SK^{2}DECOMPILE: LLM-BASED TWO-PHASE BINARY DECOMPILATION FROM SKELETON TO SKIN

Hanzhuo Tan[1,2],**Weihao Li**[1],**Xiaolong Tian**[1],**Siyi Wang**[1],**Jiaming Liu**[1],**Jing Li**[2],**Yuqun Zhang**[1,*]
[1]Research Institute of Trustworthy Autonomous Systems,
Southern University of Science and Technology
[2]Department of Computing, The Hong Kong Polytechnic University

Presenter: Jiyong

Decompilation

- In the Paper: *"Reconstruction of source code from binary executables"*

Decompilation

- In the Paper: *"Reconstruction of source code from binary executables"*
- *"Decompilation is the process of turning low-level machine code into higher-level representation."*

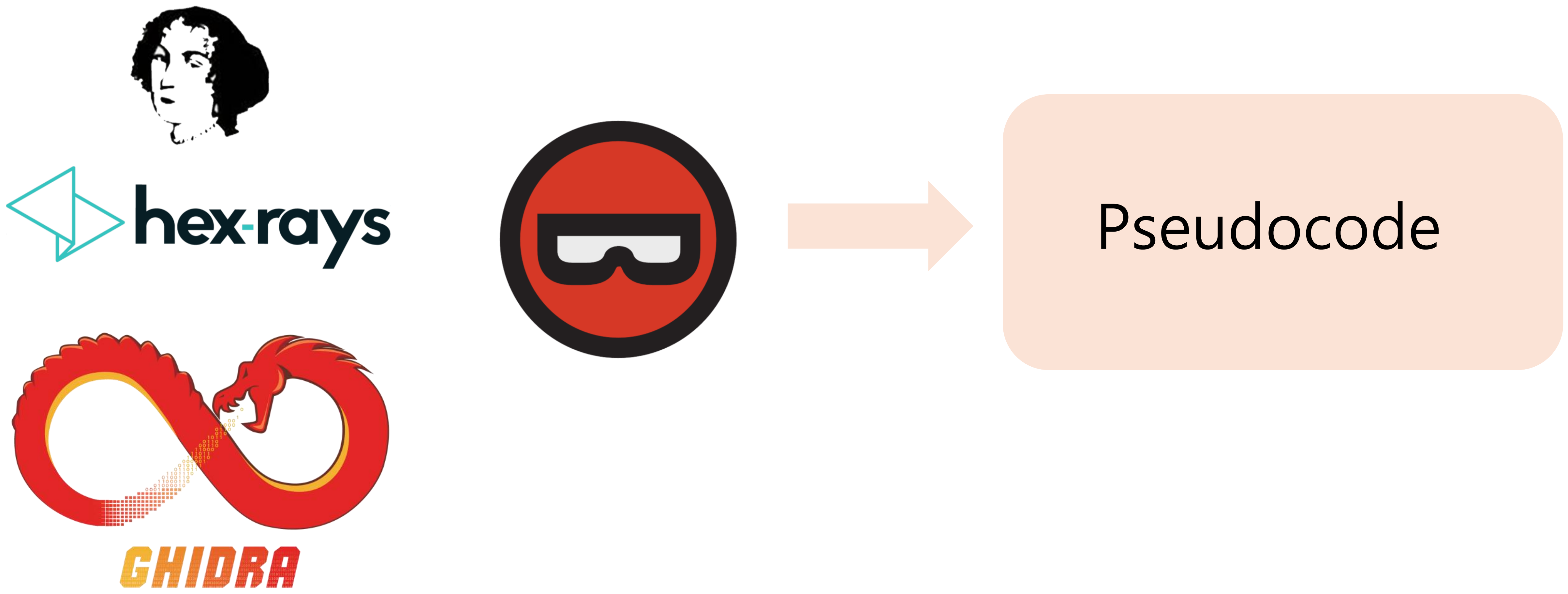
  - (The Decompilation Wiki)

