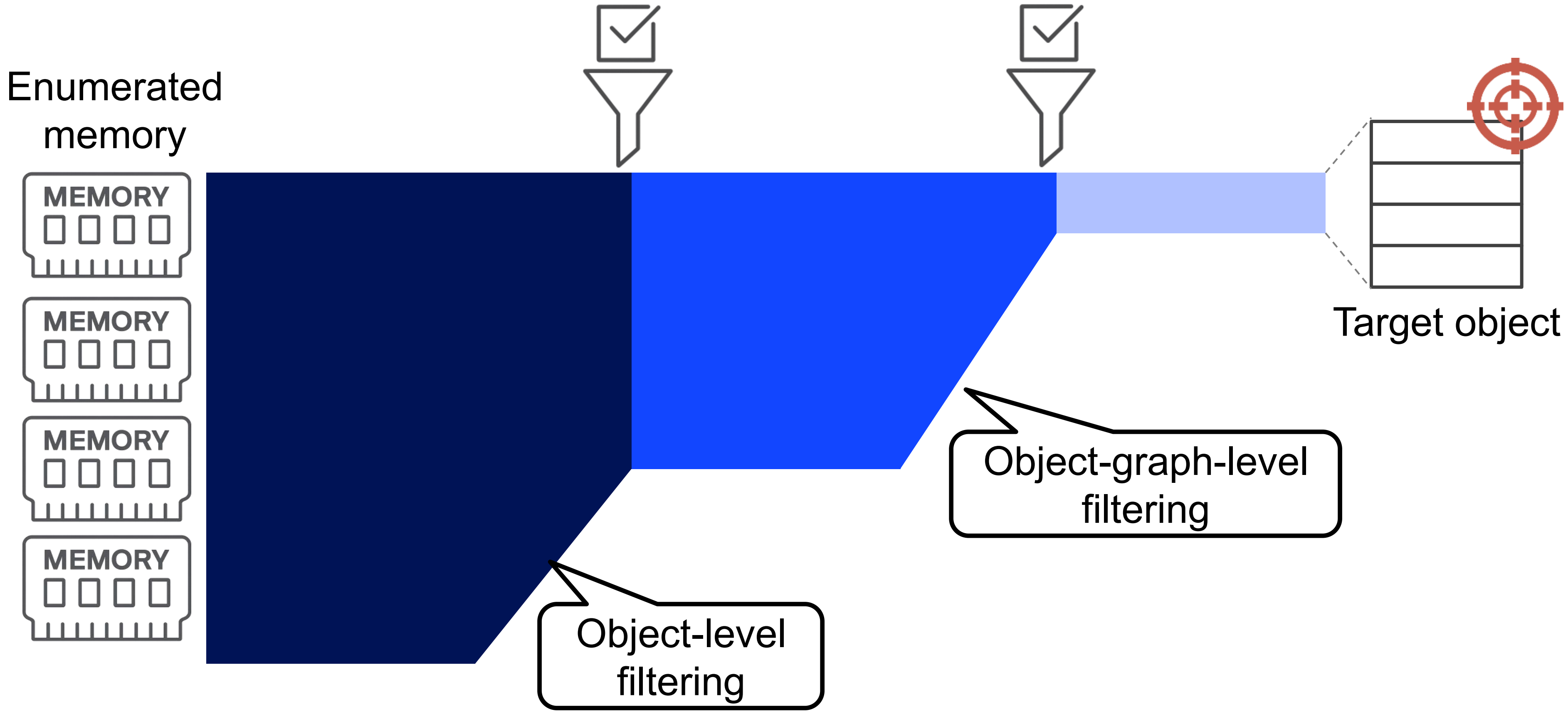
Memory Enumeration

- First, enumerate all memory regions including: **static**, **stack**, and **heap**

- This can be achieved through:

Situation	Measure
Exploit Development	Arbitrary address read (AAR) primitive
Code injection	System APIs
Memory forensics	Memory acquisition & analysis on memory dumps

Overview of Filtering Process



Pointer Enumeration

① Clustering byte sequences of pointer size

```
0x000015e5ea7bf88
0x000015e5ea7bf90
…
0x000015e5eb067e0

0x00002780c3d5300
0x00002780c3d5310
…
0x00002780c51acff
```

② Extract frequent high bytes as address ranges
& Use it to determine pointer candidates

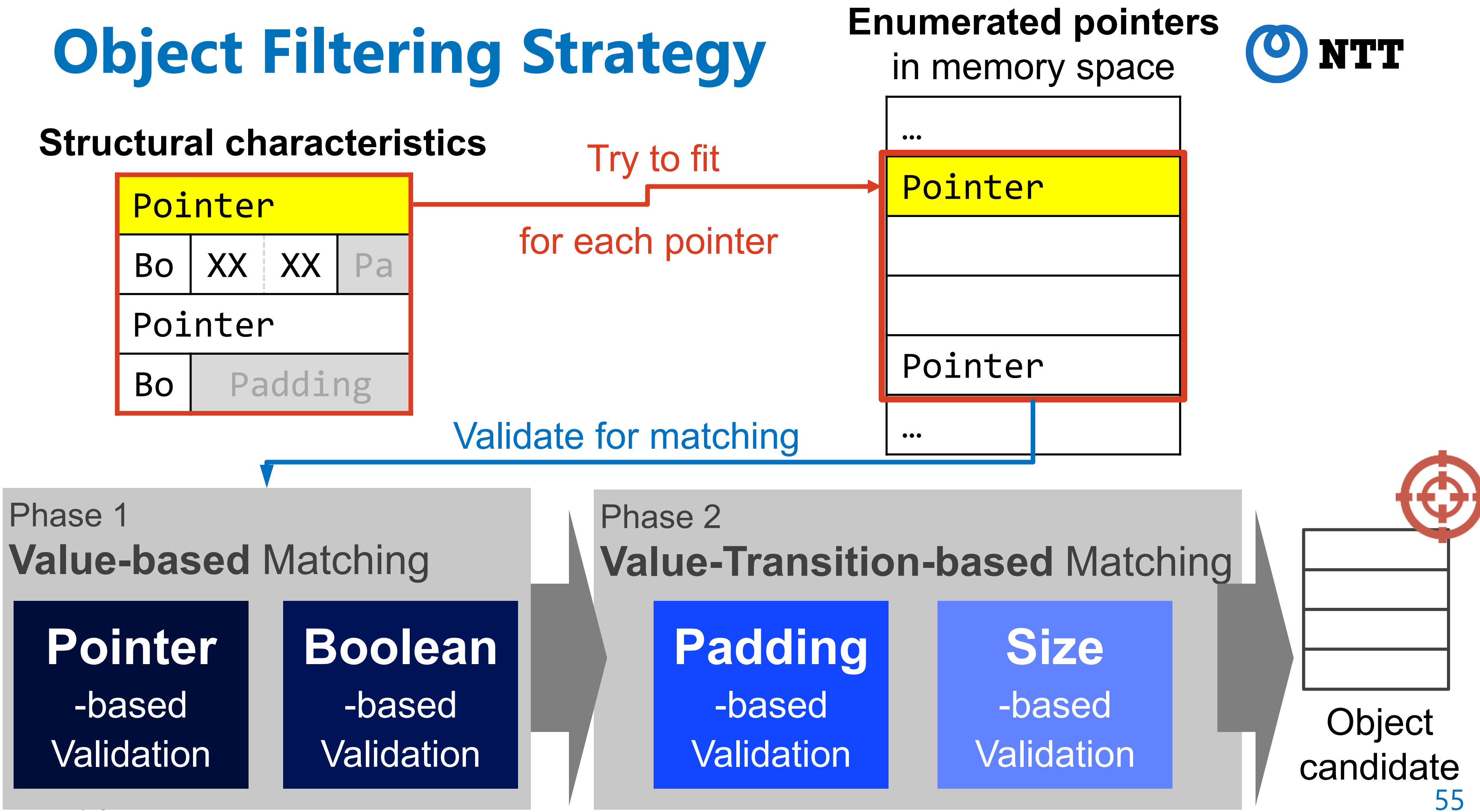
Point-to memory region is:
Valid

Valid

③ Check each pointer candidate whether the pointed-to memory region is valid

Invalid
(Not mapped memory region)

Object Filtering Strategy



Value-based Matching

Pointer-based Validation

Determine whether a member expected to be a pointer holds a plausible pointer value

Boolean-based Validation

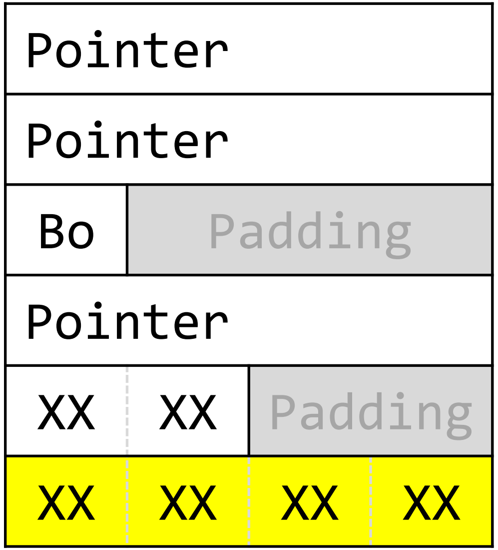
Determine whether a member expected to be a Boolean holds a valid Boolean value (e.g., 0x00 or 0xff)

Example of Value-based Matching

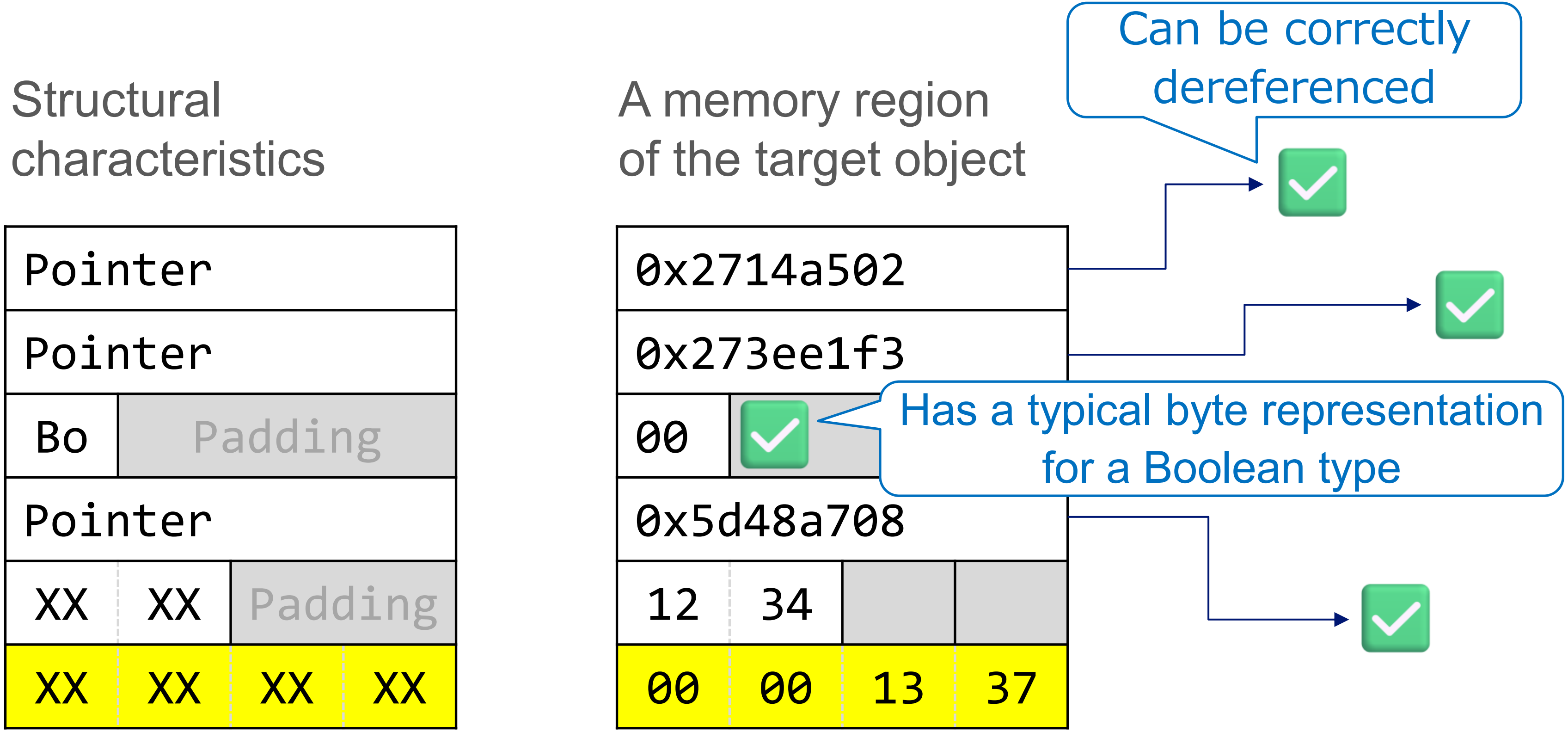
Ground truth source code

```c
struct target_object {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint16_t val;
    uint32_t target_val;
};
```

Reconstructed
structural characteristics



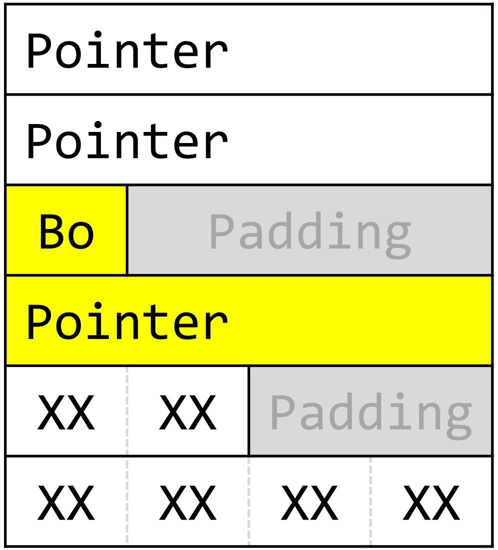
Example of Value-based Matching

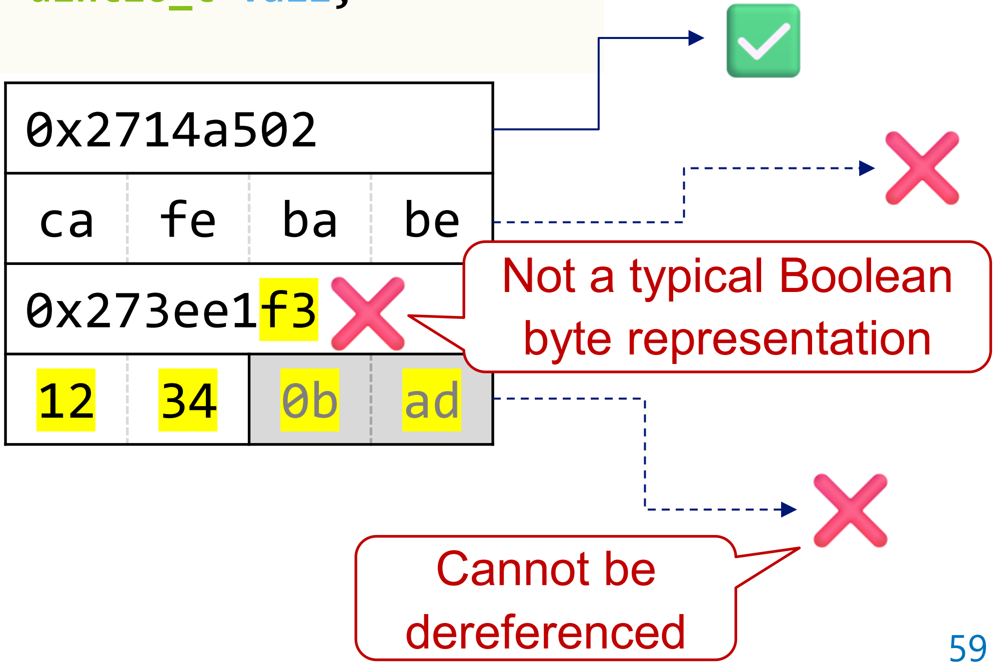


Example of Value-based Matching

```c
struct non_match_dummy {
    void *ptr1;
    uint32_t val1;
    void *ptr2;
    uint16_t val2;
};
```

