Making Meaning from Binaries

Reachability, Influence, and the Language of Analysis

Nathan Dautenhahn

Dartmouth TrustLab / Serenitix

2025-10-21

- Program analysis
  point-wise
- Where are bugs?


AAAAAAAAAAAAAAA
AAAAAAAAAAAAAAA

What is the meaning of this... *binary*?

Byte view: tiny.o (hex)

Excerpt (full in `demo-duc/build/tiny.hex.txt`):

```
00000000  cf fa ed fe 0c 00 00 01  00 00 00 00 01 00 00 00  |................|
00000010  04 00 00 00 e8 03 00 00  00 20 00 00 00 00 00 00  |......... ......|
00000020  19 00 00 00 68 03 00 00  00 00 00 00 00 00 00 00  |....h...........|
00000030  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
```

- We begin with raw bytes. Next, we disassemble with `objdump -S` to recover structure.

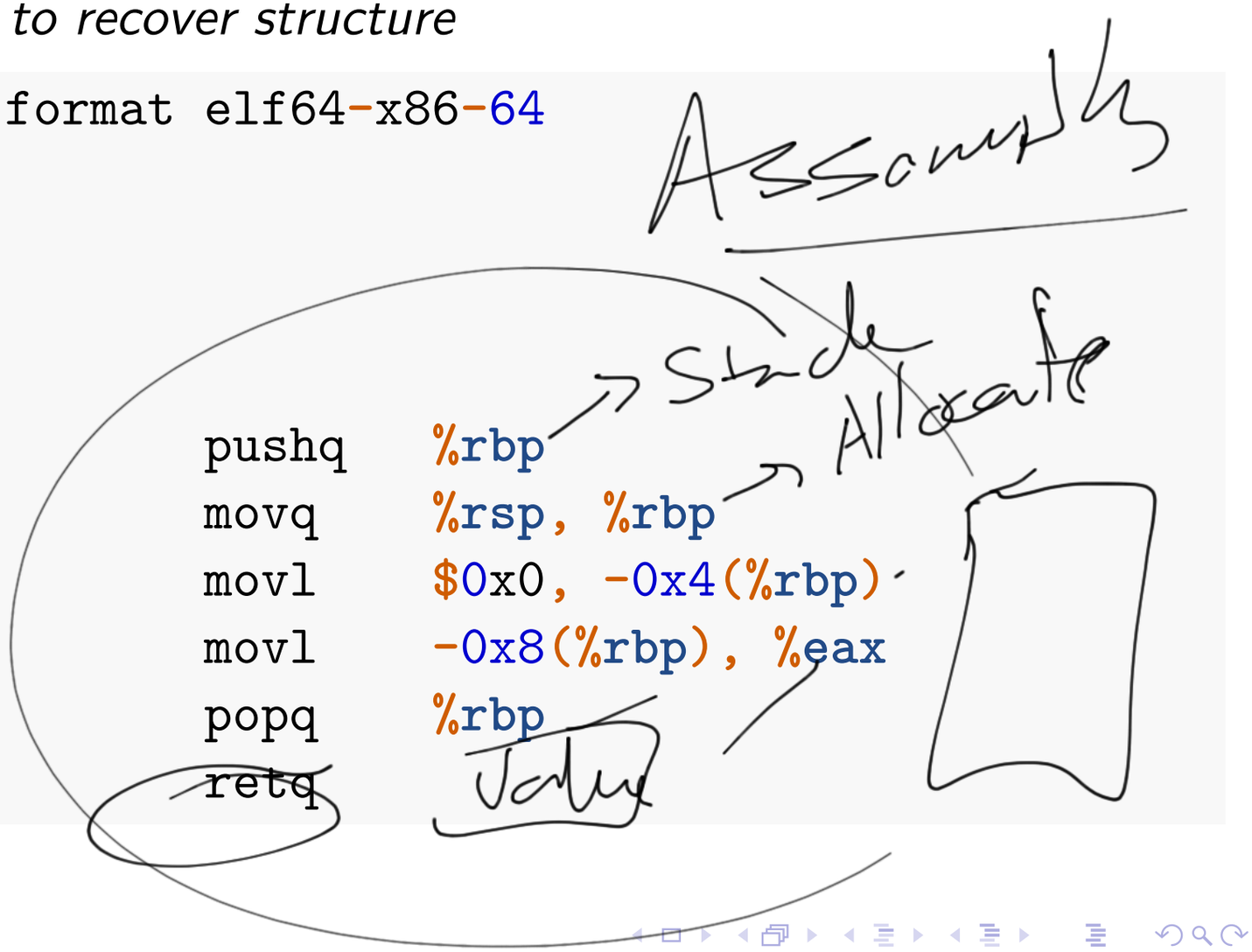
What is the meaning of this... *computation*?