Typical Decompilation Tools

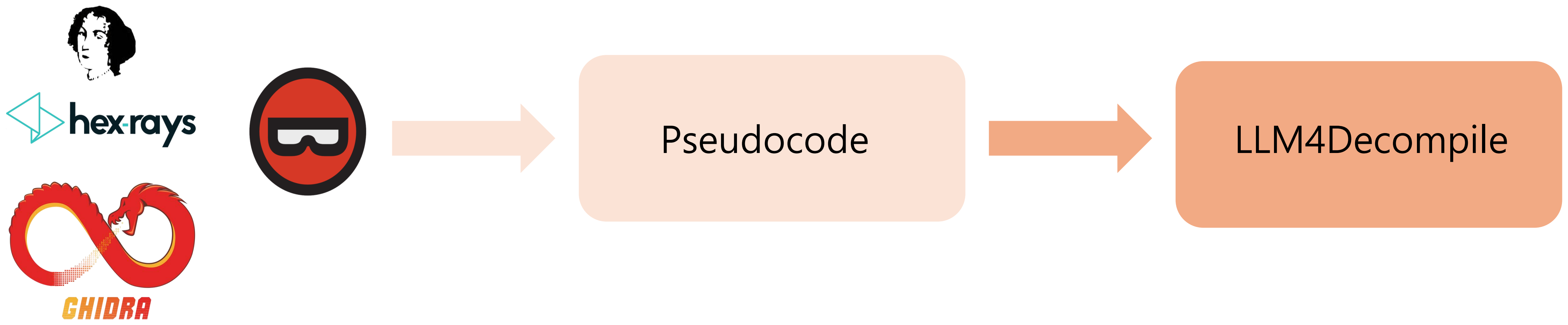
hex-rays

GHIDRA

Pseudocode

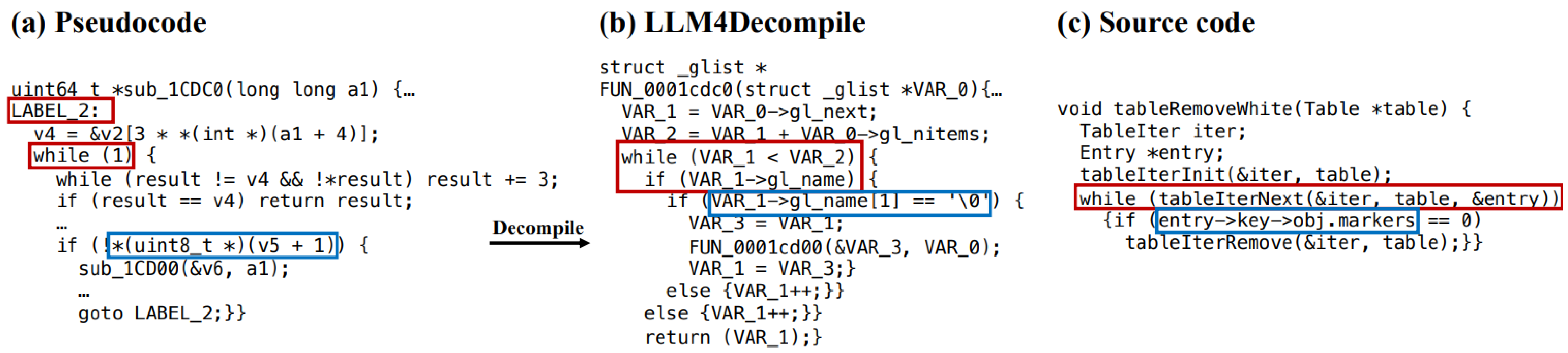


Recent approach (LLM4Decompile*)



* Author's prior work (EMNLP '24)

Problems with LLM4Decompile



(a) Pseudocode

```
uint64_t *sub_1CDC0(long long a1) {…
LABEL_2:
  v4 = &v2[3 * *(int *)(a1 + 4)];
  while (1) {
    while (result != v4 && !*result) result += 3;
    if (result == v4) return result;
    …
    if (!*(uint8_t *)(v5 + 1)) {
      sub_1CD00(&v6, a1);
      …
      goto LABEL_2;}}
```

(b) LLM4Decompile

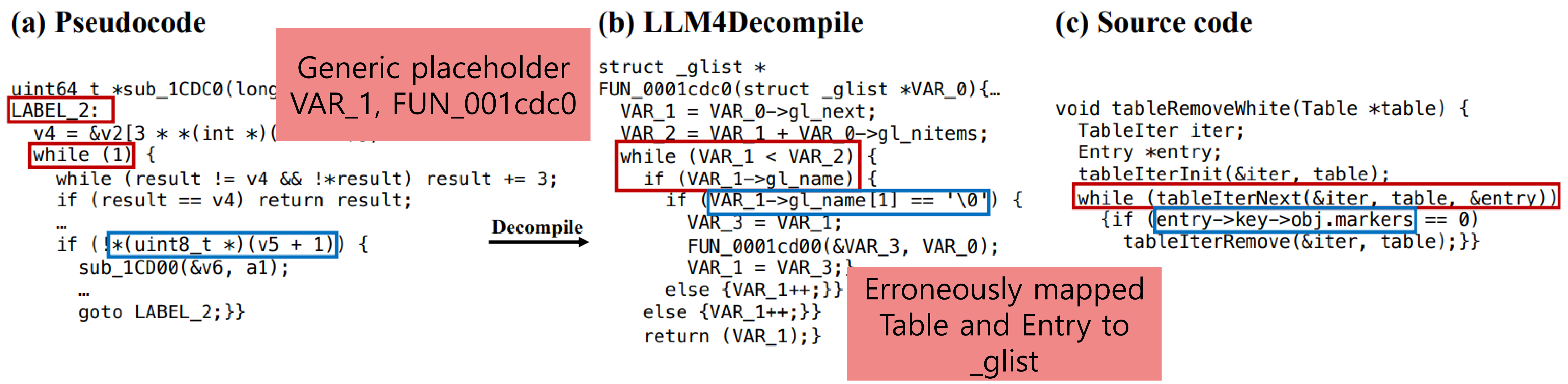
```
struct _glist *
FUN_0001cdc0(struct _glist *VAR_0){…
  VAR_1 = VAR_0->gl_next;
  VAR_2 = VAR_1 + VAR_0->gl_nitems;
  while (VAR_1 < VAR_2) {
    if (VAR_1->gl_name) {
      if (VAR_1->gl_name[1] == '\0') {
        VAR_3 = VAR_1;
        FUN_0001cd00(&VAR_3, VAR_0);
        VAR_1 = VAR_3;}
      else {VAR_1++;}}
    else {VAR_1++;}}
    return (VAR_1);}
```

(c) Source code

```
void tableRemoveWhite(Table *table) {
  TableIter iter;
  Entry *entry;
  tableIterInit(&iter, table);
  while (tableIterNext(&iter, table, &entry))
    {if (entry->key->obj.markers == 0)
      tableIterRemove(&iter, table);}}
```

Figure 1: An example with its (a) pseudocode, (b) refinement by LLM4Decompile, and (c) source code. red marks the `while` loop in different forms, blue represents the `data access`.

