Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心


# 二进制文件复合数据类型恢复

硕士研究生 高玺凯

2025年08月31日

# 问题回溯

- 总结反思
  - 语气比较平淡
  - 算法原理讲解不够细致

- 相关内容
  - 2025.07.13 徐程柯《第三方库检测技术研究》
  - 2025.04.06 高玺凯《二进制代码反编译技术》

# 内容提要

- 预期收获
- 内容引入
- 内涵解析与研究目标
- 研究背景与研究意义
- 研究历史与现状
- 算法原理
  - TypeForge
- 特点总结与未来展望
- 参考文献

# 预期收获

- 预期收获
  - 掌握二进制文件复合数据类型恢复的基本概念
  - 理解二进制文件复合数据类型恢复的研究背景和研究意义
  - 了解二进制文件复合数据类型恢复的前沿方法和未来发展

# 内容引入



编译

汇编

```
int func0(float num[], int size,
  float threshold) {
  int i, j;
  for (i = 0; i < size; i++)
    for (j = i + 1; j < size; j++)
      if (fabs(num[i] - num[j])
          < threshold)
        return 1;
  return 0;}
```

源代码
（面向人类的）

反编译结果

```
undefined4 func0(float param_1,long param_2,int param_3){
  int local_28;
  int local_24;
  local_24 = 0;
  do {
    local_28 = local_24;
    if (param_3 <= local_24) {
      return 0;}
  while (local_28 = local_28 + 1, local_28 < param_3) {
    if ((double)((ulong)(double)(*(float *)(param_2 + (long)local_24 * 4) -
        *(float *)(param_2 + (long)local_28 * 4)) &
        SUB168(_DAT_00402010,0)) < (double)param_1) {
      return 1;}}
    local_24 = local_24 + 1;
  } while( true );}
```

反编译

反汇编

Hex-Rays
STATE-OF-THE-ART BINARY CODE ANALYSIS SOLUTIONS

# 内涵解析与研究目标

- 内涵解析
  - 二进制文件：源代码经编译生成的可执行文件
    - 符号表和调试信息通常被剥离，缺失变量名称和类型信息等
  - 复合数据类型恢复（Composite Data Types Recovery）
    - 根据剥离二进制文件中的内存访问和寄存器使用，推断并重建源级复合数据类型声明的过程
    - 基本数据类型：int、char、float等
    - 复合数据类型：structure、union等

```
struct buffer {
    uint id;
    struct buf_meta* meta;
    struct buf_chunk chunks[2];
(iii) union {
        uint8_t flag;
        int error_code;
    } state;
};
```

(i)

(ii)

- 研究目标
  - 结合程序分析、深度学习等理论
  - 还原编译过程中丢失的高级语义信息，提升反编译代码的可读性，并支持漏洞检测、恶意软件分析、程序理解等下游安全任务

# 研究背景与研究意义

- 研究背景
  - 现实世界的软件通常以剥离后的二进制形式发布，缺失高级语义信息，为程序理解、反编译和安全分析带来了极大挑战
  - 统计数据显示，二进制文件中对复合数据类型变量及其成员的访问次数约为基本类型变量的1.8倍，在复杂软件中比例更高
  - 大量研究工作和现有工具（Ghidra、IDA Pro等）在基本数据类型恢复上表现良好，但在复合数据类型恢复上准确率极低，且仅能处理库定义类型，用户自定义类型仍难以恢复
- 研究意义
  - 推动二进制分析方法的发展，从基本数据类型恢复迈向复合数据类型恢复
  - 增强反编译代码的可读性，降低人工逆向成本和对专家经验的依赖
  - 有效支持反编译、漏洞检测、恶意软件分析等下游任务，提高分析效率

# 研究历史与现状

**基于静态规则的方法**

Lee等人提出TIE。该方法以形式化的类型重建理论为基础，将二进制代码提升至中间语言BIL，并通过DVSA算法进行变量恢复；随后为每个变量分配类型变量，生成并求解约束，结合子类型理论和格结构进行类型推断

**基于静态概率推理的方法**

Zhang等人提出OSPREY。该方法将变量与数据结构恢复视为概率推理问题，将内存位置建模为随机变量，结合扩展的BDA收集的数据流、points-to关系及访问模式生成多种结构提示；再将这些提示转化为概率约束，构建概率图模型并迭代求解

**基于LLM的方法**

Xie等人提出ReSym。该方法结合LLM与轻量级程序分析，其核心是将任务拆分为局部变量与结构体字段恢复，分别微调LLM，并通过Prolog推理与相似性投票聚合结果以提升一致性。实验表明，ReSym在变量名、类型和结构体恢复精度上均优于现有方法

**基于LLM的方法**

Wang等人提出TypeForge。该方法采用“两阶段合成–选择”策略：先利用类型流图与冲突感知传播合成候选类型，再结合LLM辅助的淘汰机制选择最优类型以提升反编译可读性

2010 2011 2016 2021 2022 2024 2024 2025

Lin等人提出REWARDS。该方法在二进制执行中为内存位置打上时间戳类型标记，并利用数据流传播和“类型汇点”进行解析，结合前向与后向推断解决变量复用问题。通过离线补充、变量抽象和内存视图构建，方法能恢复数据结构的语法与语义

**基于动态执行的方法**

Noonan等人提出ReTypd。该方法在二进制的中间表示上生成约束，利用支持多态、递归类型和子类型的形式化系统进行推理，并通过启发式策略将结果映射为可读的C类型。实验表明，ReTypd在类型精度和指针层级恢复上优于现有方法