Structural characteristics





Value-Transition-based Matching

Padding-based Validation

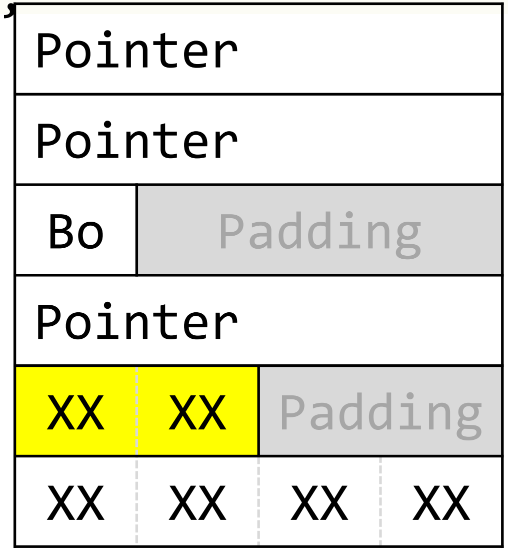
Confirm that the bytes in an area expected to be padding remains unchanged over time

Size-based Validation

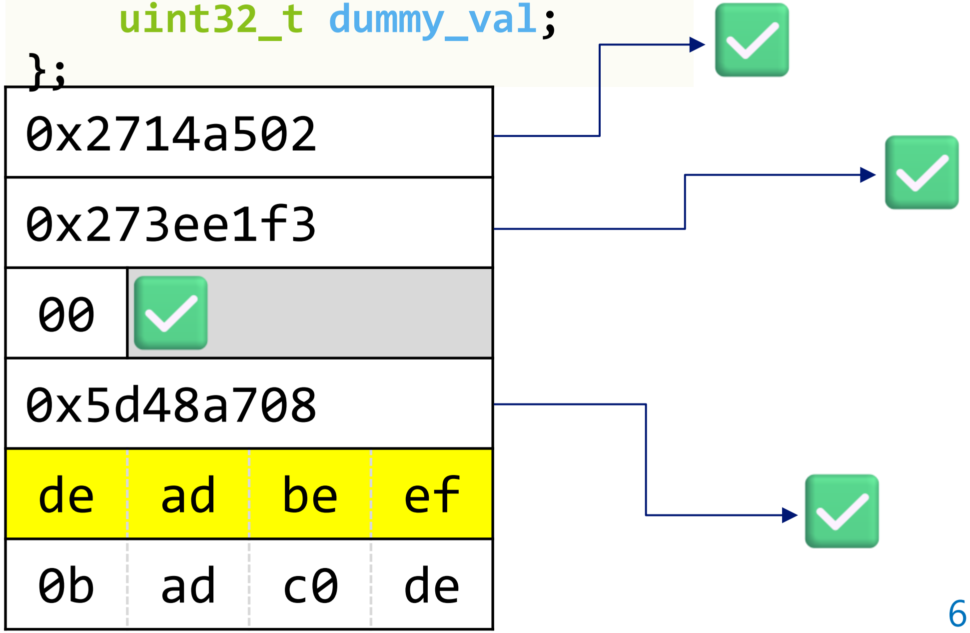
Confirm that no memory modification to each member variable exceeds the expected size over time

Motivating Example

```
struct target_object {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint16_t val;
    uint32_t target_val;
};
```




```
struct similar_dummy {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint32_t val;
    uint32_t dummy_val;
};
```



Value-Transition-based Matching 1

Padding-based Validation

```
struct target_object {
    void *ptr1;
    …
    uint16_t val;
    uint32_t target_val;
};
```

```
struct similar_dummy {
    void *ptr1;
    …
    uint32_t val;
    uint32_t dummy_val;
};
```

0x27354a08			
0x27373dfc			
00			
0x5d473c84			
12	34	00	00
00	03	13	37

0x2714a502			
0x273ee1f3			
00			
0x5d48a708			
de	ad	be	ef
0b	ad	c0	de

Value-Transition-based Matching 1

Padding-based Validation

```c
struct target_object {
    void *ptr1;
    …
    uint16_t val;
    uint32_t target_val;
};
```

```c
struct similar_dummy {
    void *ptr1;
    …
    uint32_t val;
    uint32_t dummy_val;
};
```

0x27354a08			
0x27373dfc			
00			
0x5d473c84			
56	78	00	00

The expected padding region remains unchanged

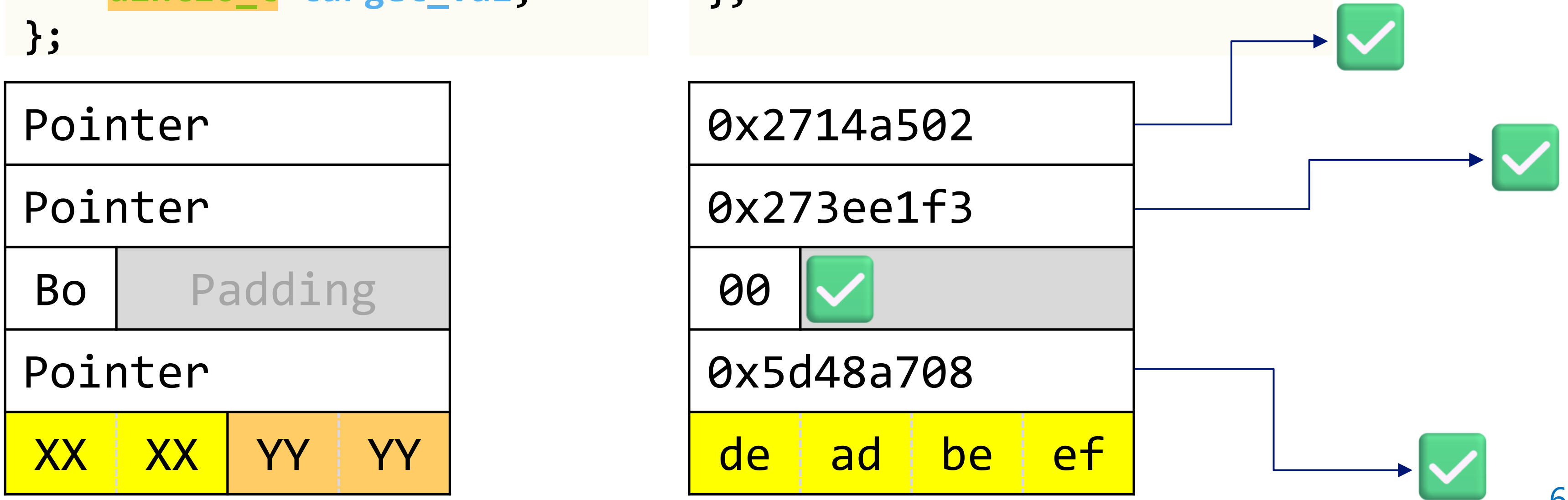
0x2714a502			
0x273ee1f3			
00			
0x5d48a708			
ca	fe	ba	be

The expected padding region has changed

Another Motivating Example

```c
struct target_object_2 {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint16_t val;
    uint16_t target_val;
};
```

```c
struct similar_dummy_2 {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint32_t dummy_val;
};
```



Value-Transition-based Matching 2

Size-based Validation

```c
struct target_object_2 {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint16_t val;
    uint16_t target_val;
};
```

```c
struct similar_dummy_2 {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint32_t dummy_val;
};
```

0x27354a08			
0x27373dfc			
00			
0x5d473c84			
12	34	56	78

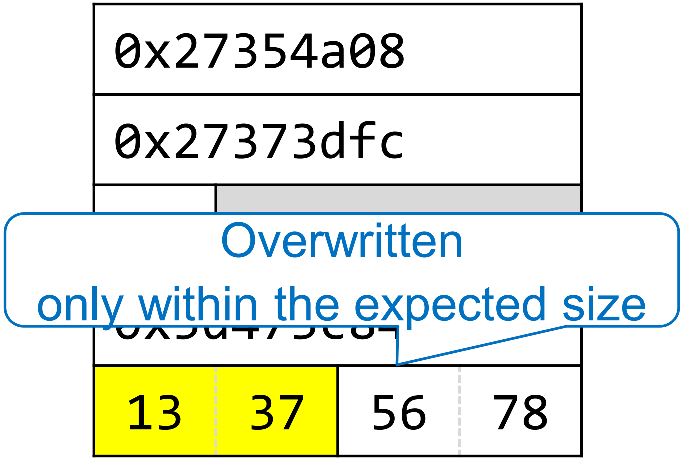
0x2714a502			
0x273ee1f3			
00			
0x5d48a708			
de	ad	be	ef

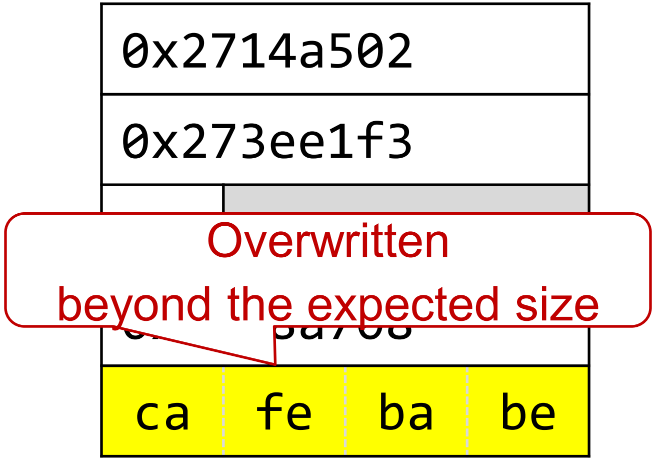
Value-Transition-based Matching 2

Size-based Validation

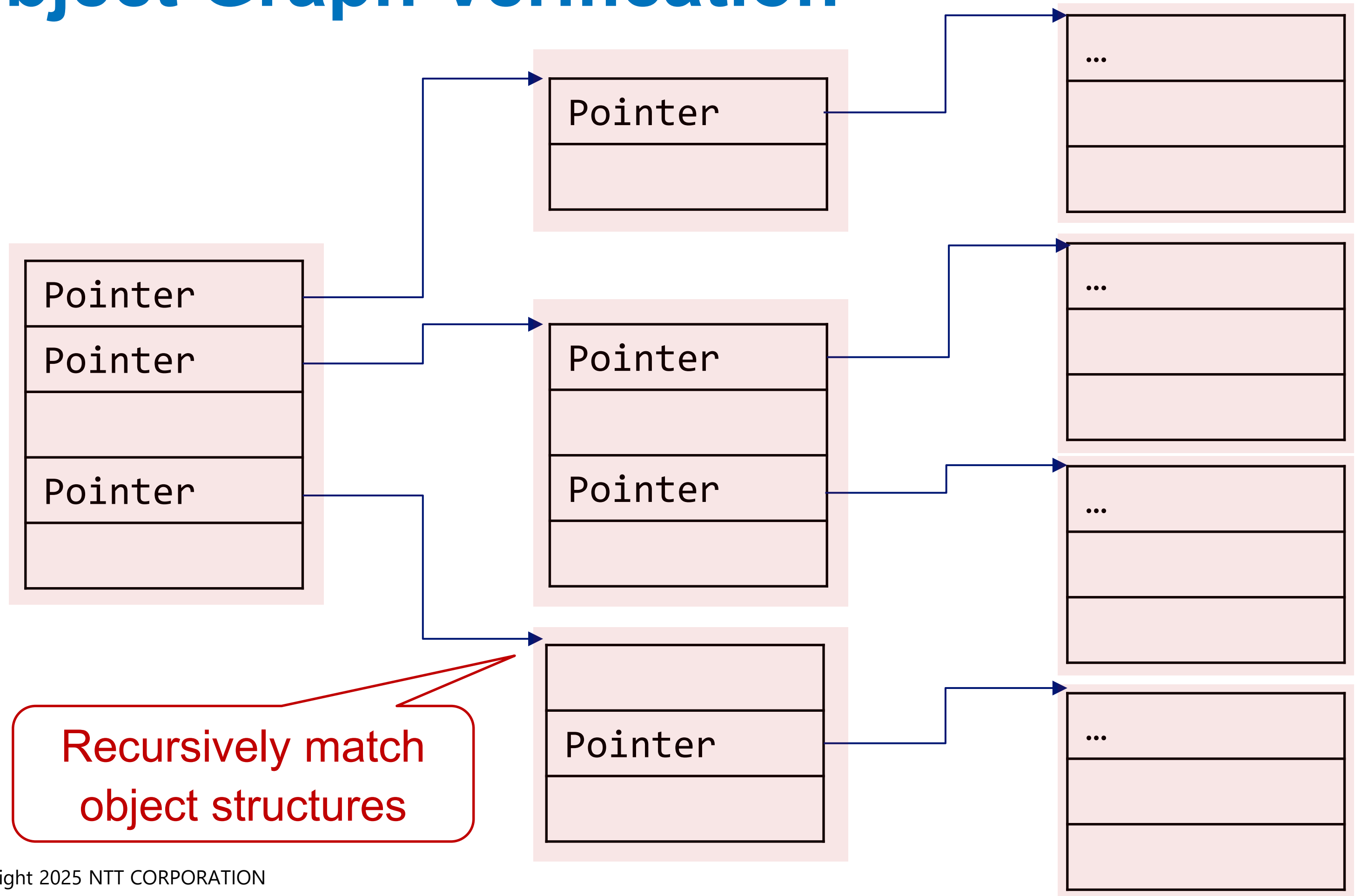
```c
struct target_object_2 {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint16_t val;
    uint16_t target_val;
};
```

```c
struct similar_dummy_2 {
    void *ptr1;
    void *ptr2;
    bool flag;
    void *ptr3;
    uint32_t dummy_val;
};
```