Problems with LLM4Decompile



(a) Pseudocode

```
uint64_t *sub_1CDC0(long
LABEL_2:
  v4 = &v2[3 * *(int *)(
  while (1) {
    while (result != v4 && !*result) result += 3;
    if (result == v4) return result;
    …
    if (!*(uint8_t *)(v5 + 1)) {
      sub_1CD00(&v6, a1);
      …
      goto LABEL_2;}}
```

Generic placeholder VAR_1, FUN_001cdc0

Decompile

(b) LLM4Decompile

```
struct _glist *
FUN_0001cdc0(struct _glist *VAR_0){…
  VAR_1 = VAR_0->gl_next;
  VAR_2 = VAR_1 + VAR_0->gl_nitems;
  while (VAR_1 < VAR_2) {
    if (VAR_1->gl_name) {
      if (VAR_1->gl_name[1] == '\0') {
        VAR_3 = VAR_1;
        FUN_0001cd00(&VAR_3, VAR_0);
        VAR_1 = VAR_3;}
      else {VAR_1++;}}
    else {VAR_1++;}}
  return (VAR_1);}
```

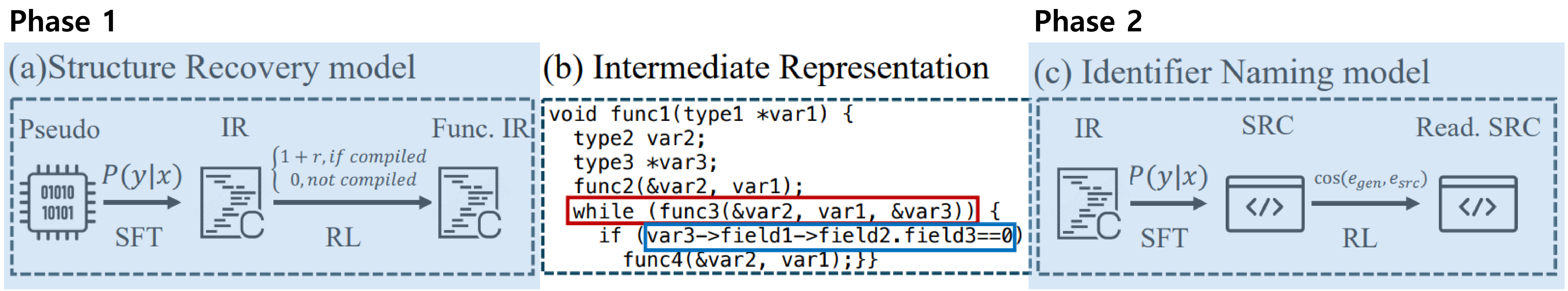
Erroneously mapped Table and Entry to _glist

(c) Source code

```
void tableRemoveWhite(Table *table) {
  TableIter iter;
  Entry *entry;
  tableIterInit(&iter, table);
  while (tableIterNext(&iter, table, &entry))
    {if (entry->key->obj.markers == 0)
      tableIterRemove(&iter, table);}}
```

Figure 1: An example with its (a) pseudocode, (b) refinement by LLM4Decompile, and (c) source code. red marks the while loop in different forms, blue represents the data access.

SK²Decompile (**Sk**eleton-**Sk**in Decompile)


Phase 1
Phase 2
(a)Structure Recovery model
Pseudo
IR
Func. IR
01010
10101
$P(y|x)$
SFT
$\begin{cases} 1+r, \text{if compiled} \\ 0, \text{not compiled} \end{cases}$
RL
(b) Intermediate Representation
(c) Identifier Naming model
IR
SRC
Read. SRC
$P(y|x)$
SFT
$\cos(e_{gen}, e_{src})$
RL

```
void func1(type1 *var1) {
  type2 var2;
  type3 *var3;
  func2(&var2, var1);
  while (func3(&var2, var1, &var3)) {
    if (var3->field1->field2.field3==0)
      func4(&var2, var1);}}
```

Figure 2: Overview of *SK*^{2}*Decompile* with two-phase decompilation process (a) Structure Recovery and (c) Identifier Naming. Obfuscated source code (b) serves as the Intermediate Representation (IR) connecting these two phases. For each model, a supervised fine-tuning (SFT) process is performed followed by Reinforcement Learning (RL) with its respective reward.

