2025 IEEE Symposium on Security and Privacy (SP) | 979-8-3315-2236-0/25/$31.00 ©2025 IEEE | DOI: 10.1109/SP61157.2025.00193

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

Yanzhong Wang[1,2], Ruigang Liang[1,2,*], Yilin Li[1,2], Peiwei Hu[1,2], Kai Chen[1,2,*], and Bolun Zhang[1,2]
[1]*State Key Laboratory of Cyberspace Security Defense, Institute of Information Engineering, CAS, China*
[2]*School of Cyber Security, University of Chinese Academy of Sciences, China*
{*wangyanzhong, liangruigang, liyilin2023, hupeiwei, chenkai, zhangbolun*}*@iie.ac.cn*

***Abstract*—Static binary analysis is a widely used approach for ensuring the security of closed-source software. However, the absence of type information in stripped binaries, particularly for composite data types, poses significant challenges for both static analyzers and reverse engineering experts in achieving efficient and accurate analysis. Existing methods often struggle with inaccuracies and scalability limitations when dealing with such data types. To address these problems, we present TypeForge, a novel approach inspired by the workflow of reverse engineering experts, which uses a two-stage synthesis-selection strategy to automate the recovery of composite data types from stripped binaries. We design a new graph structure, the Type Flow Graph (TFG) to represent type information within stripped binaries. In the first stage, TFG-based Type Synthesis focuses on efficiently and accurately building constraints and synthesizing possible composite type declarations from the stripped binaries. In the second stage, we propose an LLM-assisted double-elimination framework to select the best-fit type declaration from the candidates by assessing the readability of the decompiled code. Our comparison with state-of-the-art approaches demonstrates that TypeForge achieves F1 scores of 81.7% and 88.2% in Composite Data Type Identification and Layout Recovery, respectively, substantially outperforming existing methods. Additionally, TypeForge achieves an F1 score of 72.1% in Relationship Recovery, a particularly challenging task for previous approaches. Furthermore, TypeForge has significantly lower time overhead, requiring only about 3.8% of the time taken by OSPREY, the best-performing existing approach, making it a promising solution for various real-world reverse engineering tasks.**

1. Introduction

Conducting security analysis on closed-source software is widely recognized as a challenging task. One key reason is that stripped binaries in closed-source software typically lack variable and type information, which is lost during compilation. Accurately recovering this missing type information is critical for various downstream tasks, including vulnerability detection [1], [2], [3], [4], [5], decompilation [6], [7], [8], [9], and malware analysis [10], [11], [12]. For instance, static analysis tools can more efficiently identify vulnerabilities by locating read and write accesses to specific data structure members, while decompilers can use recovered type information to enhance code readability. Currently, widely used binary analysis platforms like Ghidra [13] and IDA Pro [14] can recover primitive data types, such as `int` and `long`, with reasonable accuracy, but they encounter significant difficulties in recovering composite data types. These mainstream platforms still rely heavily on manual efforts from users to recover composite data types, resulting in considerable time consumption and a strong dependence on expert experience. Our statistics show that accesses to composite data type variables and their fields in binaries are nearly 1.8 times more frequent than accesses to primitive variables, and this ratio increases further in more complex software. Additionally, our statistics indicate that the state-of-the-art binary analysis platform, IDA Pro [14], can accurately recover only 6% of composite data types, only limited to library-defined types, while user-defined composite data types remain unrecoverable.

Several recent studies have attempted to address the challenge of recovering composite data types in stripped binaries. TIE [15] and OSPREY [16] utilize whole-program memory dependency analysis, such as *Value-Set Analysis* (VSA) [17], along with a series of heuristic rules to infer composite data types from stripped binaries. DIRTY [9] leverages a transformer model to predict known types from a training set, but it has limitations in recovering user-defined composite data types because DIRTY ignores data flow relationships between variables, resulting in inconsistent type predictions for variables that should share the same type. TYGR [18] combines intra-procedural analysis with graph neural networks to predict variable types within functions, but it fails to recover the member layout of composite types. ReSym [19] leverages large language models and inter-procedural analysis, and while it can recover composite type layouts to some extent, its accuracy remains relatively low for practical applications.