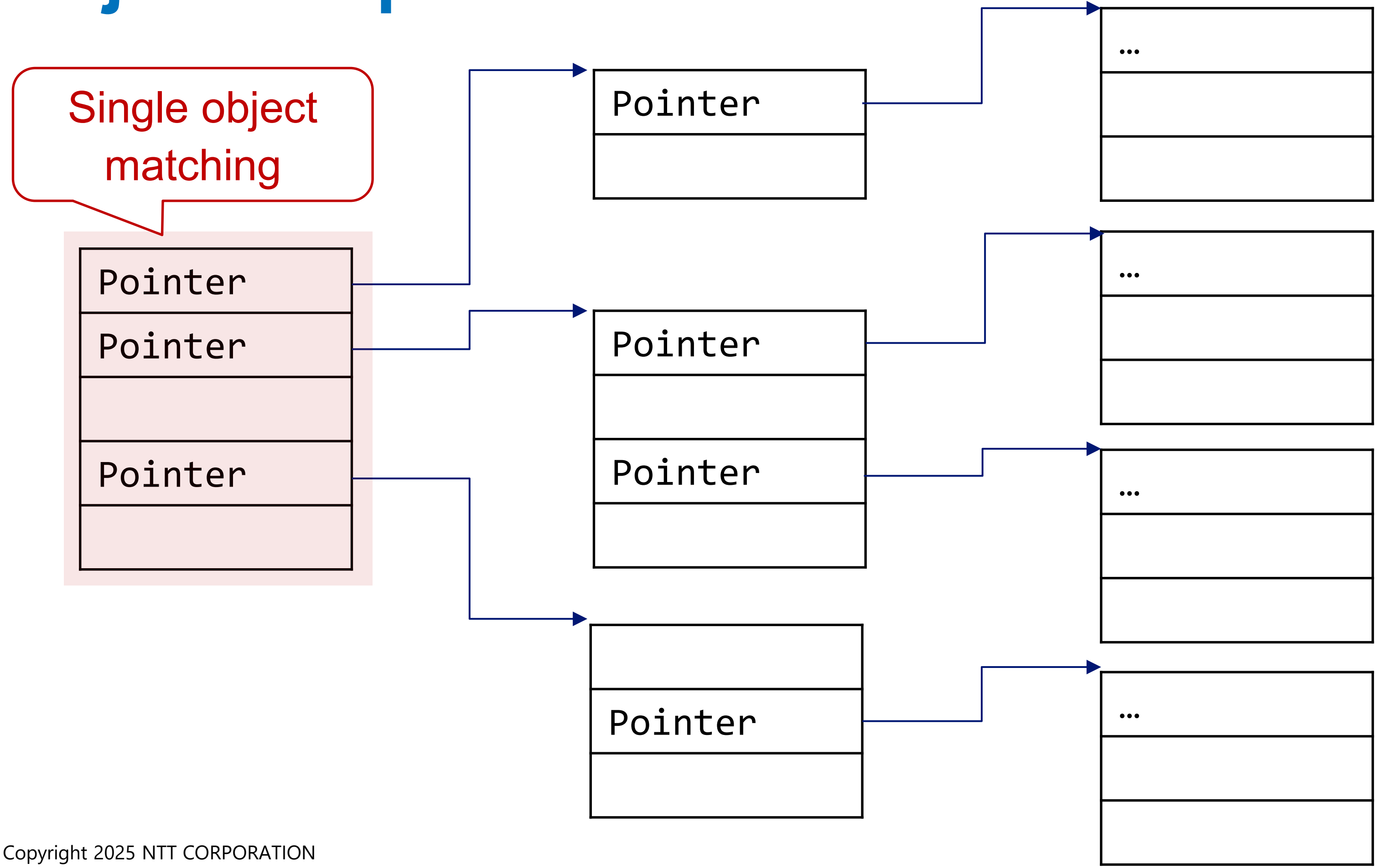


Object Graph Verification

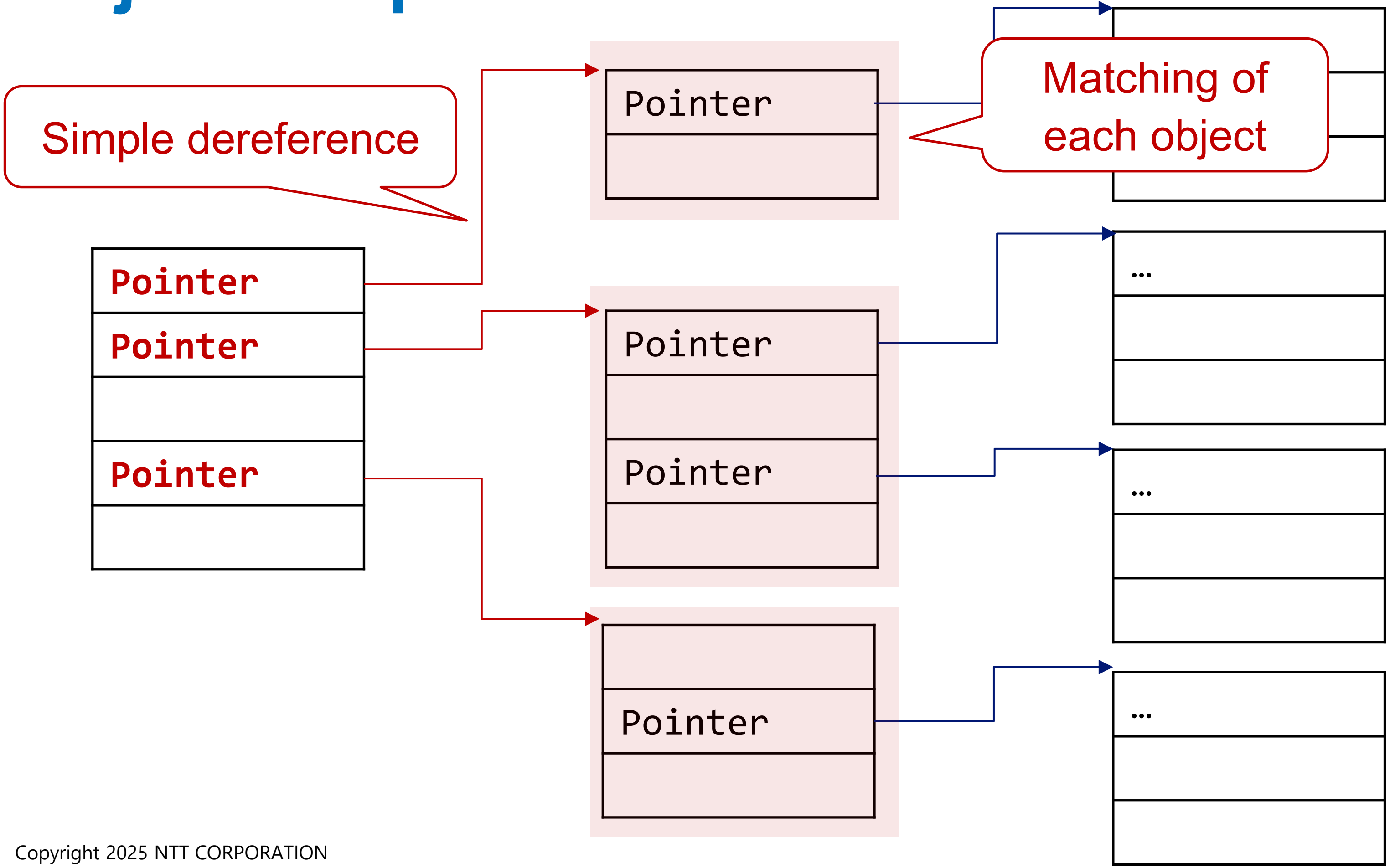


Recursively match object structures

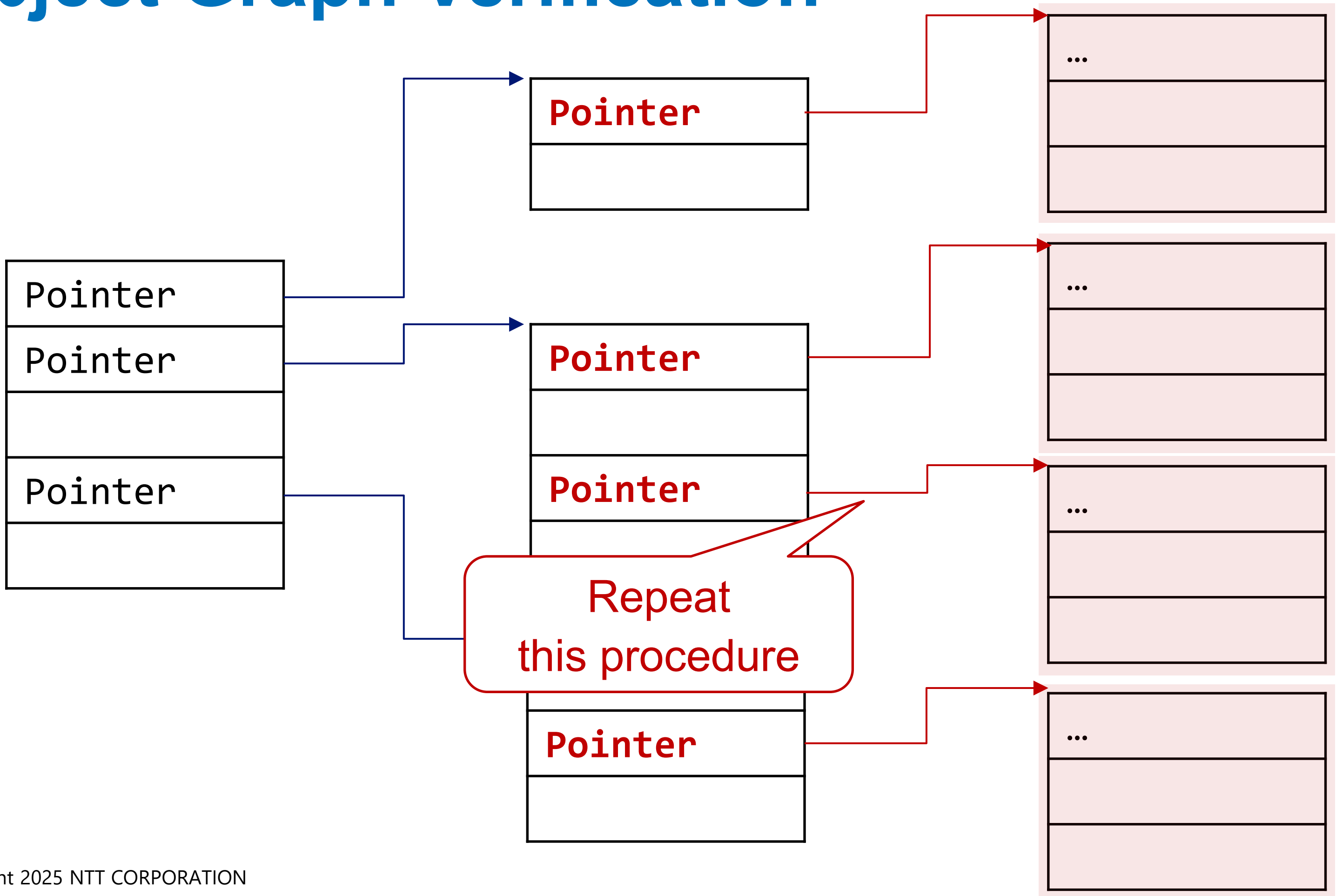
Object Graph Verification



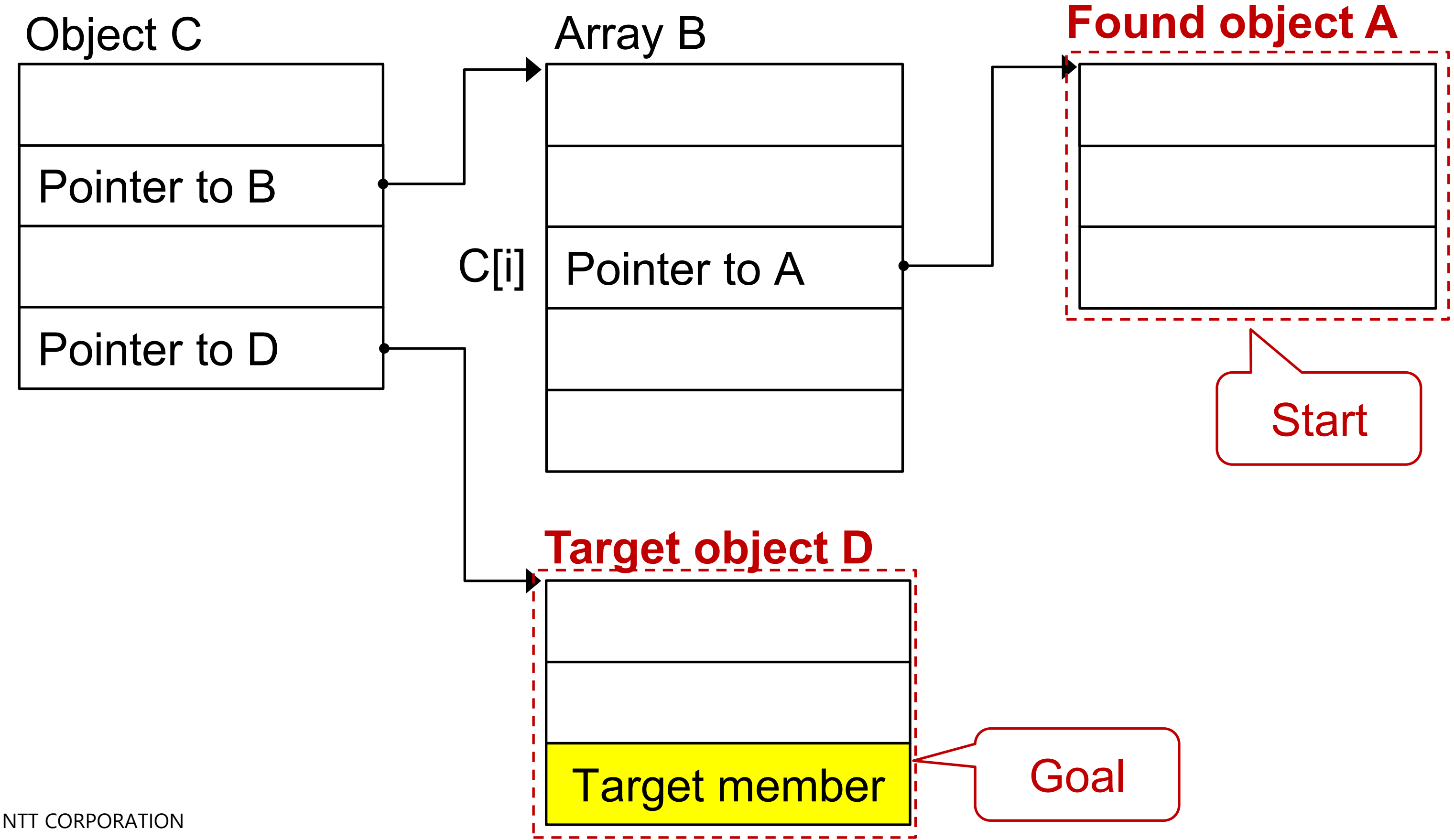
Object Graph Verification