Intermediate Representation

- Obfuscated version of the original source code

Intermediate Representation

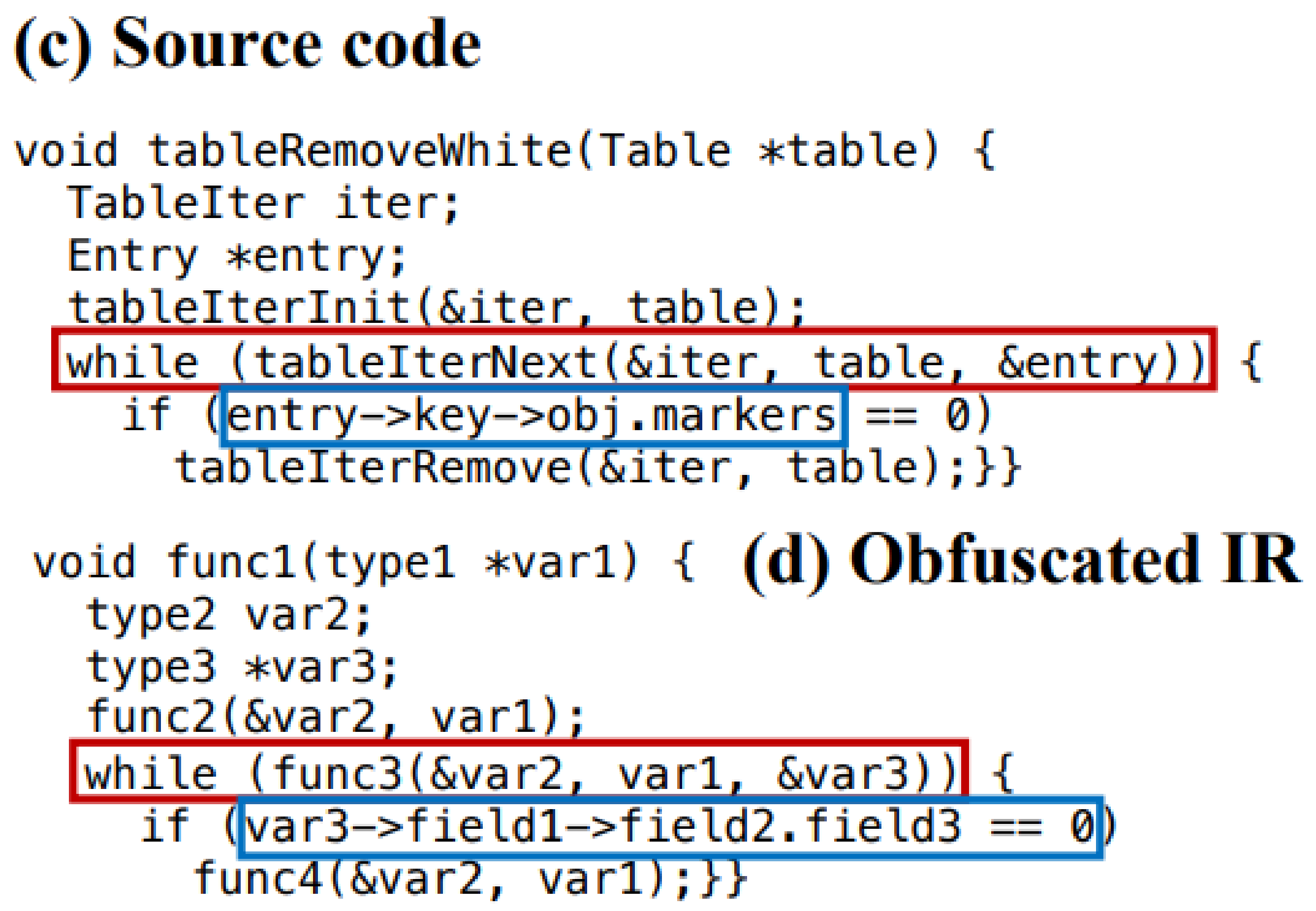
- Obfuscated version of the original source code

Intermediate Representation

- Original source with all identifiers replaced by generic placeholders

- WHY?
  - Discards irrelevant details from pseudocode to make Structure Recovery phase tractable
  - Preserve sufficient information to reconstruct source code in Identifier Naming phase

Intermediate Representation Example



(c) Source code

```
void tableRemoveWhite(Table *table) {
  TableIter iter;
  Entry *entry;
  tableIterInit(&iter, table);
  while (tableIterNext(&iter, table, &entry)) {
    if (entry->key->obj.markers == 0)
      tableIterRemove(&iter, table);}}
```

(d) Obfuscated IR

```
void func1(type1 *var1) {
  type2 var2;
  type3 *var3;
  func2(&var2, var1);
  while (func3(&var2, var1, &var3)) {
    if (var3->field1->field2.field3 == 0)
      func4(&var2, var1);}}
```