**基于静态规则的方法**

Chen等人提出DIRTY。该方法在反编译结果上引入Transformer，联合预测变量类型与名称；通过数据布局编码器利用变量大小与偏移信息修正预测，并在多任务解码器中利用类型与名称的相关性提升一致性与准确率

**基于深度学习的方法**

Zhu等人提出TYGR。该方法在ANGR平台上将二进制提升至VEX IR，并结合轻量级数据流分析生成函数级数据流图，输入到GNN以推断变量类型。其创新点在于设计了基于图的数据流表示，兼顾可扩展性与准确性

**基于深度学习的方法**

# TypeForge: Synthesizing and Selecting Best-Fit Composite Data Types for Stripped Binaries

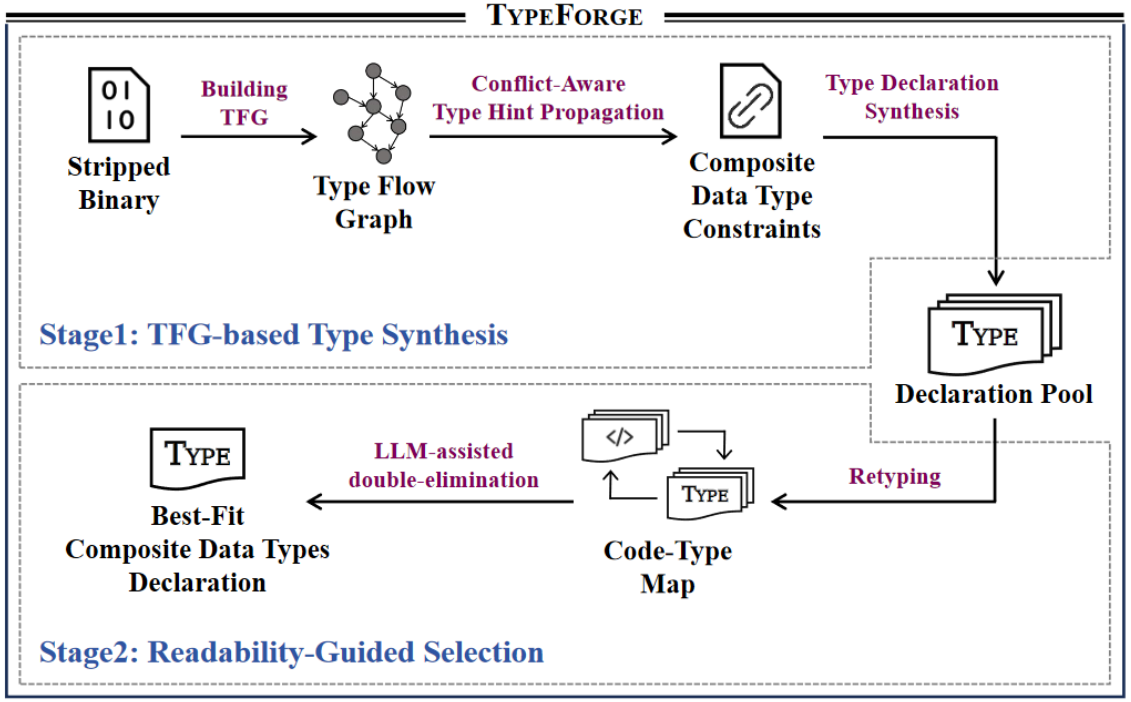
# TypeForge TIPO

| | | |
|---|---|---|
| T | 目标 | 从剥离的二进制文件中恢复复合数据类型声明 |
| I | 输入 | 剥离的二进制文件*1 |
| P | 处理 | 1. 基于TFG生成复合数据类型声明候选<br>2. 基于可读性选择最终复合数据类型声明 |
| O | 输出 | 恢复的复合数据类型声明（包括布局、关系和相关变量）*n |

| | | |
|---|---|---|
| P | 问题 | 现有方法难以高效准确地构建复合数据类型约束，无法在准确率与效率之间取得平衡；难以处理约束歧义，即同一约束可能对应多种复合类型声明 |
| C | 条件 | 剥离的二进制文件可被正常反编译（未加密、未加壳） |
| D | 难点 | 如何高效准确地构建符合复合数据类型约束；<br>如何从候选中准确选择最契合的复合类型声明 |
| L | 水平 | S&P 2025 |