Dissasembled view: tiny.o (`objdump -S`) to recover structure

```
build/elf/tiny.stripped.o:  file format elf64-x86-64

Disassembly of section .text:

0000000000000000 <main>:
       0: 55                    pushq   %rbp
       1: 48 89 e5              movq    %rsp, %rbp
       4: c7 45 fc 00 00 00 00  movl    $0x0, -0x4(%rbp)
       b: 8b 45 f8              movl    -0x8(%rbp), %eax
       e: 5d                    popq    %rbp
       f: c3                    retq
```



What it really means?

```
int main() { int x; return x; }
```

The Fundamental Question

- What does it mean to understand a binary?
- What does it mean to *compute*?
- What does it mean to *understand* a computation?
- [**Key Idea**] Meaning = interpreting **state** at each point of execution.
- What does `C` `int main() { int x; return x; }` mean?

Meaning Through Semantics

- Meaning = values + their interpretation
- Interpretation arises from **types** and **semantics**
- Compilation strips this information away
- Binaries are opaque because semantics are missing

lose symbolic

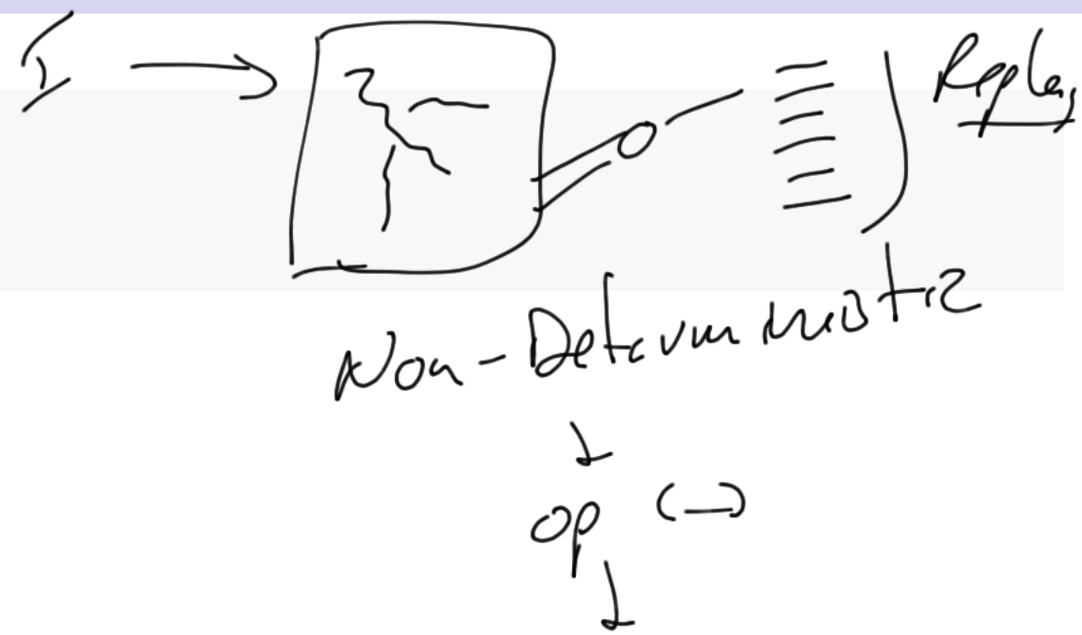
concrete

Why We Care

- Program understanding $\rightarrow$ safe and secure software
- Exploit understanding $\rightarrow$ vulnerability semantics
- Malware understanding $\Rightarrow$ unintended interpretation
- [**Key Insight**] Weird machines on top of weird machines.