Psyche-C

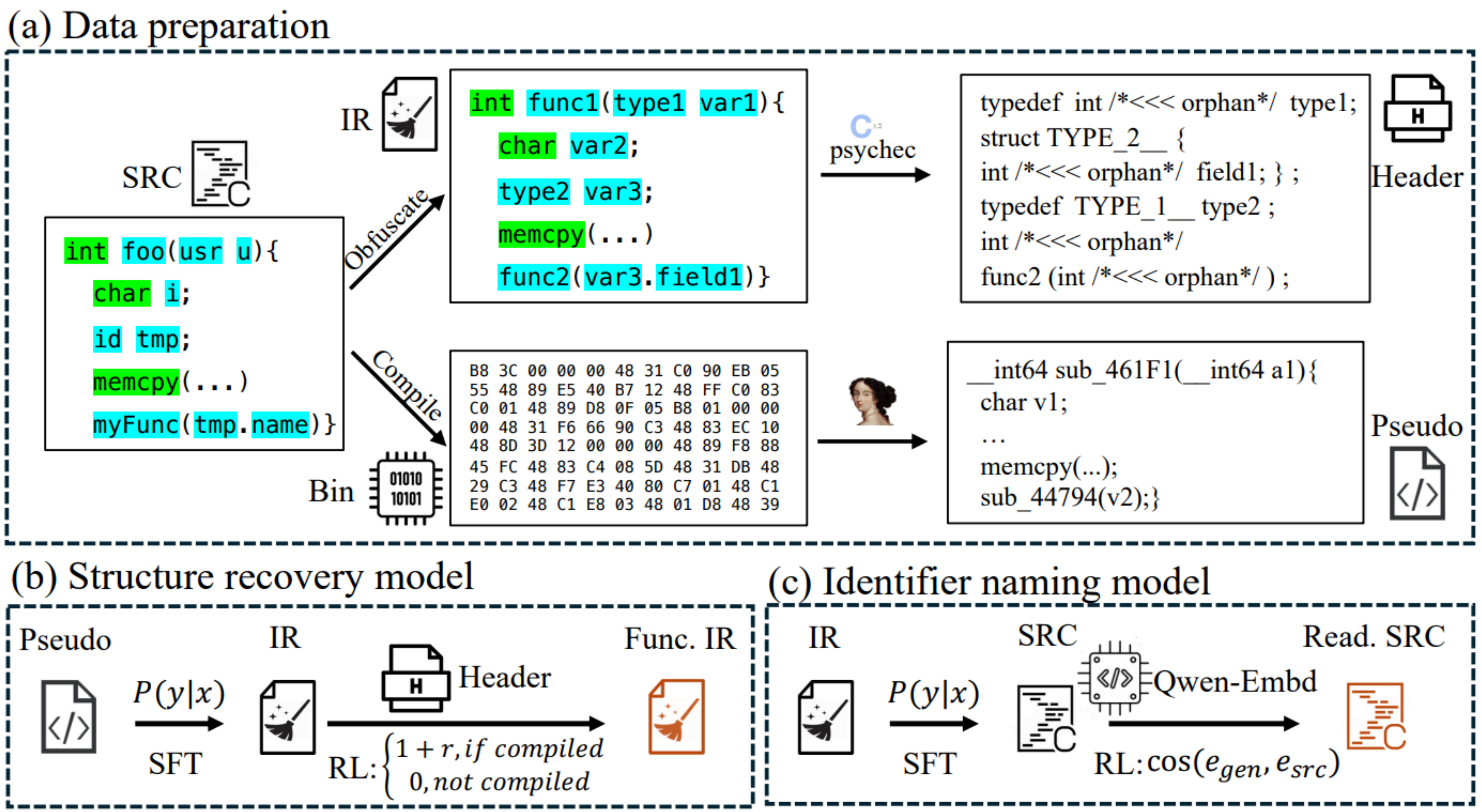
Psyche-C is a platform for implementing static analysis of C programs. At its core, it includes a C compiler frontend that performs both syntactic and semantic analysis. Yet, as opposed to actual C compilers, to provide accurate syntax (through disambiguation) analysis and partial semantic analysis in zero setup or broken build environments, Psyche-C doesn't rely on a symbol table during parsing. Despite this, it nevertheless provides full syntax and semantic analysis for complete source code.

Bellow are the main characteristics of Psyche-C:

- Clean separation between the syntactic and semantic compiler phases.
- Algorithmic and heuristic syntax disambiguation.
- Optional type inference as a recovery mechanism from `#include` failures (not yet in master).
- API inspired by that of the Roslyn .NET compiler and LLVM's Clang.

Enhancements with RL

- Reinforcement Learning (reward system)
  - Structure Recovery
    - For each generated IR, provide header from ground truth (using Psychec-C). Verify compilability and grant a reward only upon success
    - Reward accurate recovery of placeholder identifiers by computing the Jaccard similarity coefficient between the generated and ground-truth sets.
  - Identifier Naming
    - Formulate reward as the semantic similarity between the embedded generated code and the reference source code, measured by cosine similarity