In summary, existing efforts suffer from the following limitations. **L1:** *Difficulty in accurately and efficiently recovering the full declaration of composite data types.* Existing methods either prioritize efficiency, resulting in numerous false positives that lead to both missing legitimate members and introducing spurious members within composite data

*Corresponding Author.

DOI 10.1109/SP61157.2025.00193

types [9], [19], or trade time for accuracy, making them difficult to scale to more complex and larger binary programs [15], [16]. Furthermore, existing methods often struggle to reconstruct relationships between composite types, such as structure pointer references and structure nesting. These relationships are crucial for downstream security analysis tasks, for example, a composite type containing two pointer-type members that reference itself would signal to reverse engineers that it likely represents a node in a doubly linked list or tree structure. **L2:** *Difficulty in effectively handling ambiguities while recovering composite data types.* Most efforts [16], [17], [19] rely on memory access patterns collected from stripped binaries to recover composite data types. However, different composite data types often exhibit the same memory access patterns in stripped binaries. For example, access pattern `dereference(base + offset)` could originate from either structure member accesses or array element accesses in the source code, making it difficult to accurately determine their actual types.

By observing how human reverse engineering experts manually recover composite data types, we find that they typically begin by synthesizing an initial composite data type declaration based on partial type hints (such as memory access patterns) in binary. They then continuously analyze the binary, iteratively refining and modifying this initial declaration. When facing uncertainty, they consider multiple potential alternatives as candidates and ultimately use their expertise to select the best-fit composite type declarations. Inspired by the above observation, we realize that replicating the workflow of reverse engineering experts, which involves (1) continuously analyzing the binary and synthesizing potential composite data type declarations as candidates and (2) selecting the best-fit type declaration from these candidates, can effectively overcome the limitations of accuracy and ambiguity in existing methods. However, implementing such an automated technique presents some challenges.

Challenges. C1: *Existing techniques struggle to construct constraints for composite data types efficiently and accurately, resulting in an extensive search space when synthesizing declarations.* Due to the inherent complexity of internal member layouts and relationships in composite data types, accurate constraints are always necessary when synthesizing possible type declarations; without them, an enormous search space results. However, as previously mentioned, existing data flow analysis techniques used to construct composite type constraints for stripped binaries cannot effectively balance accuracy and scalability, limiting their practical use in real-world reverse engineering scenarios. **C2:** *Difficulty in selecting the best-fit composite data type declaration from candidates.* Reverse engineering experts rely heavily on domain-specific experience and a deep understanding of the target binary to select the final type declaration from the candidates, often consuming significant time and effort. However, translating this expert experience into an automated process is challenging, as the complexity and diversity of stripped binaries make it difficult to quantify this expertise into corresponding algorithms.

Our Approach. In this paper, we introduce TypeForge, a novel automated approach that employs a two-stage synthesis-selection strategy to emulate the workflow of a reverse engineering expert and overcome the aforementioned challenges. **To address C1**, we propose *TFG-based Type Synthesis* for accurately and efficiently synthesizing potential composite data type declarations from stripped binaries. Specifically, we first design a novel data flow abstraction called the *Type Flow Graph* (TFG) based on variables and primitive types inferred by existing tools. Compared to previous work, TFG not only efficiently gathers type hints representing both the internal layout and the relationships between composite data types, but also supports constructing whole-program TFG through parallelized analysis, minimizing false positives while maintaining low time overhead. Based on TFG, we propose a *Conflict-Aware Type Hint Propagation* algorithm to process the whole-program TFG for constructing accurate composite type constraints. Unlike the inter-procedural analysis used in existing methods [16], [19], *Conflict-Aware Type Hint Propagation* effectively prevents erroneous propagation of type hints through conflict detection, significantly reducing false positives while maintaining low time overhead. Finally, informed by the factors contributing to ambiguity in composite data type recovery, we design an *Adaptive Sliding Window* algorithm that synthesizes all possible declarations based on the constructed type constraints.