State Machine Model of Computation

- ```
  int x;
  printf("%d\n", x);
  ```



- Is bug? Why or why not?
- Is exploitable? Why or why not?
- [**Main point**] It's only bug based on a particular interpretation, ie semantic valuation
- Safety and security are about ensuring we stay in the safe zones
  ```

State Machine Model of Computation

- ```
  int x;
  printf("%d\n", x);
  ```

- ```
  char *p = malloc(10);
  free(p);
  p[0] = 'A';
  ```

- Is bug? Why or why not?
- Is exploitable? Why or why not?
- [**Main point**] It's only bug based on a particular interpretation, ie semantic valuation
- Safety and security are about ensuring we stay in the safe zones

State Machine Model of Computation

- ```
  int x;
  printf("%d\n", x);
  ```

- ```
  char *p = malloc(10);
  free(p);
  p[0] = 'A';
  ```

- ```
  memcpy(buf, data, payload_length);
  ```

- Is bug? Why or why not?
- Is exploitable? Why or why not?
- [**Main point**] It's only bug based on a particular interpretation, ie semantic valuation
- Safety and security are about ensuring we stay in the safe zones
  ```

Three Examples in Context

1. **Uninitialized Read** – missing definition before use

```
int x;
printf("%d\n", x);
```

2. **Use-After-Free** – invalidated definition reused

```
char *p = malloc(10);
free(p);
p[0] = 'A';
```

3. **Unconditioned Flow Path** – missing guard or sanitization (Heartbleed)

```
memcpy(buf, data, payload_length);
```

Generalizing: Correctness as Reachability

- Most analysis = checking state invariants on paths.
- What types of properties do we want to check?:
  - *Type safety*: is dynamic type consistent?
  - *Memory safety*: is object alive at use?
  - *Taint analysis*: is tainted data reachable at sink?
- [**Key Idea**] *Missing Sanitization*: is condition checked before operation?
- [**Key Idea**] Correctness can be defined as reachability over the computation graph

Lowering to Binary

- Compilation removes semantics
  - (types, scope)
- We recover structure by reconstructing:
  - functions
  - variables
  - types
  - names
- DWARF/heuristics help, but incomplete