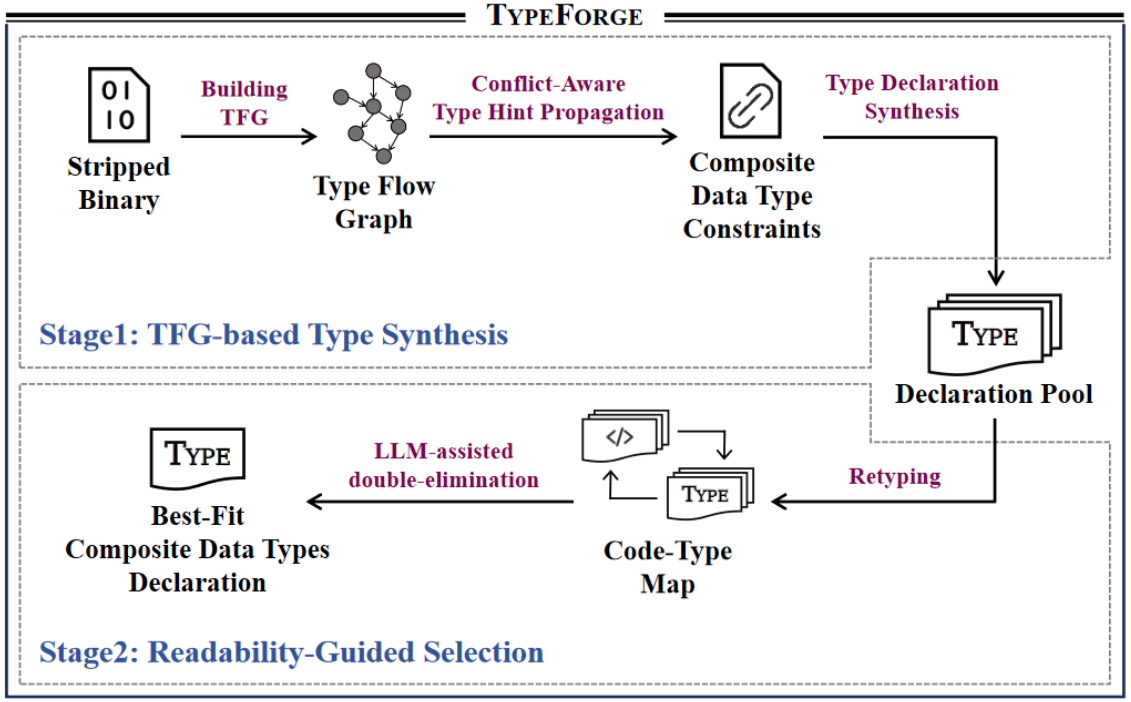
# TypeForge 算法原理

- 算法原理图
  - 动机
    - 逆向工程专家在手动恢复复合数据类型时通常会：首先基于文件中的部分类型提示（如内存访问模式）合成初始复合数据类型声明；随后不断分析文件，迭代完善和修改此初始声明，在面对不确定性时保留多个方案作为候选；最终凭借经验选出最契合的复合类型声明
  - 阶段1：基于TFG（Type Flow Graph）的类型合成
  - 阶段2：基于可读性的选择



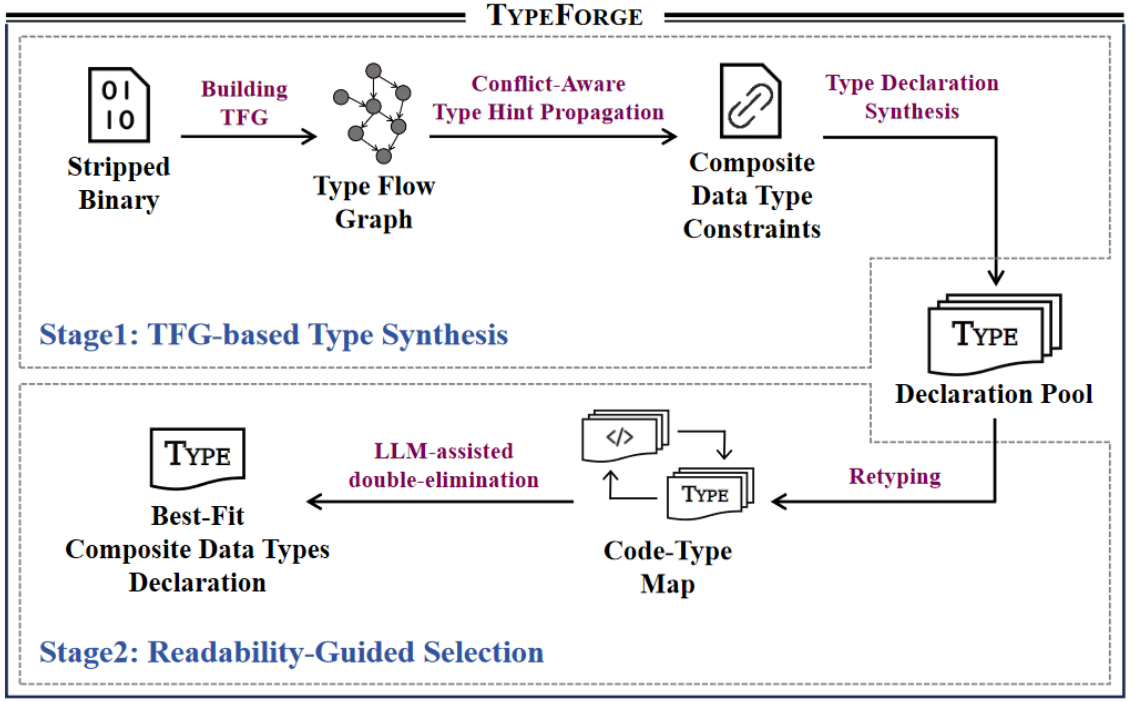
# TypeForge 算法原理

- 算法原理图
  - 基于TFG的类型合成
    - TFG构建：利用反编译器推断出的变量和基本数据类型，构建程序的TFG，以表示整个程序中的潜在类型关系
    - 约束生成：在全程序TFG上应用冲突感知的类型提示传播（Conflict-Aware Type Hint Propagation），以高效地构建精确的复合数据类型约束
    - 候选合成：使用自适应滑动窗口算法（Adaptive Sliding Window），在上述约束的基础上合成可能的复合数据类型声明作为候选