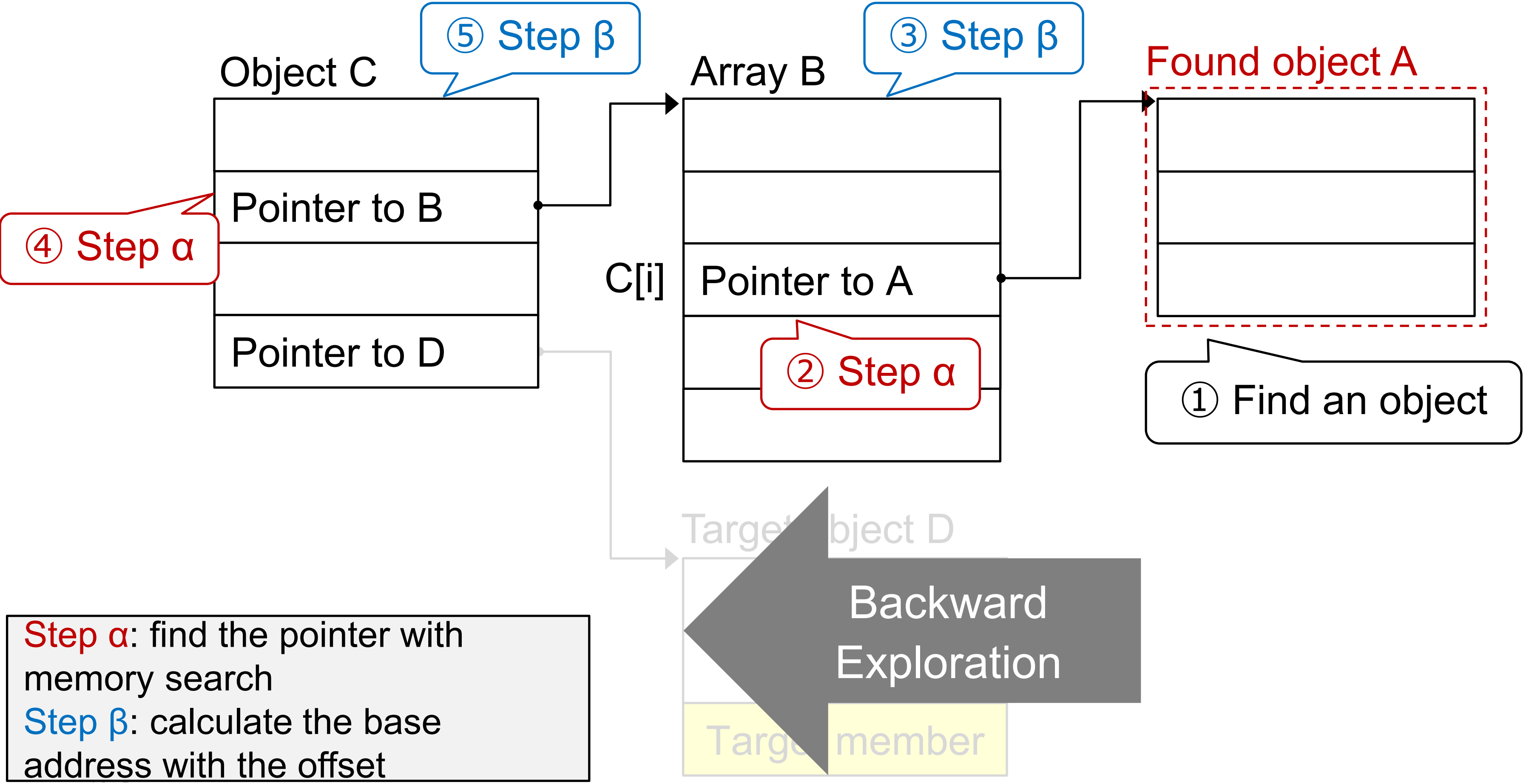
Object Graph Verification



Target Member Discovery

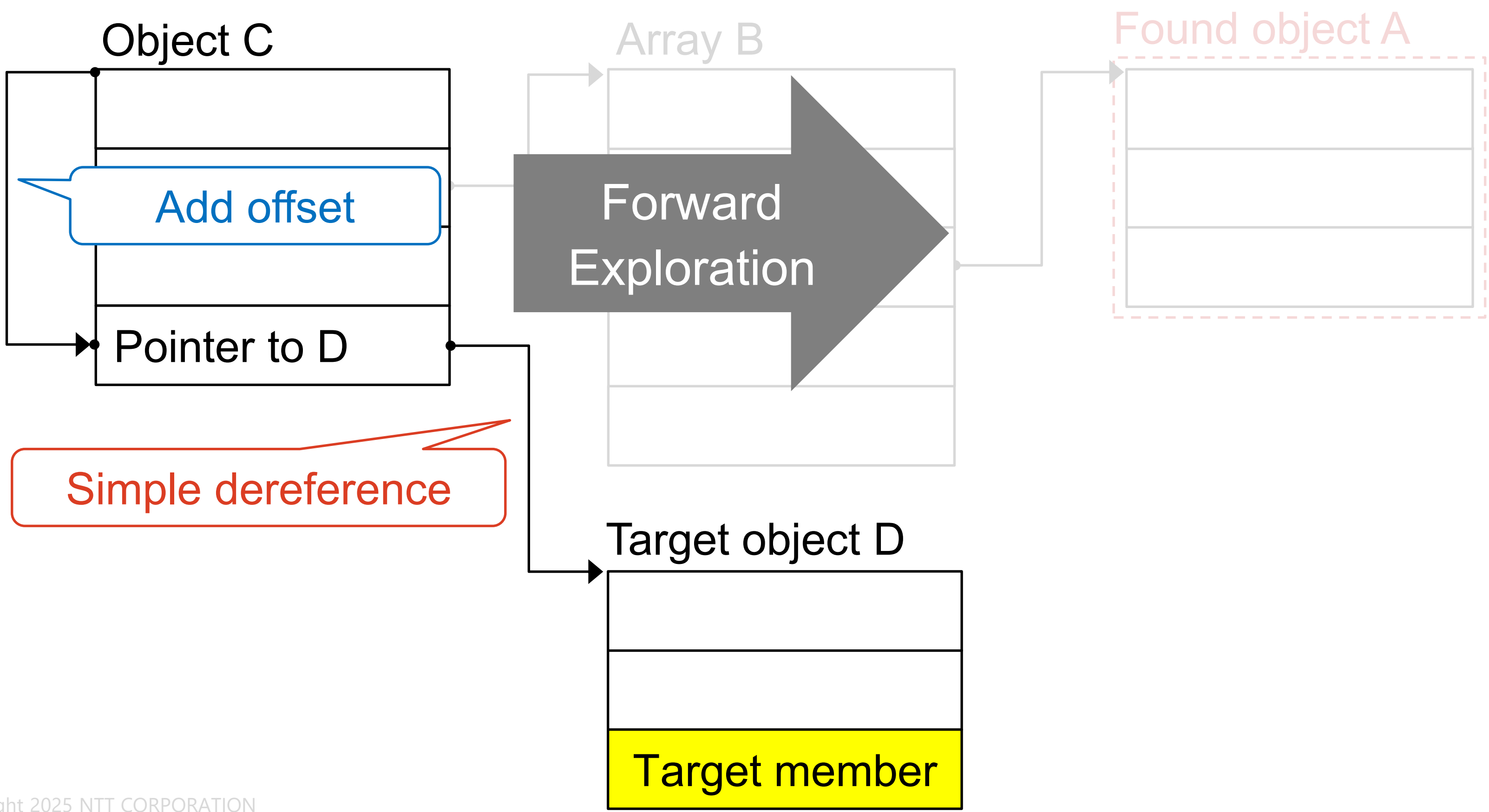


Target Member Discovery

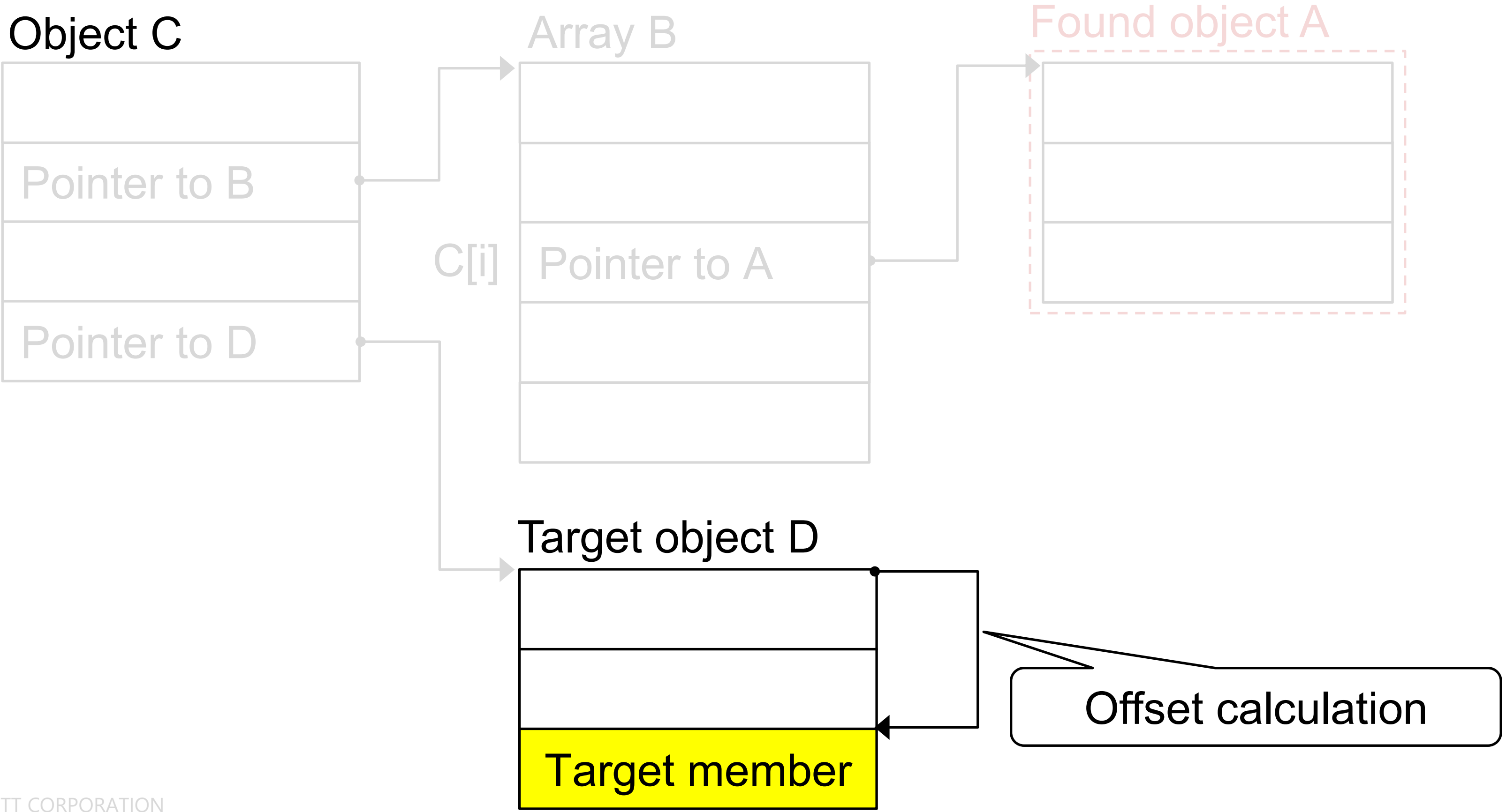


Step α: find the pointer with memory search
Step β: calculate the base address with the offset