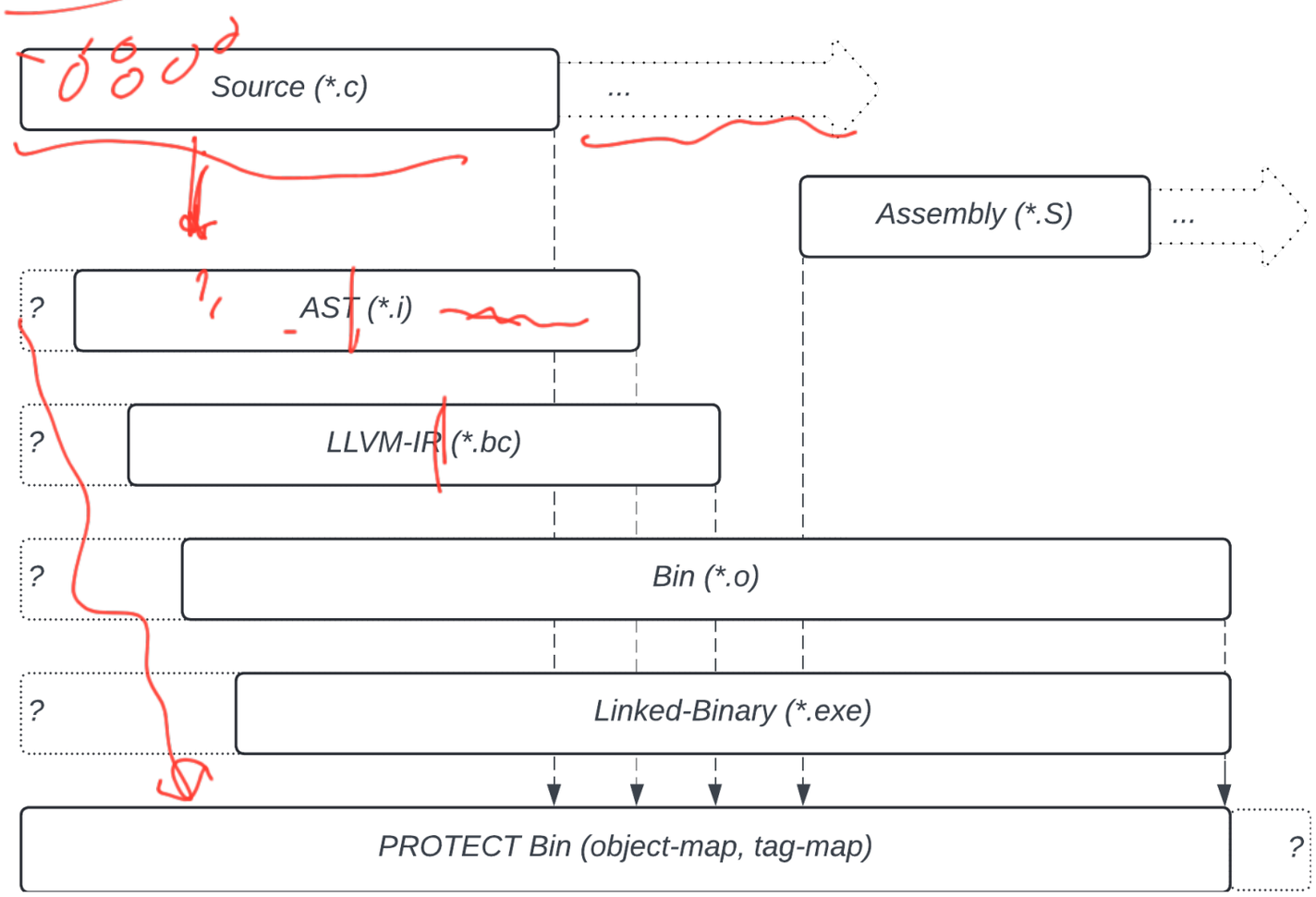


Figure 2: Object provenance planes of origin

Lowering: tiny (Source → objdump -S)

Source

```
int main() { int x; return x; }
```

objdump -S (excerpt)

```
; int main() { int x; return x; }
   sub sp, sp, #0x10
   str wzr, [sp, #0xc]
   ldr w0, [sp, #0x8]
   add sp, sp, #0x10
   ret
```

Note: full disassembly in demo-duc/build/tiny.S.txt

Lowering: uaf (Source → objdump -S)

Source

```c
#include <stdlib.h>
int main(){ char *p=malloc(8); free(p); return p[0]; }
```

objdump -S (excerpt)

```
; int main(){ char *p=malloc(8); free(p); return p[0]; }
   sub sp, sp, #0x20
   stp x29, x30, [sp, #0x10]
   add x29, sp, #0x10
   mov x0, #0x8
   bl  <malloc>
   str x0, [sp]
   ldr x0, [sp]
   bl  <free>
   ldr x8, [sp]
   ldrsb w0, [x8]
```