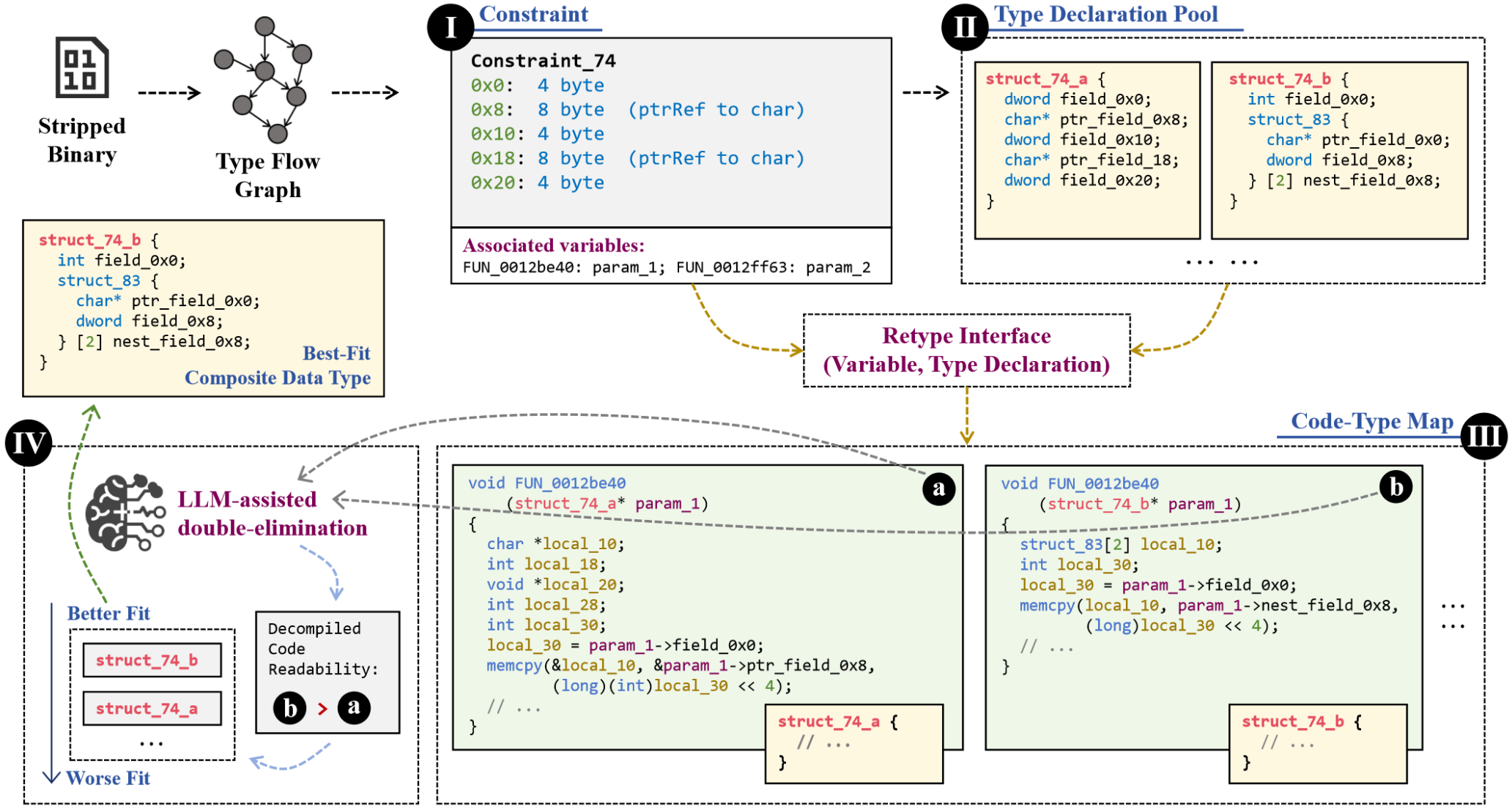
# TypeForge 算法原理

- 算法原理图
  - 基于可读性的选择
    - 构建“反编译代码—类型声明”映射：将候选类型声明提交给反编译器的retype接口，以获得不同版本的反编译代码
    - 选择最终复合数据类型声明：使用LLM辅助的双淘汰机制，对这些代码版本进行可读性评估，最终选择出可读性最高的版本，其对应的类型声明即为最终的最优复合数据类型声明



# TypeForge 算法原理

- 示例：从一个开源项目中恢复出复杂结构体



# TypeForge 基于TFG的类型合成

- 应对问题：现有方法难以高效且精确地构建复合数据类型约束，无法在准确率与效率之间取得平衡
  - 数据流分析过于复杂低效或不够精确
  - 例如：复杂的过程间值集分析或二进制依赖分析
- 解决方案
  - TFG构建：基于反编译器推断出的变量与基本数据类型，设计一种新的数据流抽象TFG，高效汇聚刻画复合类型内部布局与类型间关系的类型提示
  - 约束生成：在TFG上应用冲突感知的类型提示传播，通过冲突检测有效阻止类型提示的误传
  - 候选合成：针对约束使用自适应滑动窗口算法，合成所有可能的复合数据类型声明

# TypeForge 基于TFG的类型合成

- TFG构建
  - 动机
    - 反编译代码中，对复合类型的成员访问通常表示为varleft = *(varright + offset)
      - varleft和varright是由反编译器推断出的变量
      - varright为复合类型的基指针，offset为该类型中某个成员的位置，该成员的大小对应于varleft的大小
    - 由反编译器推断出的存在直接数据流关系的多数变量，通常共享相同数据类型
      - 除类型转换和联合体外

```
1  void Fun_02d3(long param_1) {
2    // ...
3    long local_10;
4    undefined8* local_18;
5    undefined8 local_50;
6    long uVar;
7    // ...
8    Fun_0413(&local_50, param_1);
9    local_10 = param_1;
10   local_18 = *(local_10 + 0x30);
11   uVar = *(local_18 + 0x8);
12   // ...
13 }
```