Target Member Discovery

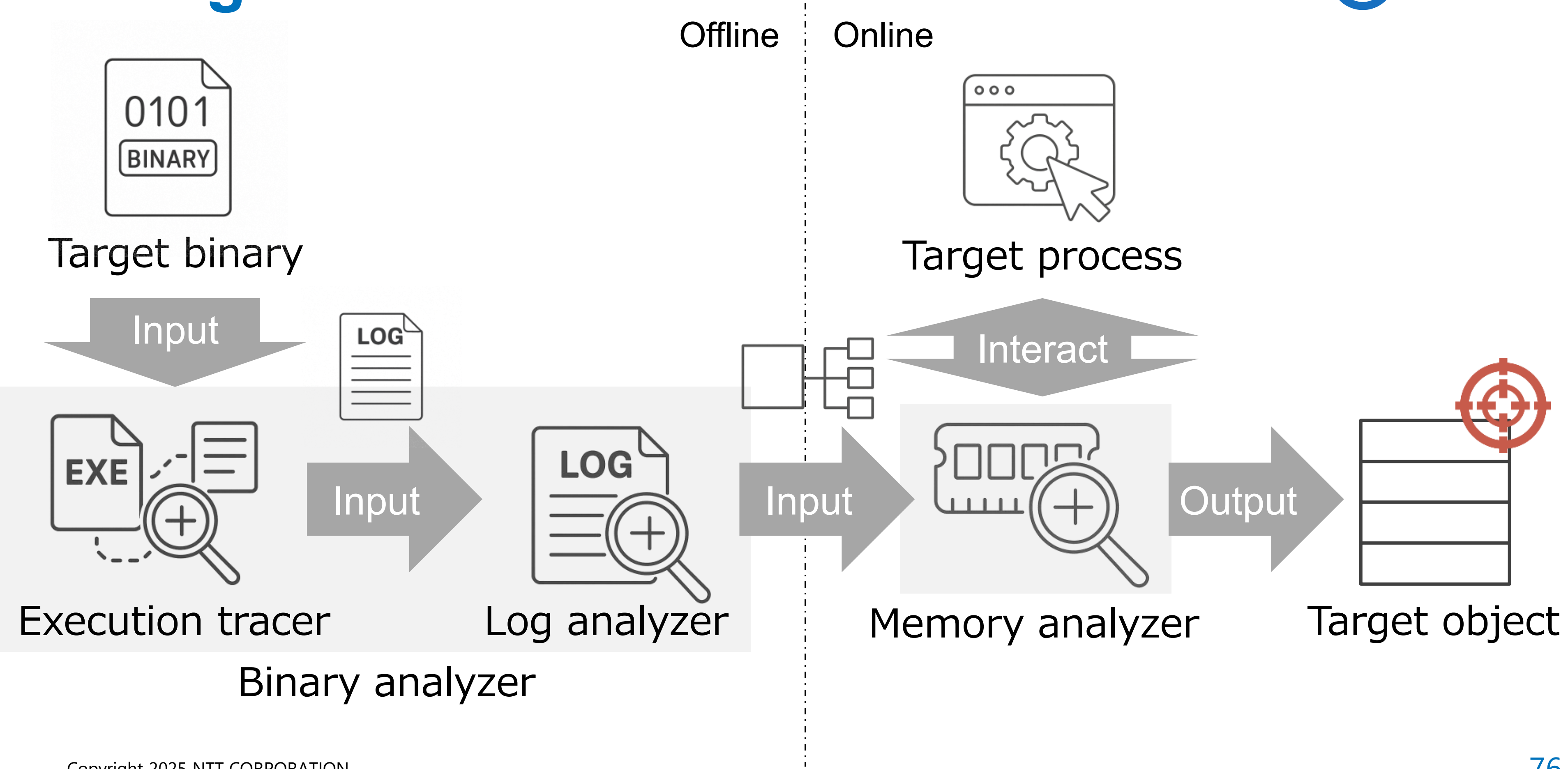


Target Member Discovery

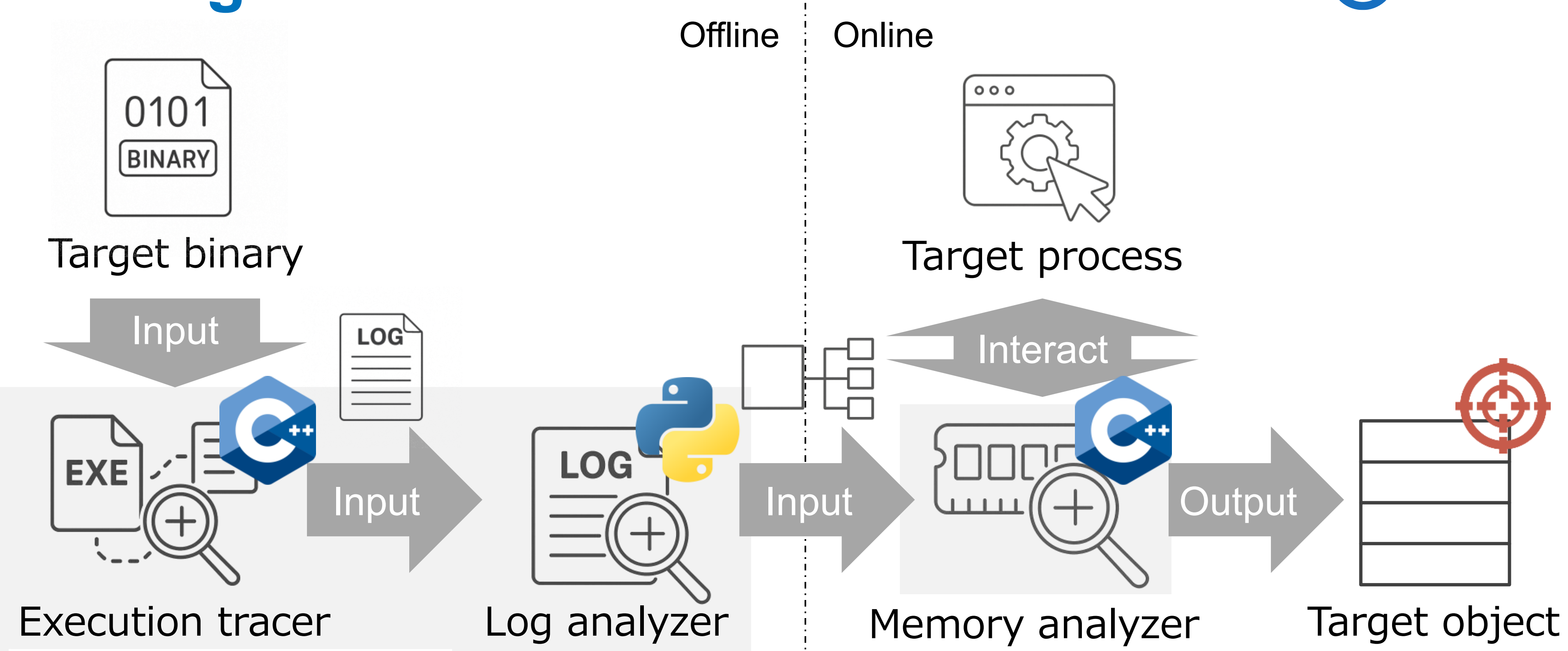


Design, Implementation, and Evaluation

Design


Offline
Online
0101
BINARY
Target binary
Input
LOG
EXE
Input
LOG
Execution tracer
Log analyzer
Binary analyzer
Target process
Interact
Input
Output
Memory analyzer
Target object

Design



DBI	Intel Pin
Taint engine	libdft64

Graph	NetworkX

Research Questions for Evaluation

NTT

RQs in our binary analysis technique

RQ1	【 Accuracy 】	Can the technique correctly extract the required structural characteristics from binaries?
RQ2	【 Performance 】	Can the analysis complete within a realistic timeframe?
RQ3	【 Universality 】	To what extent can the results of binary analysis be reused?

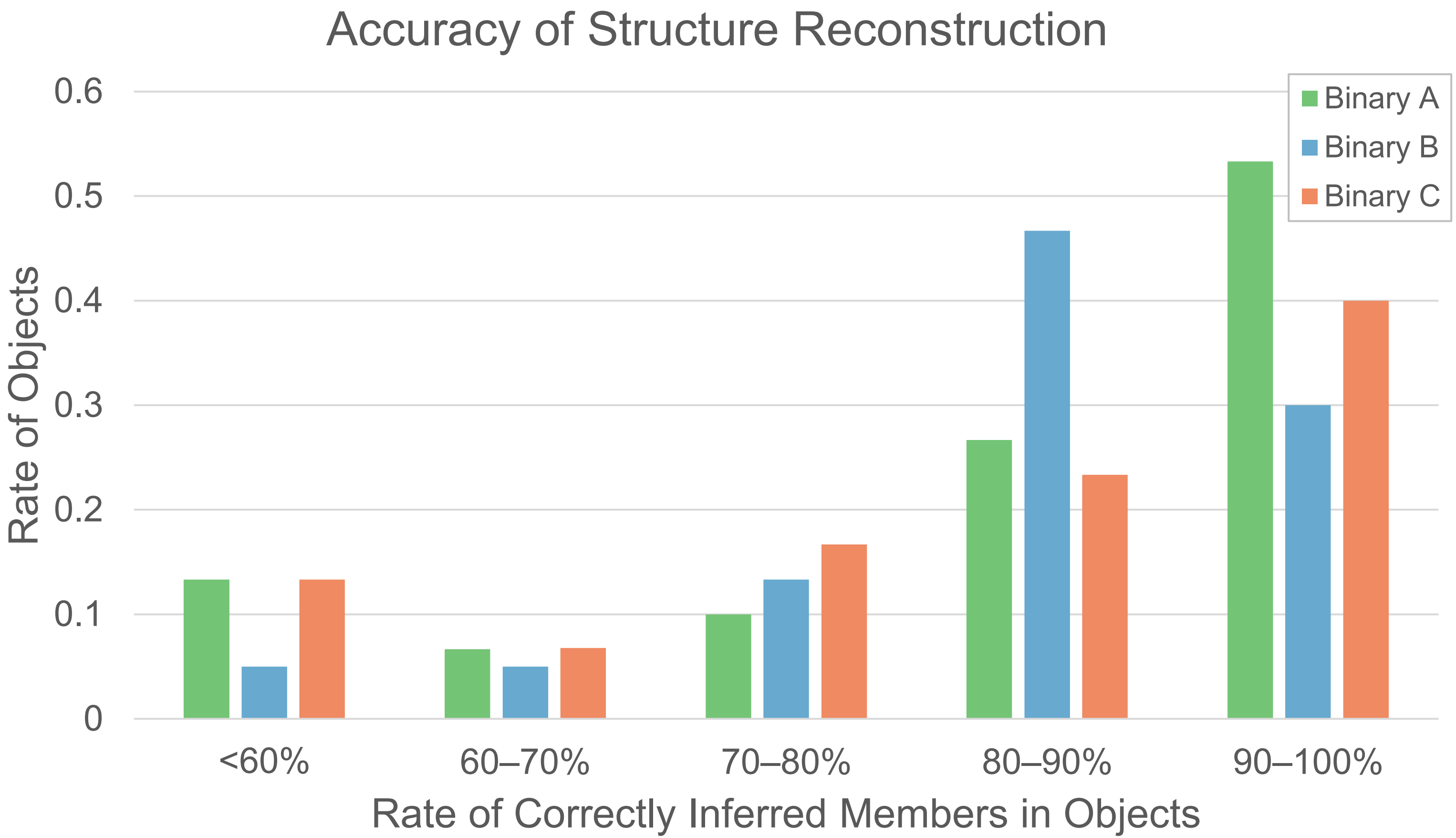
RQ1: Accuracy of Binary Analysis

Evaluation criteria of object structure reconstruction

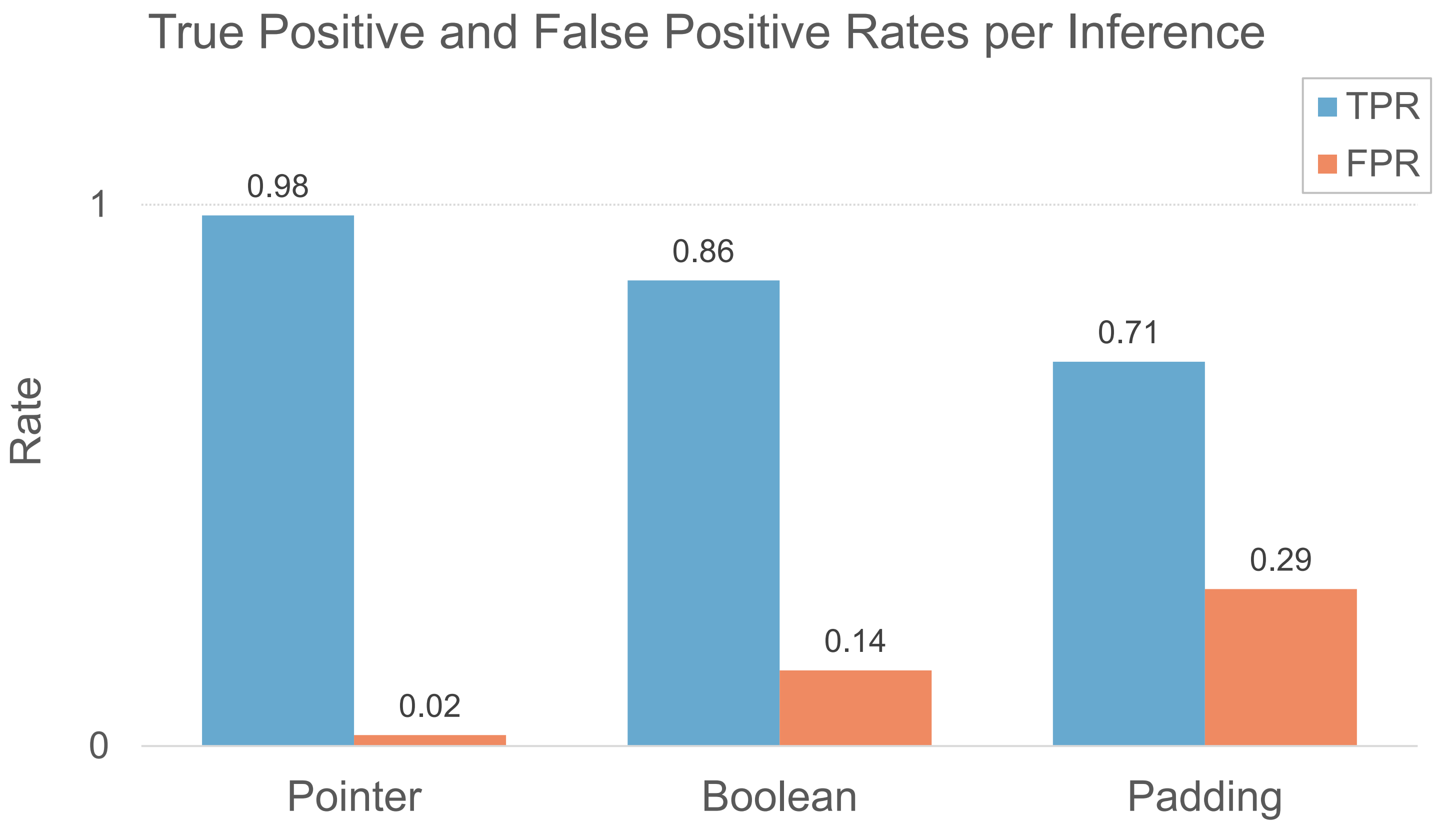
Offset	Actual size (Ground truth)	Inferred size	Result
0x000	8	8	✅ (Correct)
0x008	8	-	❌ (Not inferred)
0x010	2	1	❌ (Wrong)
…			

We evaluate: $\frac{\text{\# of correctly inferred members}}{\text{\# of all members}}$