Common Binary Analyses

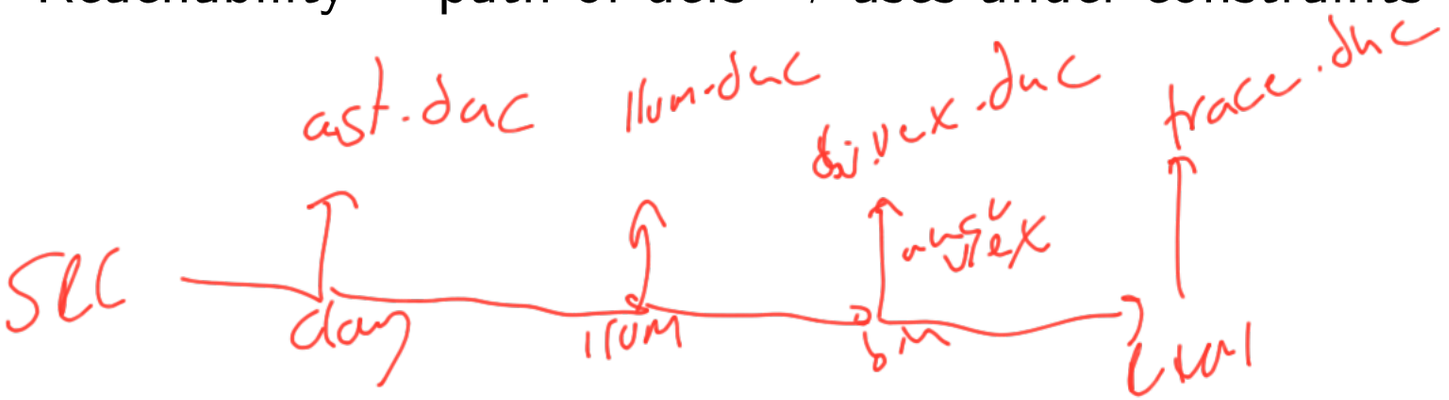
- Reverse engineering $\rightarrow$ structure recovery
- Malware analysis $\rightarrow$ behavior classification
- Fuzzing $\rightarrow$ reachability discovery
- Symbolic execution $\rightarrow$ path constraint reasoning

Recovering Meaning

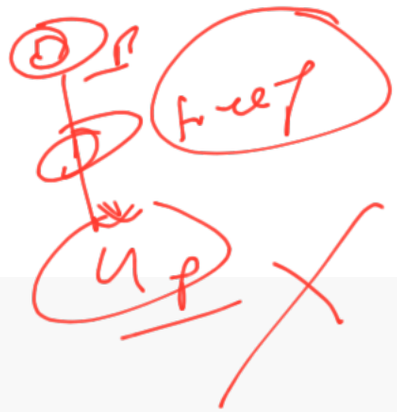
- Need to rebuild symbolic structure: functions, types, variables.
- DWARF helps where available.
- Otherwise, we infer from use/def patterns.

The Language of Analysis (DUC)

- Def–Use Calculus (Def · Use · Expr)
- Uniform representation for state transitions
- Each node = state transition
- Each edge = flow of influence
- Guards = each edge may be guarded by a predicate condstraint
- Each analysis is a query over def/use relations
- Reachability = path of defs → uses under constraints



DUC Examples



Example:

```
void * p = malloc(10);      // def of var p, alloc of object *p
free(p);                    // free of object *p
*p = 0;                     // use of free'd object *p
```

DUC IR (simplified):

```
def p = malloc()     // Alloc *p storage space
def *p = free(p)     // Kill *p
use *p -> write(0)    // Write to old *p which does not exist
```

Tainted Flow Paths: Influence and Reachability Graph

- Given a DUC
- Select any node
- *Influences*: forward slice from node — nefarious source analysis (taintable objects)
- *Influenced-by*: revserse slice — sensitive sink objects (e.g., syscall)
- *Unconditioned Flow Path*: Path from nefarious source that does never sanitized

DUC as Unifying Model

- Same structure supports forward and backward analysis.
- Enables reasoning about values, aliases, and flows uniformly.
- Forms basis for hyperplane framework.

From Meaning to Measurement

- Reachability $\rightarrow$ Exposure metric
- Influence $\rightarrow$ Authority metric
- Quantify: how much can untrusted input affect critical states?

Vision: Hyperplanes of Meaning

- Pipelines: Lift -> Analyze -> Project -> Check
- Planes: Lifters and Projectors
  - Intent plane (programmmer)
  - Source plane (types)
  - Binary plane (values)
  - Runtime plane (observations)
- Lifters:
  - AST, LLVM, Binary, Angr, etc.