(a) Data preparation
SRC
IR
Obfuscate
Compile
Bin
psychec
Header
Pseudo
(b) Structure recovery model
Pseudo
$P(y|x)$
SFT
IR
Header
Func. IR
RL: $\begin{cases} 1+r, if\ compiled \\ 0, not\ compiled \end{cases}$
(c) Identifier naming model
IR
$P(y|x)$
SFT
SRC
Qwen-Embd
Read. SRC
RL: $\cos(e_{gen}, e_{src})$

```
int foo(usr u){
   char i;
   id tmp;
   memcpy(...)
   myFunc(tmp.name)}
```

```
int func1(type1 var1){
   char var2;
   type2 var3;
   memcpy(...)
   func2(var3.field1)}
```

```
typedef int /*<<< orphan*/ type1;
struct TYPE_2__ {
int /*<<< orphan*/ field1; } ;
typedef TYPE_1__ type2 ;
int /*<<< orphan*/
func2 (int /*<<< orphan*/ ) ;
```

```
B8 3C 00 00 00 48 31 C0 90 EB 05
55 48 89 E5 40 B7 12 48 FF C0 83
C0 01 48 89 D8 0F 05 B8 01 00 00
00 48 31 F6 66 90 C3 48 83 EC 10
48 8D 3D 12 00 00 00 48 89 F8 88
45 FC 48 83 C4 08 5D 48 31 DB 48
29 C3 48 F7 E3 40 80 C7 01 48 C1
E0 02 48 C1 E8 03 48 01 D8 48 39
```

```
__int64 sub_461F1(__int64 a1){
  char v1;
  ...
  memcpy(...);
  sub_44794(v2);}
```

Figure 2: Overview of the *SK^{2}Decompile* framework. (a) Data preparation: We obfuscate identifiers to produce an Intermediate Representation (IR). Headers are inferred using psychec to serve as ground truth for checking compilability during the RL stage of Structure Recovery (b). We also compile the code and use IDA to generate initial pseudo code. *SK^{2}Decompile* employs a two-phase decompilation process comprising (b) Structure Recovery and (c) Identifier Naming, where obfuscated source code serves as the IR connecting the two phases. Each model undergoes Supervised Fine-Tuning (SFT) followed by Reinforcement Learning (RL) with phase-specific rewards.

Re-executability

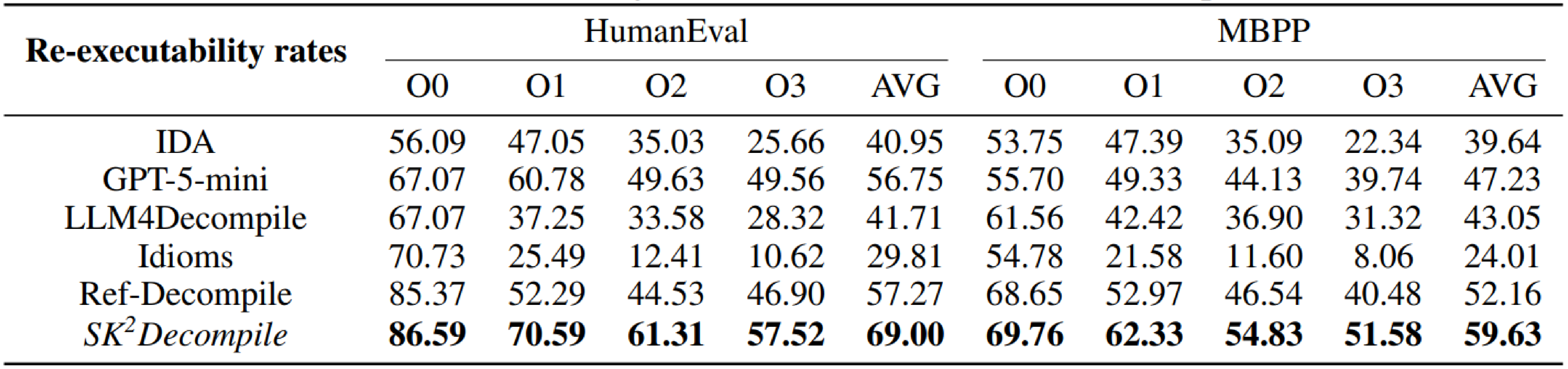


Table 1: Re-executability results between the studied decompilers.

Re-executability rates	HumanEval					MBPP				
	O0	O1	O2	O3	AVG	O0	O1	O2	O3	AVG
IDA	56.09	47.05	35.03	25.66	40.95	53.75	47.39	35.09	22.34	39.64
GPT-5-mini	67.07	60.78	49.63	49.56	56.75	55.70	49.33	44.13	39.74	47.23
LLM4Decompile	67.07	37.25	33.58	28.32	41.71	61.56	42.42	36.90	31.32	43.05
Idioms	70.73	25.49	12.41	10.62	29.81	54.78	21.58	11.60	8.06	24.01
Ref-Decompile	85.37	52.29	44.53	46.90	57.27	68.65	52.97	46.54	40.48	52.16
SK^2Decompile	**86.59**	**70.59**	**61.31**	**57.52**	**69.00**	**69.76**	**62.33**	**54.83**	**51.58**	**59.63**