To address C2, we observe that the higher the accuracy of the composite data types recovered by the expert, the more precise the syntax and semantics of the decompiled code, significantly improving its readability. Therefore, rather than designing a specific algorithm, we propose *Readability-Guided Selection*, an LLM-assisted double-elimination [20] mechanism to select the best-fit composite data type declaration from the candidates synthesized in stage one. Specifically, we first provide the candidate type declarations to the decompiler via the retype interface[1], generating different variants of decompiled code. Next, these different variants of decompiled code are subjected to multiple rounds of pairwise comparison, with the LLM-assisted mechanism serving as the judge, assessing their readability and ranking them accordingly. Finally, this mechanism selects the composite data type declaration that produces the most readable decompiled code variant.

We evaluate TypeForge by comparing it with state-of-the-art approaches, and our results demonstrate that TypeForge achieves high F1 Scores of 81.7% and 88.2% in *Composite Data Type Identification* and *Layout Recovery*, respectively, superior to these existing approaches. Notably, TypeForge can recover relationships between composite data types with an F1 Score of 72.1%, which most existing approaches cannot handle. Additionally, TypeForge achieves a significantly lower time overhead, requiring only about 3.8% of the time taken by the best-performing existing approach, OSPREY, demonstrating that it can scale to larger

[1] Mainstream decompilers (e.g., IDA Pro, Ghidra, Binary Ninja) allow users to manually create and specify variable types through the retype interface, helping to produce clearer and more accurate decompiled code.

binaries with acceptable time consumption.

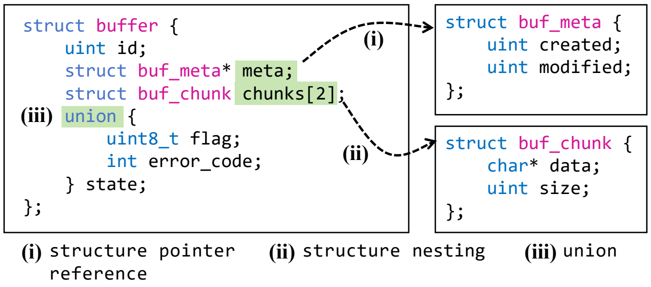
Contribution. In summary, our work makes the following contributions:

- We propose a two-stage synthesis-selection approach to recover composite data types from stripped binaries, emulating the workflow of reverse engineering experts and addressing challenges of accuracy and ambiguity.
- We propose *TFG-based Type Synthesis*, an accurate and efficient technique for synthesizing possible composite data type declarations from stripped binaries. Additionally, we introduce an LLM-assisted double-elimination mechanism to assess and rank the readability of decompiled code.
- We implement TypeForge with over 12,000 lines of code, and our evaluation on a real-world dataset demonstrates that TypeForge outperforms state-of-the-art approaches in both accuracy and efficiency. We release our research artifacts[2] to facilitate future comparisons and advancements in this domain.

2. Background and Motivation

2.1. Background

Binary Type Inference Binary type inference is the process of reconstructing source-level type information from memory locations and registers in stripped binaries, which is essential for downstream security analysis. Numerous existing tools and research efforts focus on recovering primitive types, employing either intra-procedural value-set analysis [13], [14], [15] or AI-based methods [9], [18] with considerable success. However, recovering composite data types remains a significant challenge. Composite data types include structures, unions, and pointers to these types. In real-world scenarios, these composite data types are often combined through various relationships, such as pointer references to other structures, nested structures, and arrays of structures. Figure 1 shows an example of composite data types from a real-world project, with a simplified declaration for clarity. The structure `buffer` contains four members: `id`, `meta`, `chunks`, and `state`. Specifically, `id` is a member with the primitive type `uint`; `meta` is a pointer to the `buf_meta` structure; `chunks` is an array of nested `buf_chunk` structure; `state` is a union containing a `short` type member and an `int` type member. Consequently, recovering composite data types requires not only reconstructing the layout of structure members but also identifying the pointer references **(i)** and nested relationships **(ii)** between structures, as well as the unions **(iii)** composed of different types. Unlike primitive types, recovering composite data types typically requires inter-procedural analysis to collect type hints from the whole program, which presents greater challenges. Although existing studies [16], [19] attempt to recover composite data types, they are hindered by their data-flow abstraction and inter-procedural analysis techniques, resulting in their ability only to recover partial information about structure member offsets and sizes, and struggle to balance efficiency with accuracy, limiting their practical application in real-world scenarios.