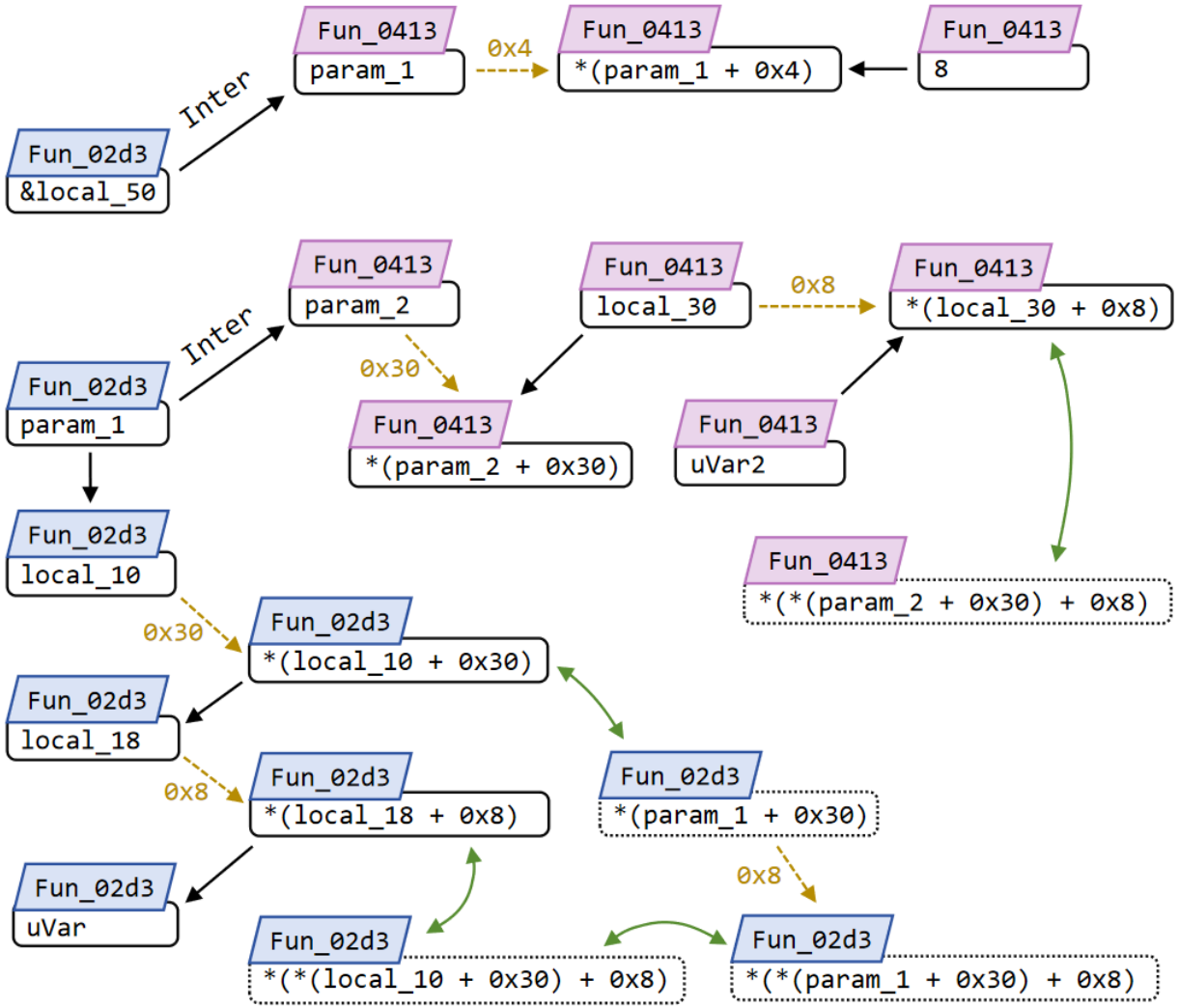
```
void Fun_0413
  (long param_1, long param_2) {
  // ...
  long local_30;
  long uVar2;
  // ...
  *(param_1 + 0x4) = 8;
  *(param_2 + 0x30) = local_30;
  *(local_30 + 0x8) = uVar2;
  // ...
}
```

隐含表达式：

uVar = *(*(local_10 + 0x30) + 0x8)

# TypeForge 基于TFG的类型合成

- TFG构建
  - TFG定义为元组$G = (N, E)$
    - $N$为节点集合
      - 节点$n \in N$为一个嵌套成员访问表达式
    - $E$为有向边集合
      - dataflow类型：表示直接数据流关系（黑色）
      - member类型：表示成员关系，边上的标注表示偏移量（黄色）
      - typealias类型：表示节点具有相同类型（别名）（绿色）



# TypeForge 基于TFG的类型合成

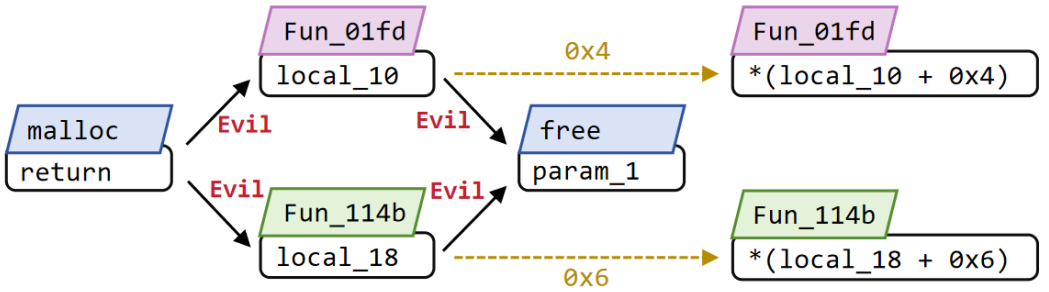
- 约束生成
  - 应对问题：由于反编译器无法识别类型转换和联合体，TFG中由dataflow或typealias边连接的节点并不总是类型一致，使得虚假类型提示被错误传播，导致使约束构建不准确
  - 动机
    - 不同的复合数据类型具有不同的大小，这些大小通常可以在函数调用点处收集到
    - 不同的复合数据类型具有不同的成员布局
  - 在TFG上应用冲突感知的类型提示传播
    - 通过大小信息和布局信息的双重冲突检测，识别并移除所有恶性边（evil edges）
      - 不再进一步区分，把可能冲突的边都删掉

```
1  void handle_config() {
2    ds_cfg* cfg;
3    cfg = (ds_cfg*)malloc(0x40);
4    cfg->id = rand();
5    // ...
6    free(cfg);
7  }
8
9  void handle_stat() {
10   ds_stat* stat;
11   stat = (ds_stat*)malloc(0x10);
12   stat->state = 0xFF;
13   // ...
14   free(stat);
15 }
```

```
void Fun_01fd() {
  void* local_10;
  local_10 = malloc(0x40);
  *(local_10 + 0x4) = rand();
  // ...
  free(local_10);
}

void Fun_114b() {
  void* local_18;
  local_18 = malloc(0x10);
  *(local_18 + 0x6) = 0xFF;
  // ...
  free(local_18);
}
```