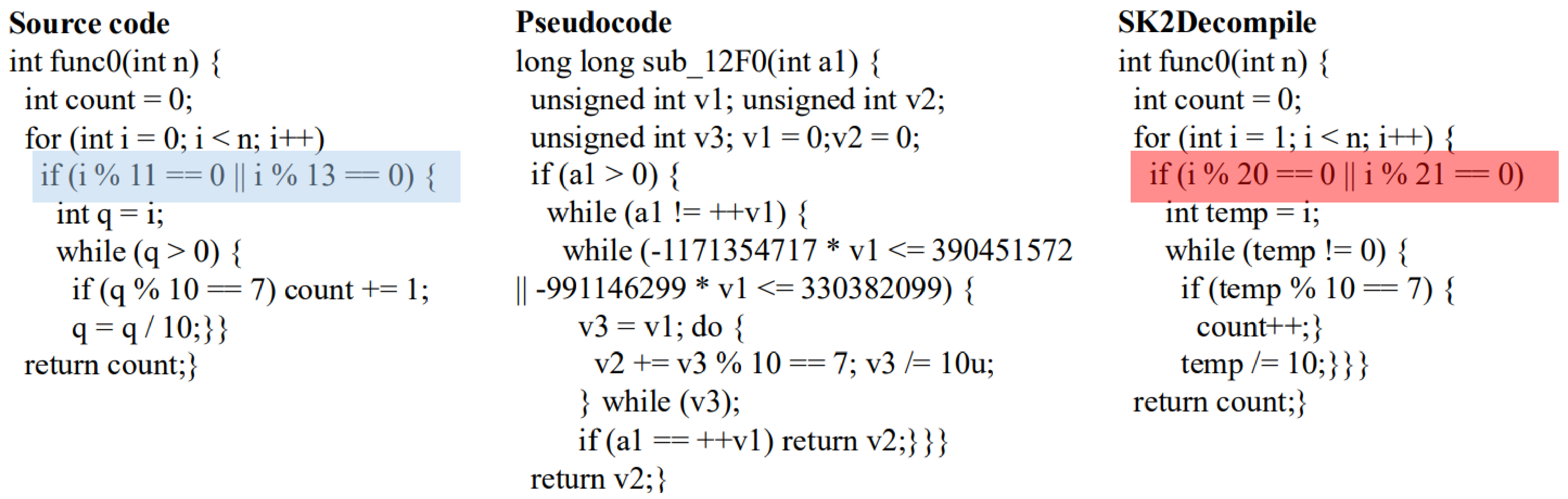
Table 2: R2I results between the studied decompilers with the compilation optimization levels O0, O3 and the averaged results on -O{0,1,2,3}.

R2I	HumanEval			MBPP			ExeBench			GitHub2025		
	O0	O3	AVG	O0	O3	AVG	O0	O3	AVG	O0	O3	AVG
IDA	38.16	40.74	39.45	41.06	34.37	37.72	48.38	51.39	49.89	35.27	43.24	39.26
GPT-5-mini	49.97	37.03	43.49	44.05	31.15	37.60	31.69	28.46	30.08	32.93	27.13	30.03
LLM4Decompile	73.10	72.64	72.87	66.23	72.35	69.29	60.12	57.85	58.99	44.98	53.96	49.47
Idioms	76.60	53.95	65.30	**70.16**	55.74	62.95	**73.37**	54.26	63.82	**71.43**	51.84	61.63
SK^{2}Decompile	**76.62**	**77.72**	**77.17**	69.62	**78.02**	**73.82**	68.75	**77.24**	**72.99**	69.78	**73.45**	**71.62**

Table 3: GPT-judge results between the studied decompilers.

GPT-judge	HumanEval			MBPP			ExeBench			GitHub2025		
	O0	O3	AVG	O0	O3	AVG	O0	O3	AVG	O0	O3	AVG
IDA	3.08	2.67	2.88	3.05	2.57	2.81	2.20	1.91	2.05	2.37	2.19	2.28
GPT-5-mini	4.49	**4.07**	4.23	**4.35**	3.88	4.08	**2.53**	2.33	2.37	3.04	2.86	2.87
LLM4Decompile	3.88	3.29	3.42	3.81	3.22	3.41	2.47	2.12	2.22	2.52	2.56	2.62
Idioms	4.30	2.70	3.22	4.07	2.61	3.13	2.46	1.71	2.01	2.51	2.10	2.18
SK^{2}Decompile	**4.51**	4.05	**4.24**	4.31	**3.95**	**4.12**	2.48	**2.47**	**2.42**	**3.05**	**3.02**	**3.06**

SK2Decompile Limitations



Source code

```
int func0(int n) {
 int count = 0;
 for (int i = 0; i < n; i++)
   if (i % 11 == 0 || i % 13 == 0) {
     int q = i;
     while (q > 0) {
       if (q % 10 == 7) count += 1;
       q = q / 10;}}
 return count;}
```