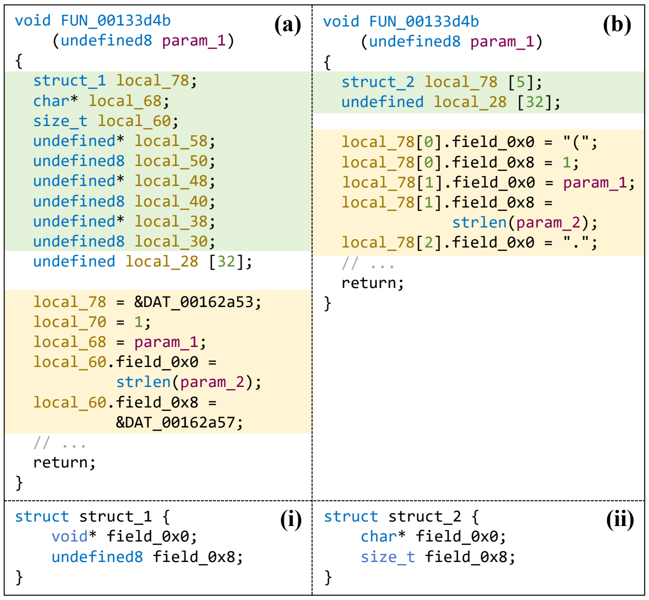
```
struct buffer {
    uint id;
    struct buf_meta* meta;
    struct buf_chunk chunks[2];
    union {
        uint8_t flag;
        int error_code;
    } state;
};
```

```
struct buf_meta {
    uint created;
    uint modified;
};
```

```
struct buf_chunk {
    char* data;
    uint size;
};
```

(i) structure pointer reference **(ii)** structure nesting **(iii)** union

Figure 1: Declaration of a user-defined structure.



```
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;
}
```

```
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;
}
```

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

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

Figure 2: Motivating example that inspired the design of TypeForge. **(a)** shows the decompiled code after retyping `local_78` as `struct_1` **(i)**, while **(b)** shows it retyped as an array of `struct_2` **(ii)** in the same function.

Readability assessment of decompiled code. Due to the loss of source-level symbols during compilation, there is no standard reference for assessing the readability of decompiled code. Existing code readability metrics focus on specific syntactic characteristics. For example, the McCabe Cyclomatic Complexity [21] used by revng-c [22] mainly evaluates control flow complexity. Although some studies [23] incorporate various syntactic features into readability metrics, they still fail to account for code semantics. For instance, an expression like `dereference(base + offset)` within a loop often indicates array access rather than structure member access. Thus, decompiled code is more readable when this variable is retyped as an array rather than a structure. Readability assessment methods that do not capture such semantic differences are neither comprehensive nor objective.

[2] Available at https://github.com/noobone123/TypeForge

2.2. Motivating Example

Figure 2 is a motivating example that inspired the design of TYPEFORGE. In Figure 2, **(a)** and **(b)** are different variants of the decompiled code for the function `FUN_00133d4b`, **(i)** and **(ii)** represent potential composite data type declarations for `local_78` as considered by experts during the reverse engineering process. The only difference is that in **(a)**, the variable `local_78` is retyped as `struct_1` **(i)**, while in **(b)**, it is retyped as an array of `struct_2` **(ii)**. To select the best-fit type between them, we assess the readability of the decompiled code generated after retyping with each declaration. It is evident that **(b)** has better readability compared to **(a)**, **(b)** significantly reduces the number of variables (highlighted in green) and organizes the code more logically by structuring related data into an array of structures (highlighted in blue), which more intuitively shows the sequence and manipulation of related data.

3. Approach

3.1. Overview

Figure 3 illustrates the workflow of TYPEFORGE, where the input is a stripped binary, the output is recovered composite data type declarations, including the layout, relationships, and related variables. TYPEFORGE works in two stages: *Stage1, TFG-based Type Synthesis* (Section 3.2), which constructs type constraints through TFG and synthesizes possible composite data type declarations as candidates, while *Stage2, Readability-Guided Selection* (Section 3.3.2), selects the best-fit declaration from the candidates.