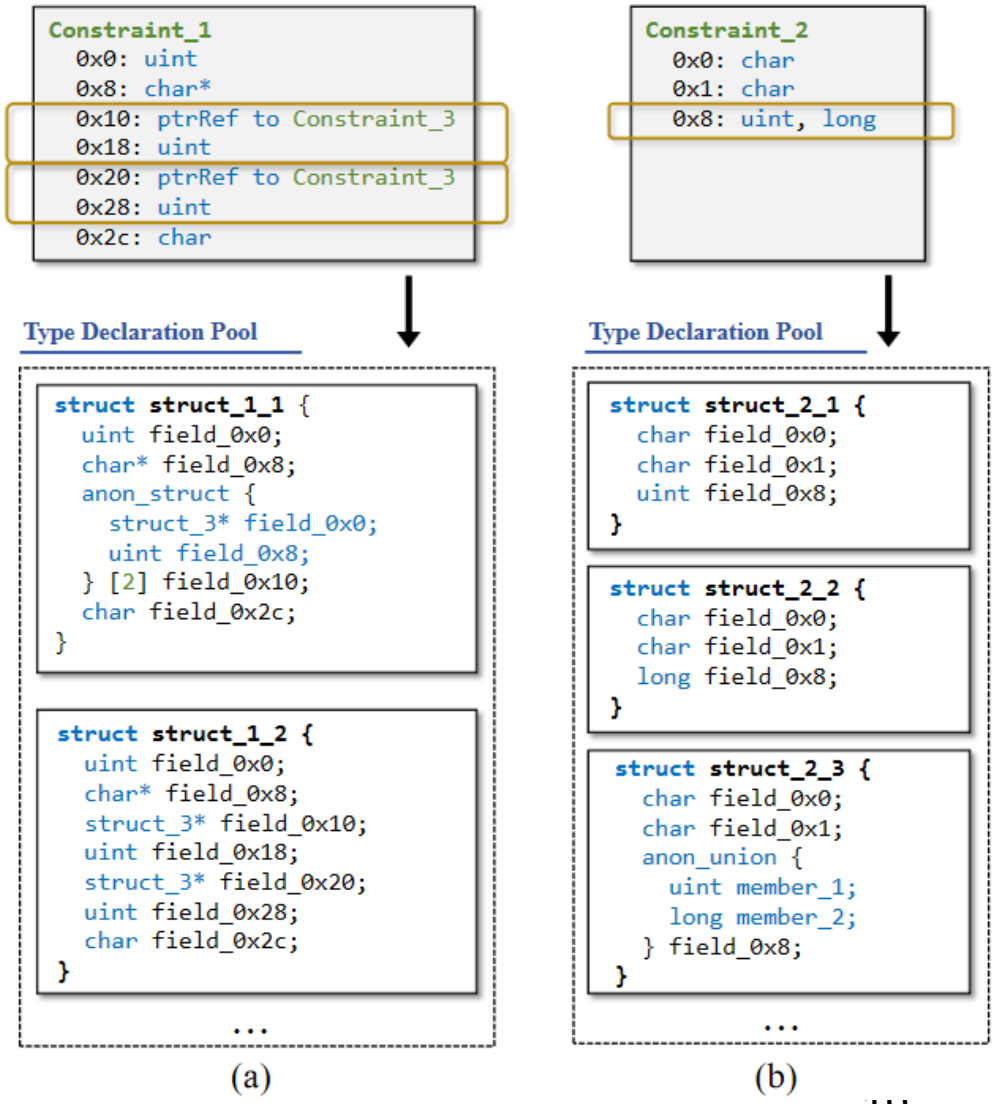
(a)




(b)

# TypeForge 基于TFG的类型合成

- 类型声明合成
  - 应对问题：现有方法通常仅根据同一约束生成单一复合数据类型声明，但同一约束可能存在歧义
    - 例如*(base + offset)既可能表示结构体成员访问，也可能表示数组元素访问
  - 设计一种自适应滑动窗口算法
    - 在每次扫描中，算法会在当前窗口内识别约束中的特征模式，并根据识别结果自适应地调整窗口大小，据此合成一系列候选类型声明，重复迭代，直至覆盖完整约束
      - 重复成员约束（右图a）
      - 冲突成员约束（右图b）



# TypeForge 基于可读性的选择

- 应对问题：现有方法难以有效处理约束歧义，即同一约束可能对应多种复合类型声明
- 挑战：在候选中选出最契合的复合类型声明需高度依赖专家经验，过程耗时且难以自动化
- 解决方案
  - 动机：类型恢复越准确，反编译代码在语法与语义上越精确、可读性越高
  - 利用LLM进行可读性驱动的选择，而无需额外设计复杂算法