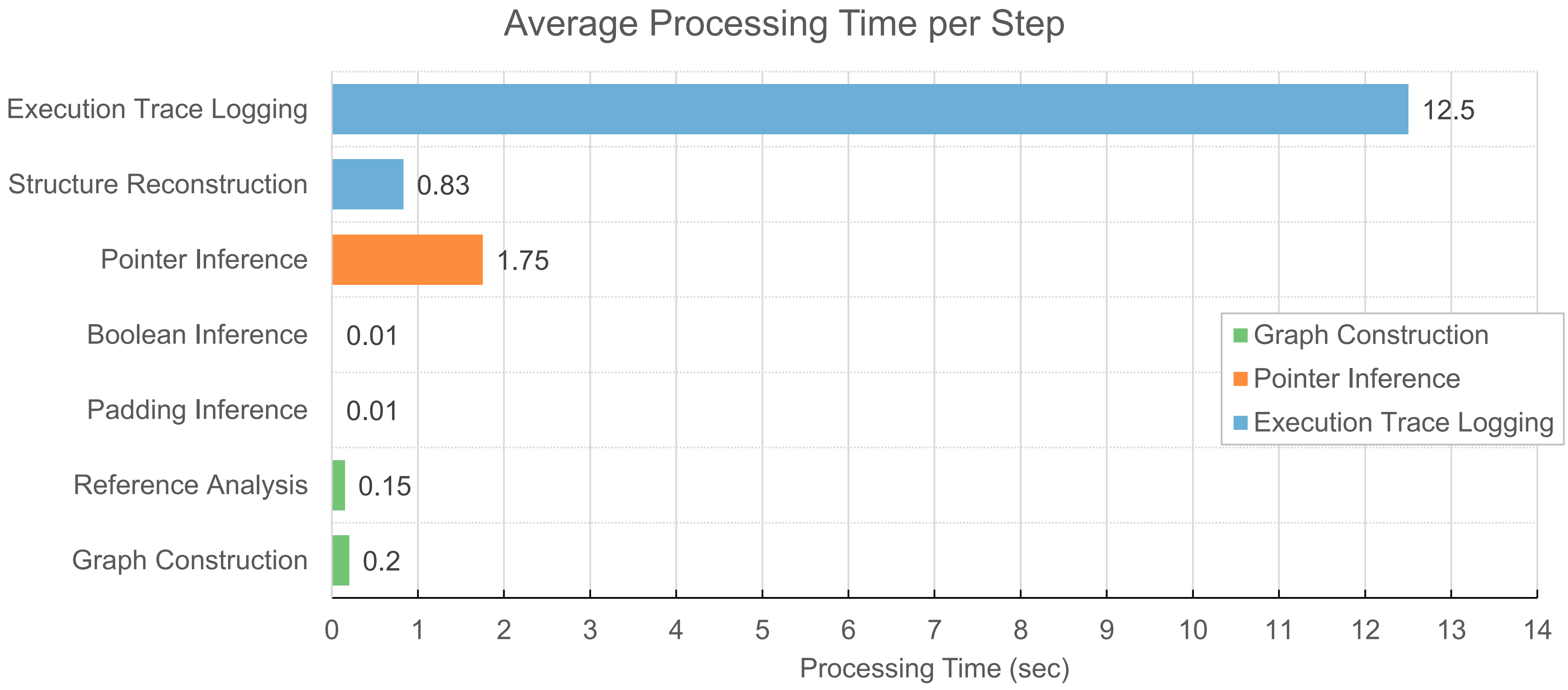
RQ1: Accuracy of Binary Analysis



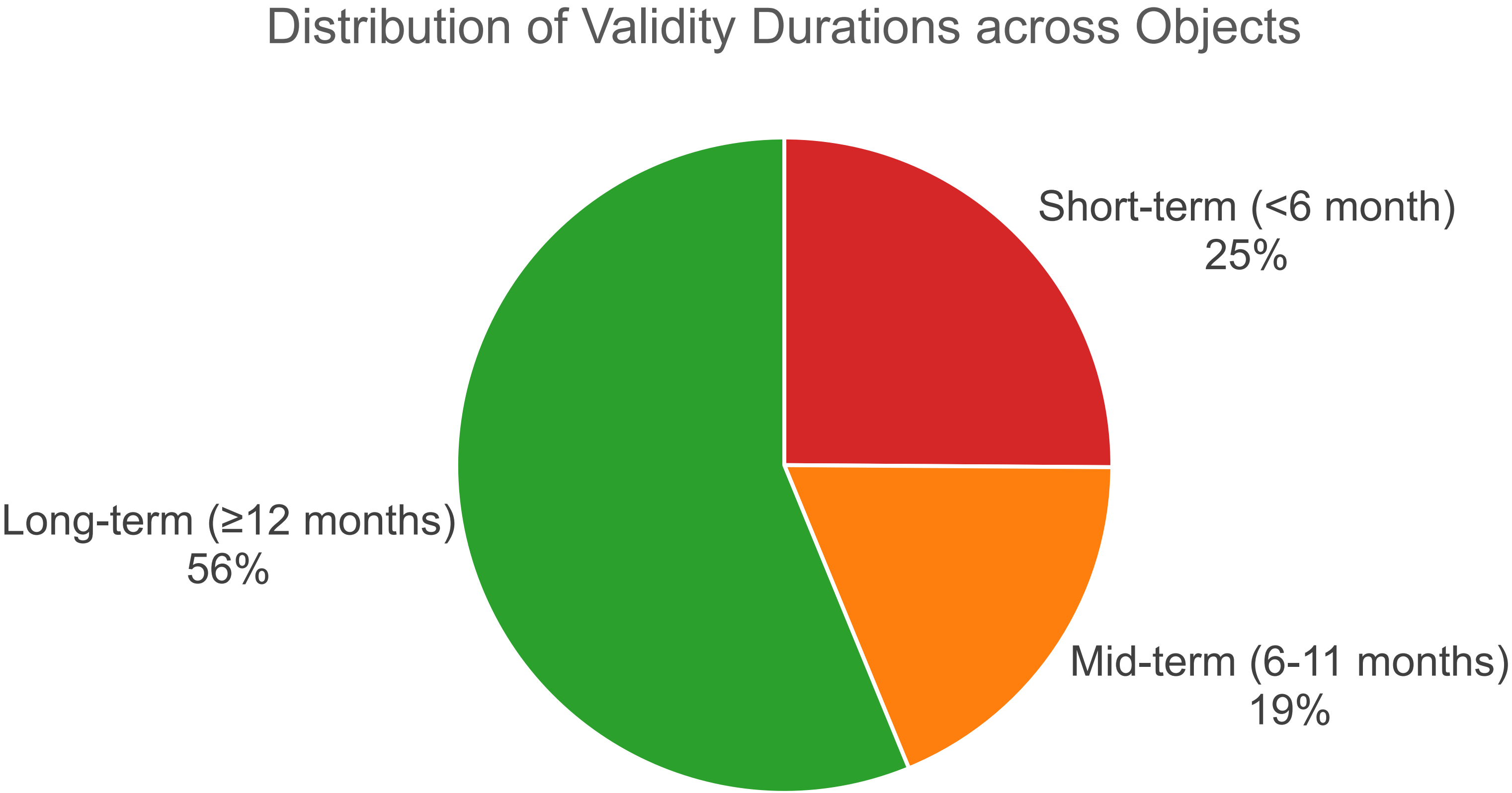
RQ1: Accuracy of Binary Analysis



RQ2: Performance of Binary Analysis



RQ3: Universality of Binary Analysis

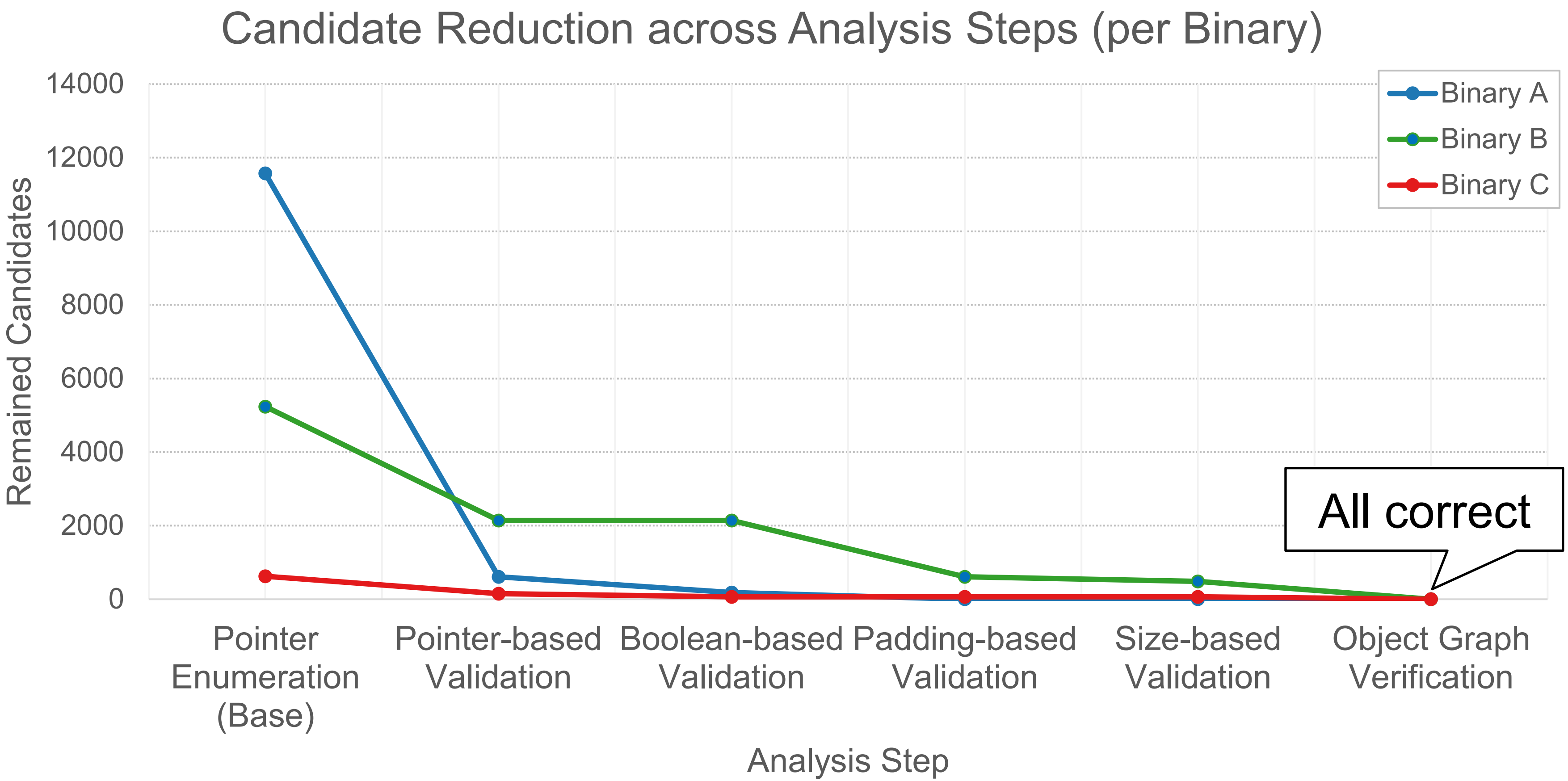


Research Questions

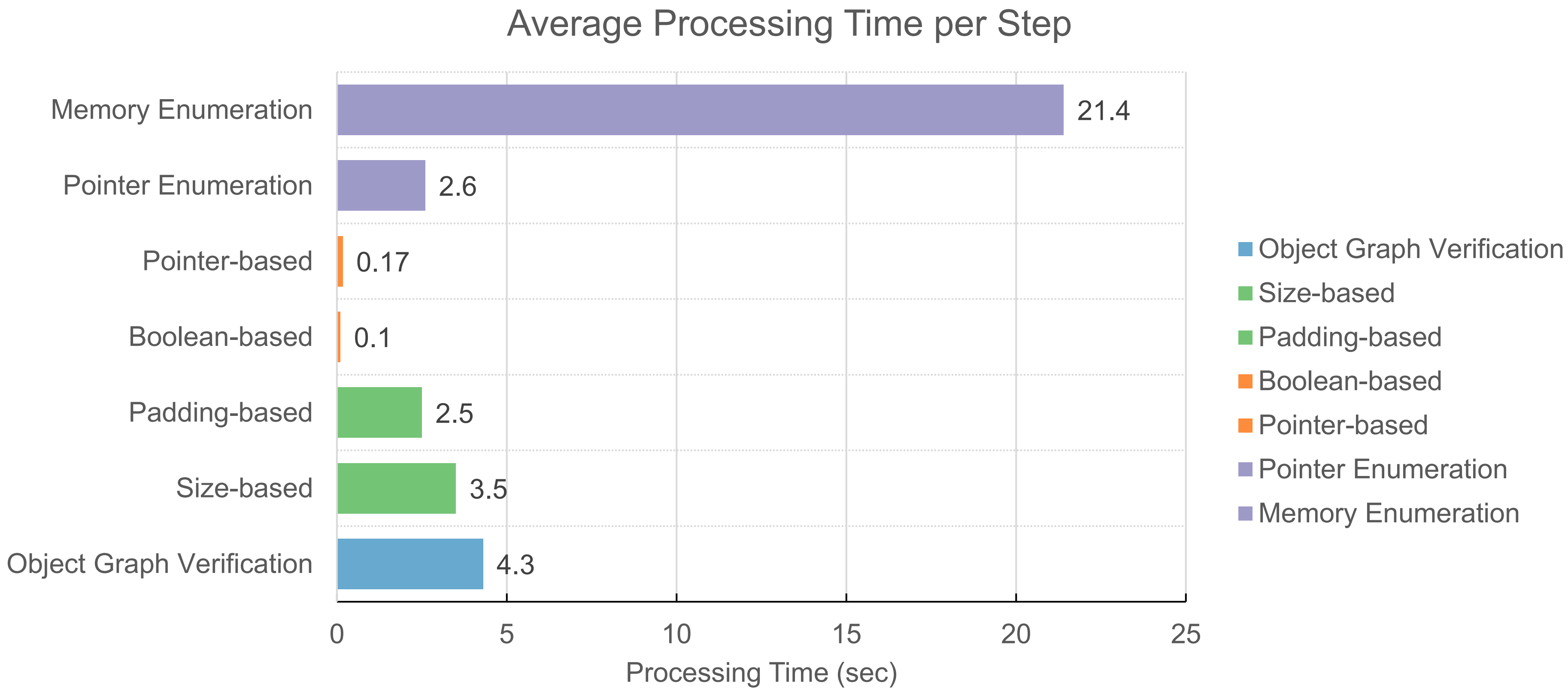
RQs in our memory analysis technique

RQ4	【 Accuracy 】 How accurate is location identification of target objects?
RQ5	【 Performance 】 Can the memory exploration complete within a realistic timeframe?
RQ6	【 Universality 】 How generalizable are our memory analysis technique across different (changed) memory layouts?