In *Stage 1*, TYPEFORGE leverages variables and primitive types inferred by the decompiler to construct a whole-program *Type Flow Graph (TFG)*, representing potential type relationships throughout the entire program (Section 3.2.1). Subsequently, TYPEFORGE employs *Conflict-Aware Type Hint Propagation* on the whole-program TFG to efficiently construct accurate constraints for composite data types (Section 3.2.2). Finally, TYPEFORGE uses *Adaptive Sliding Window* algorithm to synthesize possible composite data type declarations based on these constraints, effectively addressing inherent ambiguity (Section 3.2.3).

In *Stage 2*, TYPEFORGE submits type declarations to the decompiler's retype interface to obtain variants of the decompiled code and constructs a mapping from each decompiled code variant to its type declaration (Section 3.3.1). Then, TYPEFORGE uses an LLM-assisted double-elimination method to select the variant with the highest readability, recognizing the corresponding type declaration as the final best-fit composite data type declaration (Section 3.3.2).

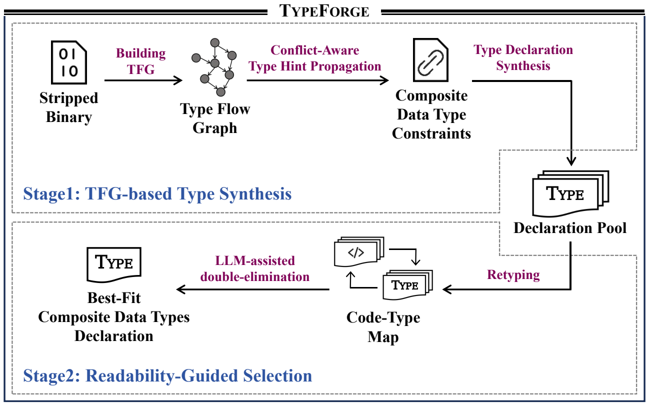

TYPEFORGE
01
10
Stripped Binary
Building TFG
Type Flow Graph
Conflict-Aware Type Hint Propagation
Composite Data Type Constraints
Type Declaration Synthesis
Stage1: TFG-based Type Synthesis
TYPE
Declaration Pool
Retyping
</>
TYPE
Code-Type Map
LLM-assisted double-elimination
TYPE
Best-Fit Composite Data Types Declaration
Stage2: Readability-Guided Selection

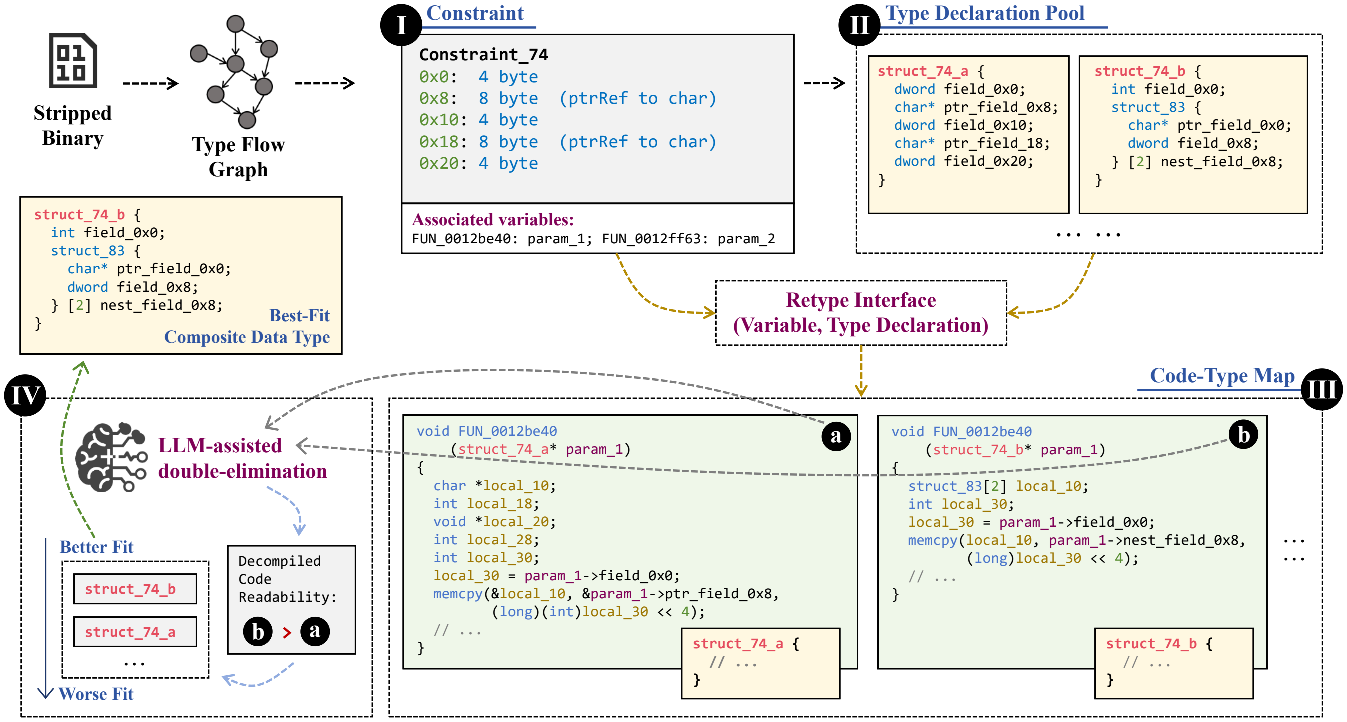
Figure 3: Workflow of TYPEFORGE.