(a)

```
void FUN_00133d4b
    (undefined8 param_1)
{
  struct_1 local_78;
  char* local_68;
  size_t local_60;
  undefined* local_58;
  undefined8 local_50;
  undefined* local_48;
  undefined8 local_40;
  undefined* local_38;
  undefined8 local_30;
  undefined local_28 [32];

  local_78 = &DAT_00162a53;
  local_70 = 1;
  local_68 = param_1;
  local_60.field_0x0 =
           strlen(param_2);
  local_60.field_0x8 =
           &DAT_00162a57;
  // ...
  return;
}
```

(b)

```
void FUN_00133d4b
    (undefined8 param_1)
{
  struct_2 local_78 [5];
  undefined local_28 [32];

  local_78[0].field_0x0 = "(";
  local_78[0].field_0x8 = 1;
  local_78[1].field_0x0 = param_1;
  local_78[1].field_0x8 =
              strlen(param_2);
  local_78[2].field_0x0 = ".";
  // ...
  return;
}
```

(i)

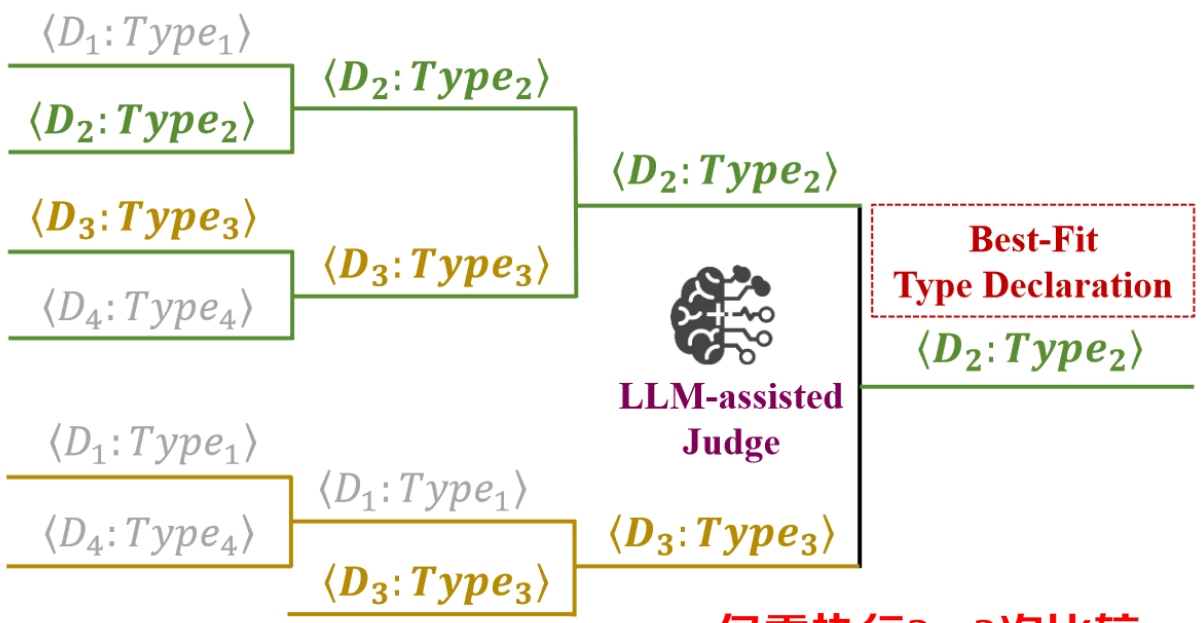
```
struct struct_1 {
    void* field_0x0;
    undefined8 field_0x8;
}
```

(ii)

```
struct struct_2 {
    char* field_0x0;
    size_t field_0x8;
}
```

# TypeForge 基于可读性的选择

- 具体步骤
  - 借助反编译器的retype接口，为每个候选类型声明依次生成对应的反编译代码版本
  - 由LLM作为“裁判”，对这些代码版本进行双淘汰比较，并基于语法清晰度（Tree-sitter）和可读性进行评估排序
  - 最终选取能生成最佳反编译代码版本的类型声明作为结果




仅需执行2n-2次比较

**Prompt for Readability Assessment**

Please assess the readability of each pair of the following decompiled code snippets, considering both the semantic content of the code and syntactic clarity metrics (where higher values indicate greater syntactic clarity).
Code snippet pairs and syntactic clarity metrics:
$\langle d_{f_1},\ metric_{f_1}\rangle\ \langle d'_{f_1},\ metric'_{f_1}\rangle$
...
$\langle d_{f_m},\ metric_{f_m}\rangle\ \langle d'_{f_m},\ metric'_{f_m}\rangle$

# TypeForge 实验设计

- 数据集：OSPREY Dataset
  - 包含函数数量：10000
  - 包含复合数据类型的变量数量：11000
  - 包含的复合数据类型：结构体及其指针、联合体及其指针、数组
- 默认LLM：GPT-4o mini
- 对比方法：DIRTY、OSPREY、ReSym、TYGR
- 评估任务
  - 复合数据类型识别（Composite Data Type Identification） 相关变量
    - 评估正确识别目标变量所属复合数据类型的准确性
    - 评价指标
      - Recall：正例样本中有多少被预测正确（漏报少则高）
      - Precision：预测为正的样本中有多少是真正的正样本（误报少则高）
      - F1：召回率与精确率的调和平均值