Pseudocode

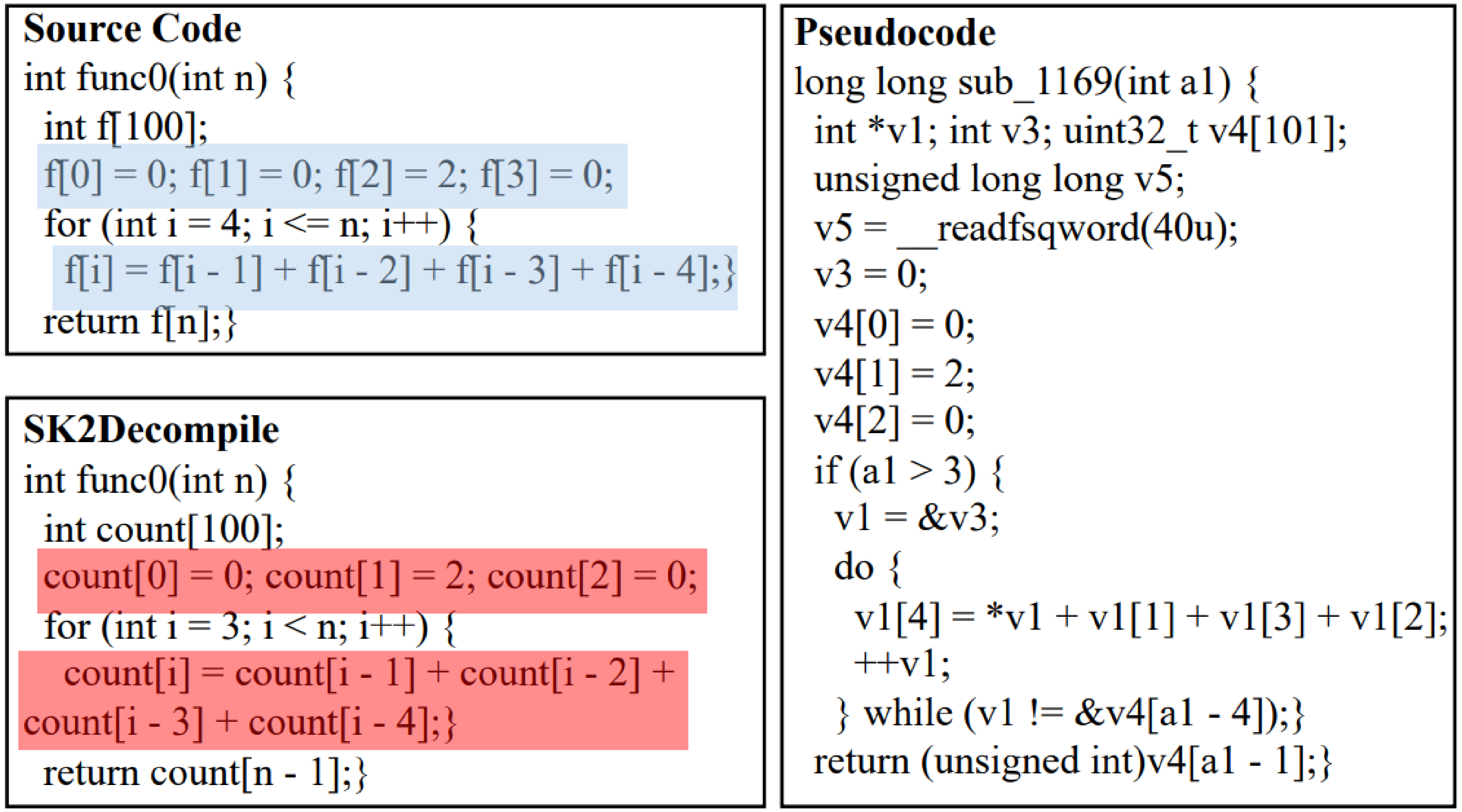
```
long long sub_12F0(int a1) {
 unsigned int v1; unsigned int v2;
 unsigned int v3; v1 = 0;v2 = 0;
 if (a1 > 0) {
   while (a1 != ++v1) {
     while (-1171354717 * v1 <= 390451572
|| -991146299 * v1 <= 330382099) {
       v3 = v1; do {
         v2 += v3 % 10 == 7; v3 /= 10u;
       } while (v3);
       if (a1 == ++v1) return v2;}}}
 return v2;}
```

SK2Decompile

```
int func0(int n) {
 int count = 0;
 for (int i = 1; i < n; i++) {
   if (i % 20 == 0 || i % 21 == 0)
     int temp = i;
     while (temp != 0) {
       if (temp % 10 == 7) {
         count++;}
       temp /= 10;}}}
 return count;}
```

Figure 8: Qualitative assessment on arithmetic precision.

SK2Decompile Limitations



Source Code

```
int func0(int n) {
  int f[100];
  f[0] = 0; f[1] = 0; f[2] = 2; f[3] = 0;
  for (int i = 4; i <= n; i++) {
    f[i] = f[i - 1] + f[i - 2] + f[i - 3] + f[i - 4];}
  return f[n];}
```

SK2Decompile

```
int func0(int n) {
  int count[100];
  count[0] = 0; count[1] = 2; count[2] = 0;
  for (int i = 3; i < n; i++) {
    count[i] = count[i - 1] + count[i - 2] +
count[i - 3] + count[i - 4];}
  return count[n - 1];}
```

Pseudocode

```
long long sub_1169(int a1) {
  int *v1; int v3; uint32_t v4[101];
  unsigned long long v5;
  v5 = __readfsqword(40u);
  v3 = 0;
  v4[0] = 0;
  v4[1] = 2;
  v4[2] = 0;
  if (a1 > 3) {
    v1 = &v3;
    do {
      v1[4] = *v1 + v1[1] + v1[3] + v1[2];
      ++v1;
    } while (v1 != &v4[a1 - 4]);}
  return (unsigned int)v4[a1 - 1];}
```

Figure 7: Qualitative assessment on pattern rarity.

Thoughts

- Decompilation does not need to generate the exact source
    - Depending on the final goal, we may just want to better understand the software

- Compiler optimization may result in decompiled code to be quite dissimilar with source code
    - Inlining