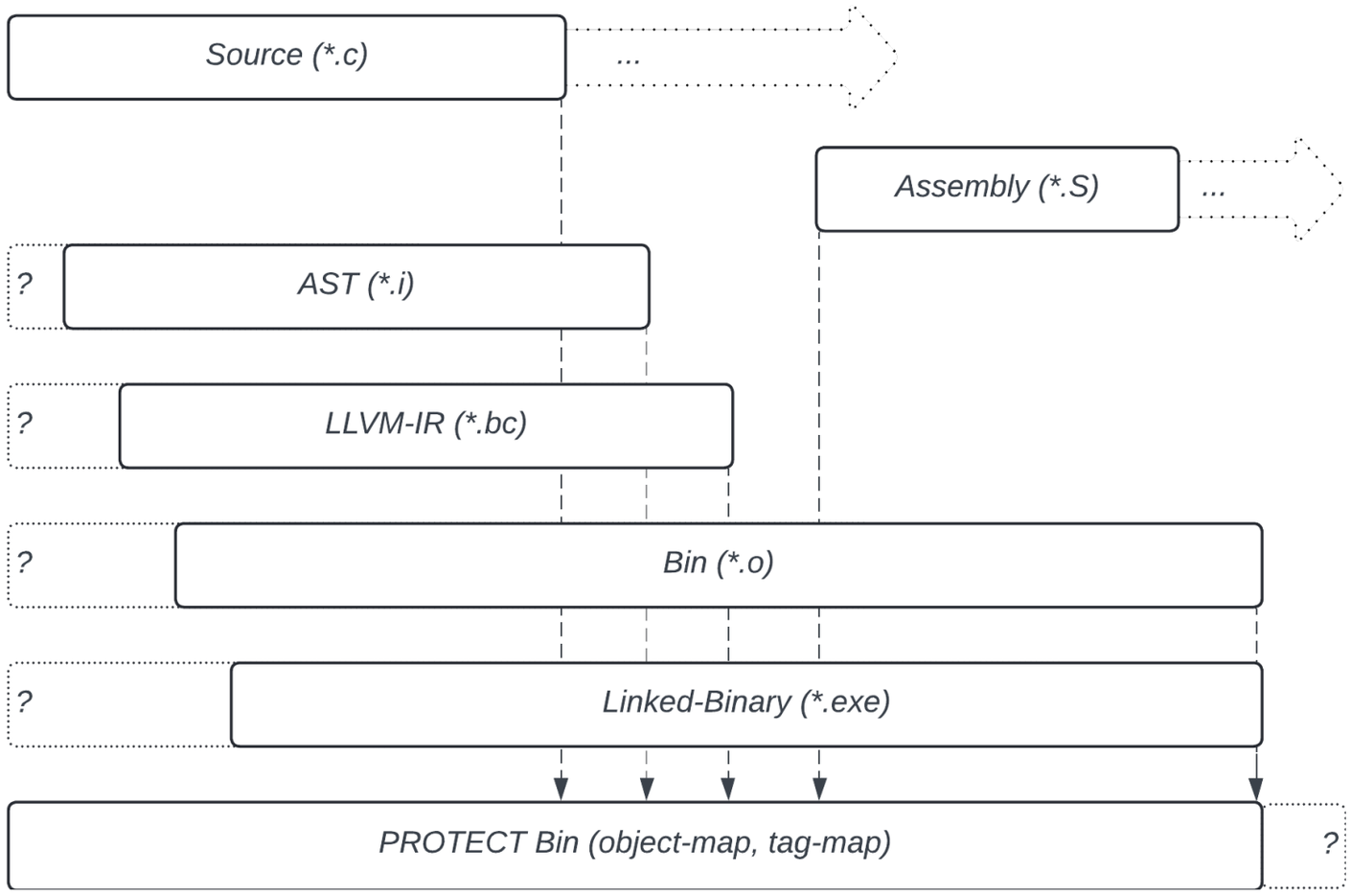


Figure 3: Object provenance planes of origin

Conclusion

- Meaning arises from state and semantics.
- Bugs = broken state invariants along paths.
- Analysis = reasoning about reachability and influence
- Binaries erase meaning; our job is to recover it.
- Def–Use Calculus offers a language to speak that meaning.