- 评估任务
  - 布局恢复（Layout Recovery） 布局
    - 对已识别出的复合数据类型，进一步评估其恢复的内部成员布局（偏移量Offset与大小Size）的准确性
    - 评价指标：Recall、Precision、F1
  - 关系恢复（Relationship Recovery）
    - 对已恢复的复合数据类型，进一步评估对它们之间关系识别的准确性
    - 评价指标：Recall、Precision、F1

关系：表示为五元组

⟨源类型布局，目标类型在源类型中的偏移量

目标类型布局，关系类型，指针引用的层级⟩

关系类型包括结构体指针引用、结构体嵌套和联合体

# TypeForge 对比实验

- 评估TypeForge与对比方法在三个评估任务上的表现
  - 复合数据类型识别：TypeForge显著优于现有最佳方法TYGR
    - OSPREY无法恢复基于寄存器的变量，包括函数参数和部分局部变量
  - 布局恢复：TypeForge在精确率和F1上优于现有最佳方法TYGR，在召回率上略低
    - 在识别evil edges时产生误报，导致有效边被误删，使部分复合类型成员丢失
    - TYGR缺乏过程间分析，无法处理占大多数的结构体指针的布局
  - 关系恢复：TypeForge成功恢复了复合数据类型之间的关系
    - DIRTY并不生成新的复合数据类型，而是从训练数据中匹配已有的完整声明

| | Composite Data Type Identification | | | Layout Recovery | | | Relationship Recovery | | | Analysis Time |
|---|---|---|---|---|---|---|---|---|---|---|
| Methods | Recall | Precision | F1 | Recall | Precision | F1 | Recall | Precision | F1 | (s) |
| DIRTY | 36.8 | 62.0 | 46.2 | 4.1 | 51.8 | 7.6 | 0.8 | 0 | 0 | 10.3 |
| OSPREY | 18.5 | 65.5 | 28.9 | 87.3 | 86.5 | 86.9 | - | - | - | 528.2† |
| TYGR | 86.4 | 34.5 | 49.3 | 5.3 | 31.3 | 9.1 | - | - | - | 323 |
| ReSym | - | - | - | 35.2 | 81.1 | 49.0 | - | - | - | - |
| TypeForge | **87.9** | **76.3** | **81.7** | 80.9 | **96.9** | **88.2** | **66.6** | **78.5** | **72.1** | **7.7** |

# TypeForge 消融实验

- 评估冲突感知的类型提示传播与基于可读性的选择与对TypeForge性能的影响
  - TypeForge evil
    - 将类型提示传播到所有具有数据流边的节点（不进行冲突感知）
    - 布局恢复任务中的P与F1显著下降，但R有上升
    - 识别evil edges时产生误报，导致有效边被误删，使部分复合类型成员丢失
  - TypeForge fast
    - 仅根据语法清晰度进行最终选择（不结合LLM）
    - 在两个任务中的表现均下降

| | Composite Data Type Identification | | | Layout Recovery | | |
|---|---|---|---|---|---|---|
| | R | P | F1 | R | P | F1 |
| TypeForge | **82.7** | **95.6** | **88.7** | **63.2** | **97.8** | **76.9** |
| $\text{TypeForge}_{\text{fast}}$ | 66.2 | **69.3** | **67.7** | 55.8 | 73.4 | 63.4 |
| $\text{TypeForge}_{\text{evil}}$ | 79.3 | 87.7 | 83.2 | 87.6 | **15.5** | **26.9** |

# TypeForge 其他实验

- 评估TypeForge在不同优化级别上的性能差异
  - 复合数据类型识别：TypeForge性能较为稳定
    - 在O2优化下，许多非复合局部变量被编译器合并或移除，显著减少干扰
  - 布局恢复：TypeForge在精确率上保持稳定，在召回率和F1上有较大差异
    - 激进的编译器往往导致复合类型成员被合并或扁平化，为布局重建带来挑战

| Task | Metric | O0 | O1 | O2 | O3 |
|---|---|---|---|---|---|
| Composite Data Type Identification | Precision | 74.3 | **76.1** | 74.9 | 73.5 |
| | Recall | 77.6 | 77.5 | **82.9** | 81.8 |
| | F1 Score | 75.9 | 76.9 | **78.8** | 77.4 |
| Layout Recovery | Precision | **97.9** | 97.3 | 93.9 | 94.2 |
| | Recall | **82.2** | 46.3 | 31.2 | 30.8 |
| | F1 Score | **89.4** | 62.7 | 46.8 | 46.4 |

# 特点总结与未来展望

# 算法总结 TypeForge

- 算法贡献
  - 针对现有方法难以高效且精确地构建约束，难以平衡准确率与效率的问题
    - 提出基于TFG的数据流抽象与冲突感知的类型提示传播，高效汇聚并净化类型提示，形成准确约束；设计自适应滑动窗口算法，高效生成候选
  - 针对现有方法难以有效处理约束歧义的问题
    - 设计可读性驱动的LLM辅助选择，筛选最终声明
  - 算法不足
    - 在识别evil edges时会产生误报，导致有效数据流边被误删，部分复合类型成员丢失
    - 时间开销主要集中在可读性驱动选择阶段
      - 大语言模型在处理长提示时生成输出较慢
- 未来展望
  - 探索更智能的冲突检测机制，减少误报
  - 探索更轻量化的可读性评估机制，以提升整体效率

# 参考文献

[1] Wang Y, Liang R, Li Y, et al. TypeForge: Synthesizing and Selecting Best-Fit Composite Data Types for Stripped Binaries[C]. 2025 IEEE Symposium on Security and Privacy. San Francisco, CA: IEEE, 2025: 1-18.

# 道德经

知人者智，自知者明。胜人者有力，自胜者强。知足者富。强行者有志。不失其所者久。死而不亡者，寿。

谢谢！