RQ4: Accuracy of Memory Analysis



RQ5: Performance of Memory Analysis

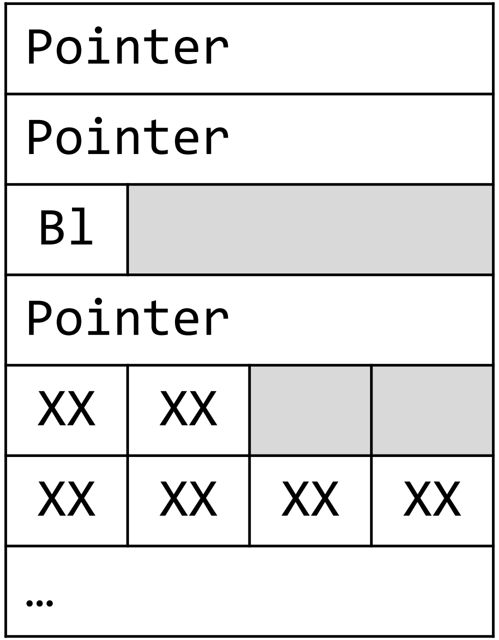


RQ6: Universality of Memory Analysis

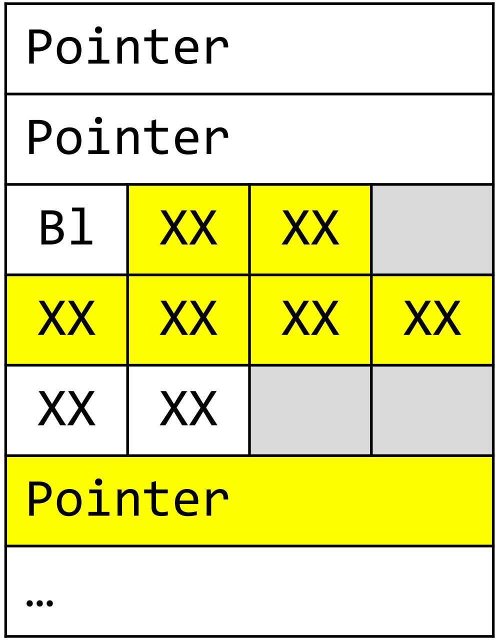
NTT

Case Studies Observed in Samples

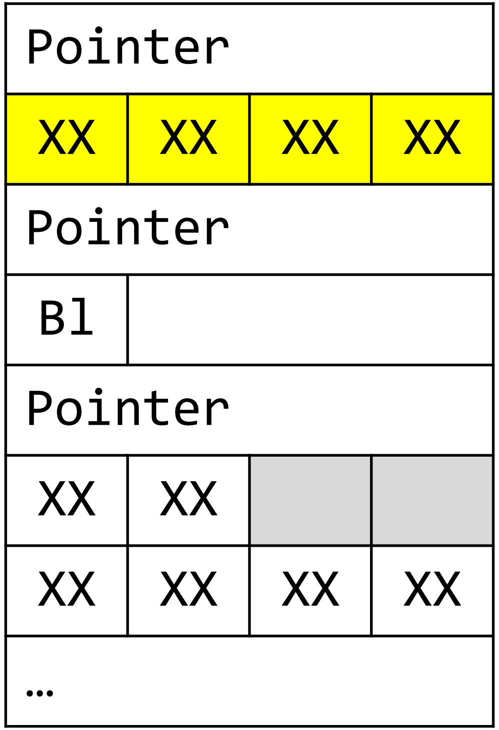
Base



Acceptable change

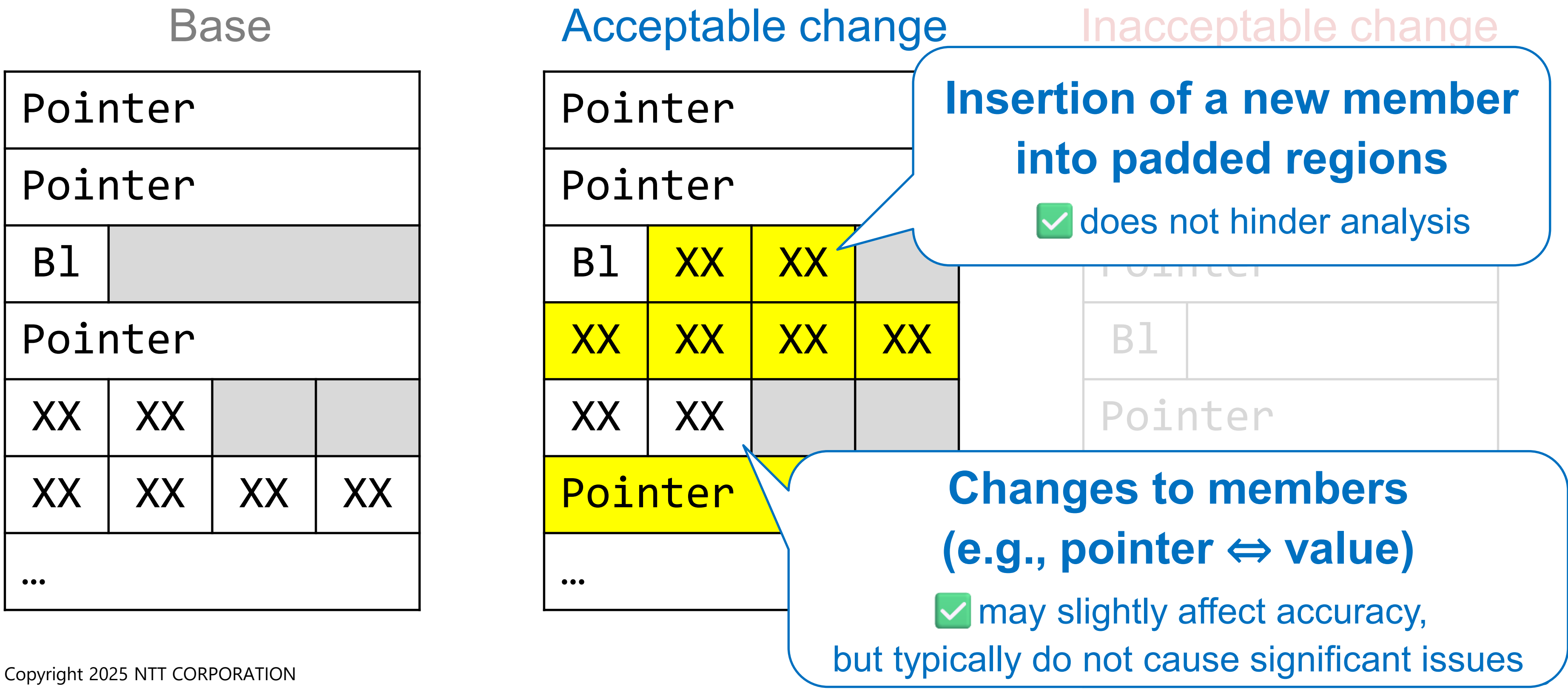


Inacceptable change



RQ6: Universality of Memory Analysis

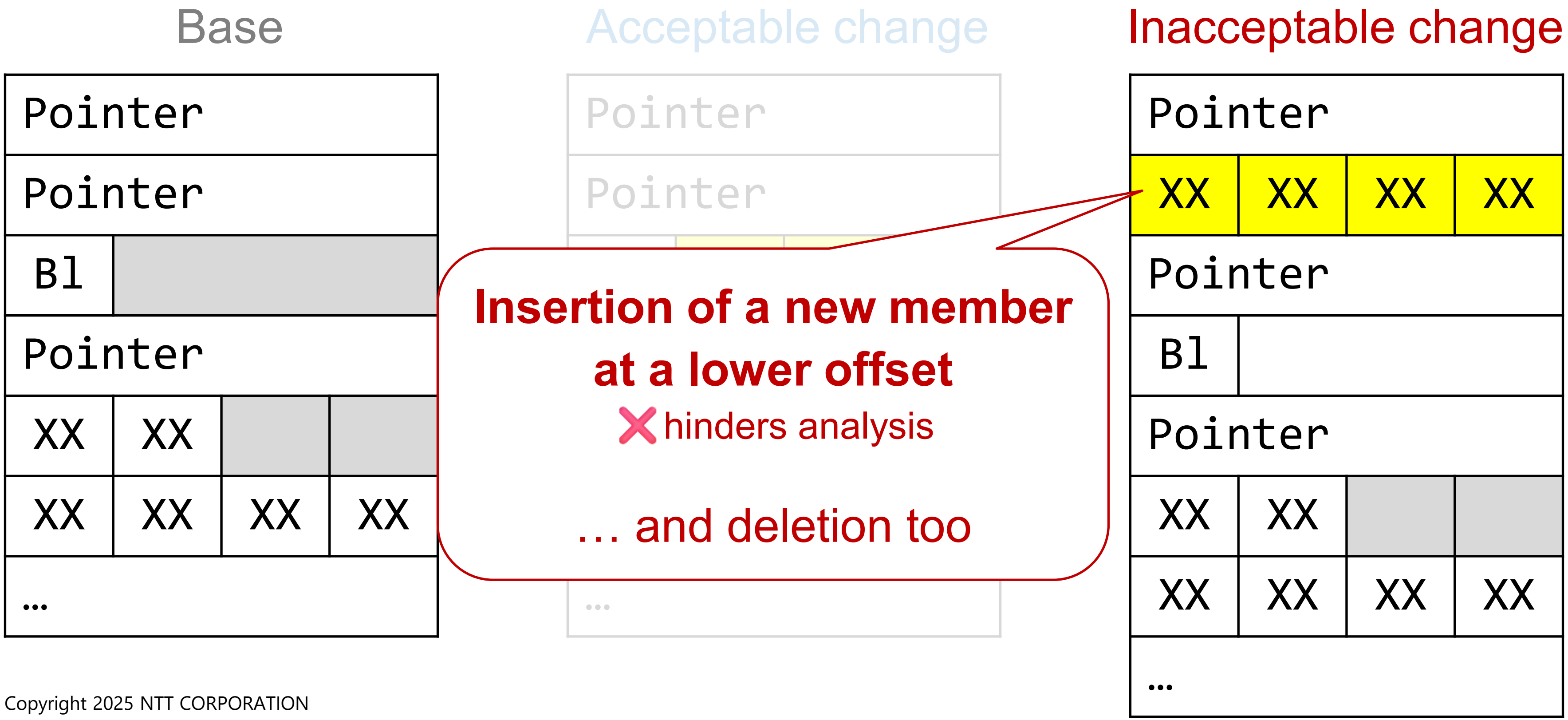
Case Studies Observed in Samples



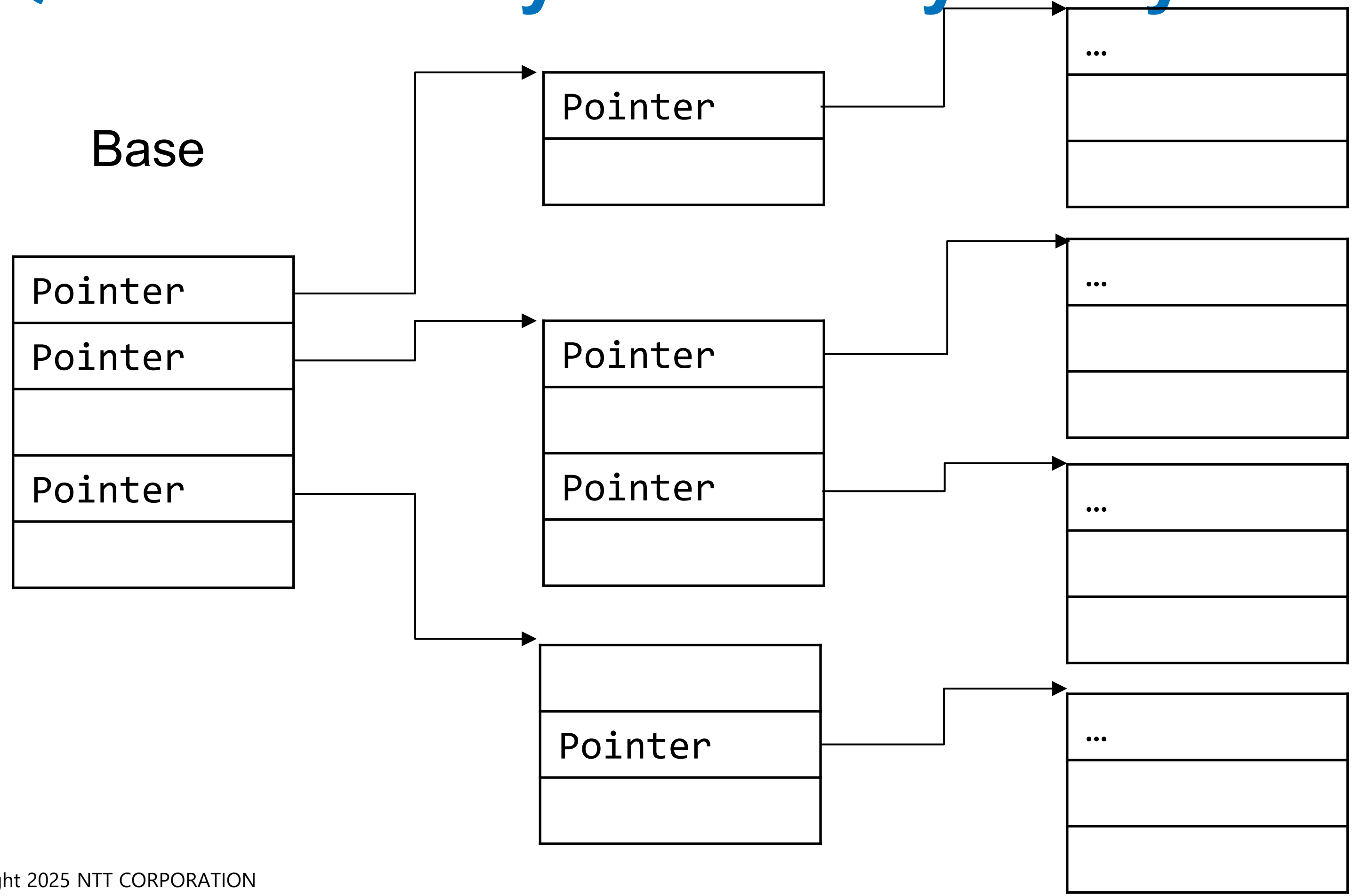
RQ6: Universality of Memory Analysis

NTT

Case Studies Observed in Samples

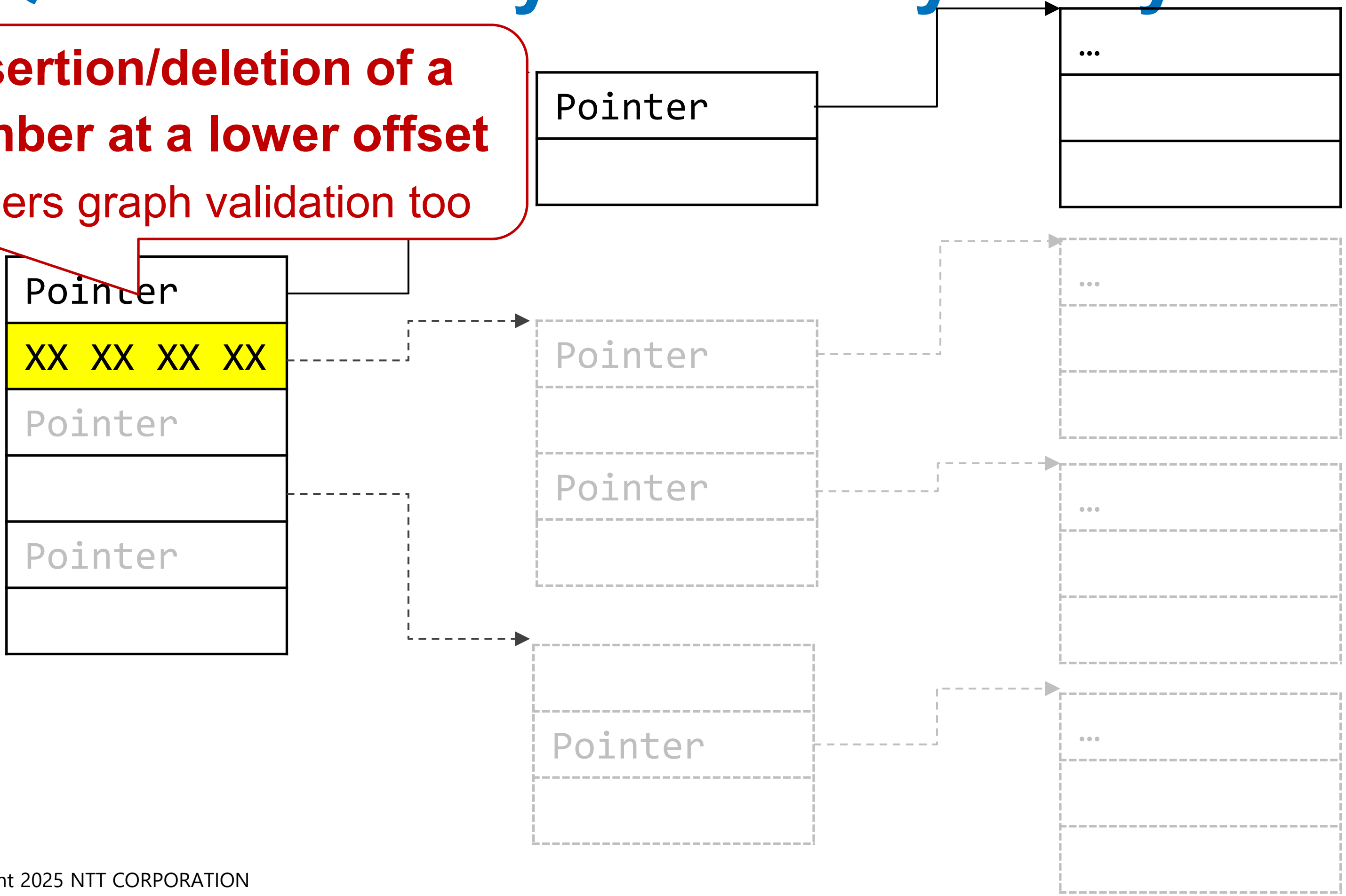


RQ6: Universality of Memory Analysis

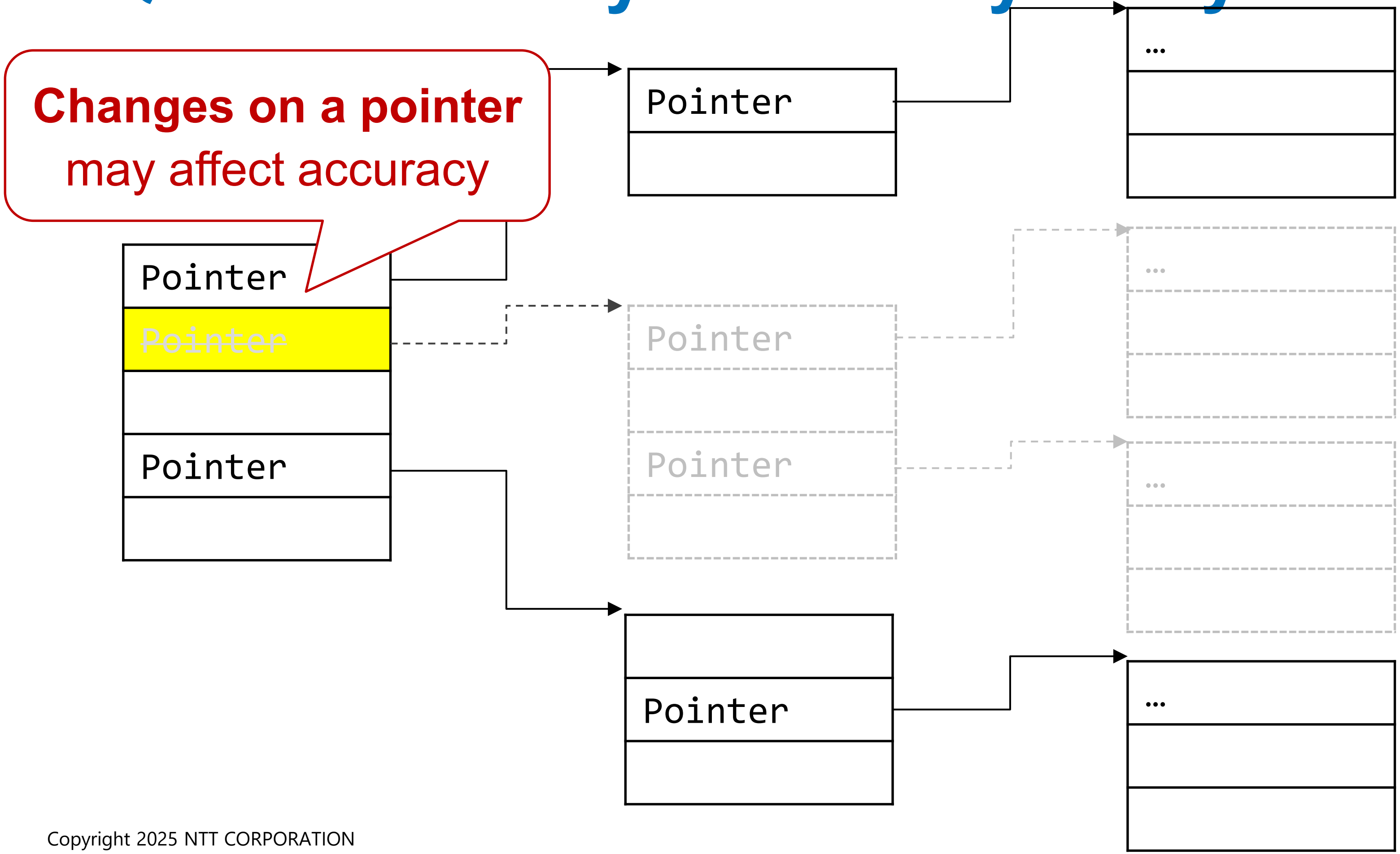


RQ6: Universality of Memory Analysis

Insertion/deletion of a member at a lower offset hinders graph validation too



RQ6: Universality of Memory Analysis



Discussion

Limitations: Binary Analysis

- Dependence on Observability of Structural Characteristics
  - Our binary analysis operates on the execution state observed during a single execution path
  - This limitation raises concerns regarding coverage in recovering object structures and their reference relationships
- Challenging cases include:
  - Objects not holding pointers at binary/memory analysis time
  - Objects holding union types or generic references (e.g., void *)

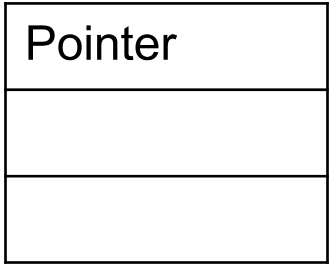
Limitations: Binary Analysis

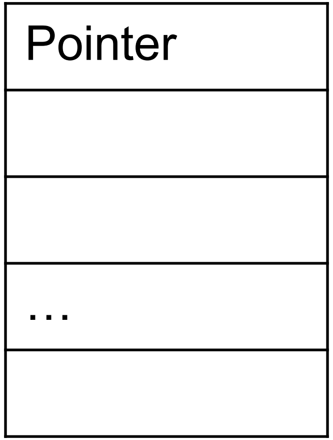
- Mitigation strategies:
  - **Execution with diverse inputs**:
    combining with techniques such as fuzzing
  - **Approximate matching**:
    using thresholds during memory analysis

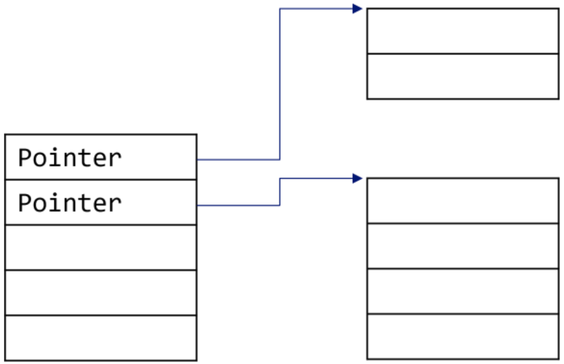
Limitations: Memory Analysis

- Difficulty in identifying **very small objects**
  - Object graphs composed of few and small objects tend to cause false positives due to insufficient structural distinctiveness
  - However, we consider cases where both value-based and structure-based characteristics are lacking to be uncommon
- Interference from **memory protection mechanisms**
  - Memory analysis may fail when raw pointer values are inaccessible due to protections
  - E.g., pointer tagging, pointer encryption

Conditions for Successful Identification

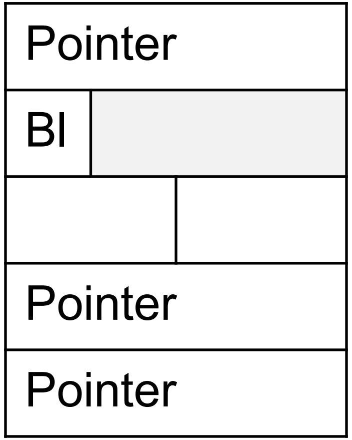


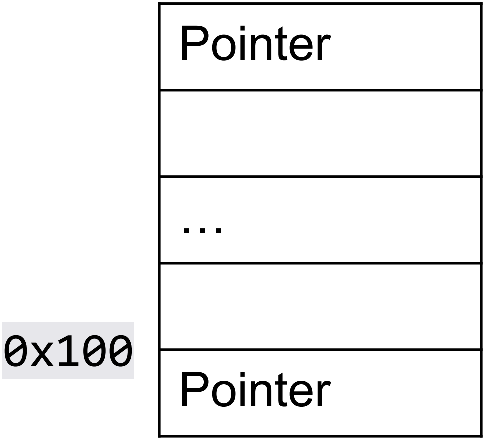


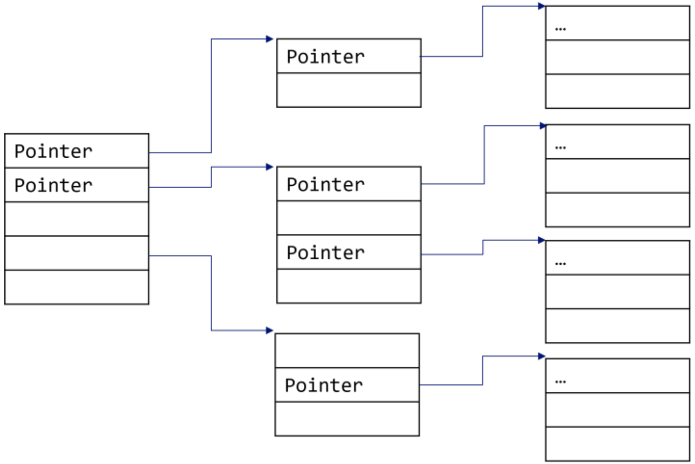


❌ Few members	❌ Large but lacking in structural characteristics	❌ Small graph
< 5 members	< 3 pointers, Booleans, etc	< 4 nodes

Conditions for Successful Identification







✅ Sufficient members & structural characteristics	✅ Has members at high offsets	✅ Large graph
≥ 5 members ≥ 3 pointers, Booleans, etc.	≥ 0x30 offset is a plus	≥ 4 nodes

Security Implications

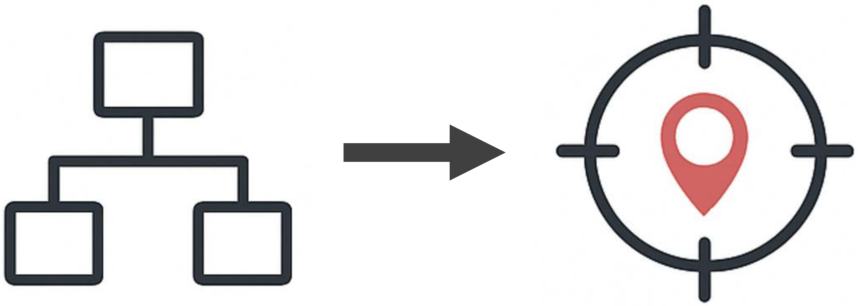
- New potential security risk: **exposing object structure**
  - Object structures now constitute security-sensitive information
  - As attackers increasingly exploit structural characteristics

- Mitigations: **object obfuscation**
  - Pointer encryption
  - Object polymorphism/metamorphism

Takeaways

A New Perspective

Locating Objects by Structural Characteristics

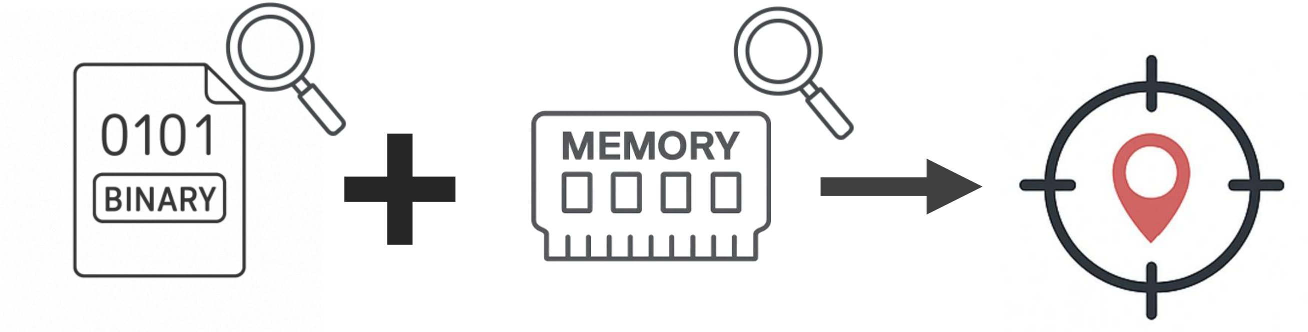


Object location is possible
without distinctive values or leaked pointers
and offsets; **just with structural characteristics**.

- Requires no info leak, no egg hunting. Structural characteristics alone is enough.
- Useful for **Red Teaming, Exploits, and Memory Forensics**.

A Novel Analysis Technique

Binary Analysis + Memory Analysis = Object Location

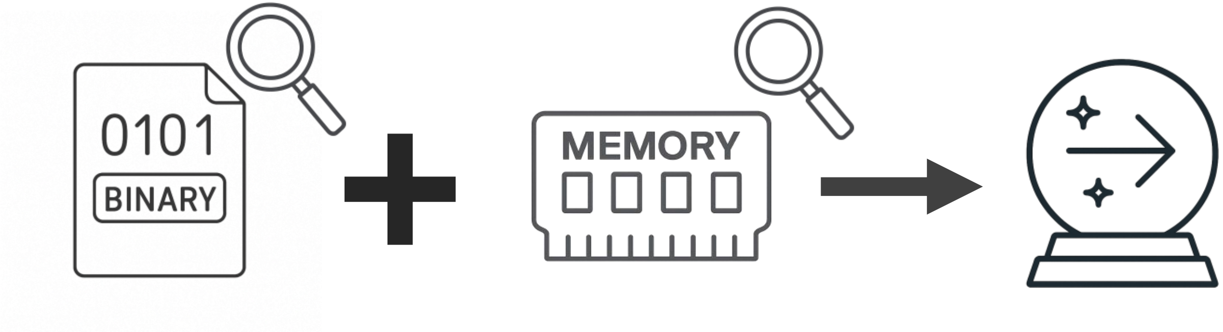


We combined binary analysis and memory analysis
to locate objects precisely with structural characteristics

- Full technical insight shared: details, evaluations, demos.
- Broadly applicable to various target binaries and processes.

A Proposed Future Direction

Integrating Binary & Memory Analysis



The future of reverse engineering lies in
bridging binary and memory analysis (We believe).

- Such integration **still remains undeveloped**, yet essential.
- Our work shows a promising path forward.

Availability

NTT

Our presentation materials, demo videos, and PoC tools will be available here later.

https://github.com/ntt-zerolab/Egg_Hunting_without_Eggs/

Thank you! Q&A?

tos.usui@ntt.com

linkedin.com/in/tusui

@hex86_64