Example. Figure 4 shows a concrete example of how TYPEFORGE recovers a structure from an open-source project. TYPEFORGE takes the stripped binary as input and first builds a whole-program TFG. Then, *Conflict-Aware Type Hint Propagation* is applied to the TFG to construct type constraints, as shown in `Constraint_74` **(I)**. `Constraint_74` includes the offsets and sizes of structure members, pointer relationships between types, and associated variables (e.g., `param_1` in `FUN_0012be40` and `param_2` in `FUN_0012ff63`, both inferred by the decompiler). Next, TYPEFORGE applies *Adaptive Sliding Window* algorithm to synthesize possible composite data type declarations based on `Constraint_74`, which are then added to the Type Declaration Pool **(II)**. `struct_74_a` and `struct_74_b` are examples of synthesized structure declarations, where `struct_74_b` merges certain members into a nested structure array, while `struct_74_a` does not. Subsequently, TYPEFORGE uses the decompiler's retype interface to collect different decompiled code variants, forming a Code-Type Map **(III)**. In this map, code snippets **(a)** and **(b)** correspond to the structure `struct_74_a` and `struct_74_b` applied to the variable `param_1` in function `FUN_0012be40`, respectively. Finally, TYPEFORGE employs an LLM-assisted double-elimination mechanism to judge the readability of decompiled code pairs within the Code-Type Map, as illustrated in **(IV)**. In the figure, the mechanism determines that decompiled code **(b)** is more readable than decompiled code **(a)**, making `struct_74_b` the preferred structure declaration. Through this process, TYPEFORGE selects the best-fit composite data type declaration. Below, we present the details of both stages.

3.2. TFG-based Type Synthesis

3.2.1. Building TFG. Existing work typically relies on heavy inter-procedural value-set analysis [17] or binary dependency analysis [24] to collect composite type hints from the whole program. However, these efforts are usually flow-sensitive and context-sensitive, often resulting in significant time overhead. Furthermore, the data flow abstractions these approaches employ frequently generate numerous false positives in complex inter-procedural analysis. To address these limitations, we introduce a novel data flow abstraction, the *Type Flow Graph (TFG)*, to efficiently and accurately collect type hints for composite data types. The design of TFG is based on the following three key observations.

Observations. O1: Member accesses of composite types in decompiled code are typically represented as expressions like $var_{left} = dereference(var_{right} + offset)$,



```
Constraint_74
0x0:  4 byte
0x8:  8 byte  (ptrRef to char)
0x10: 4 byte
0x18: 8 byte  (ptrRef to char)
0x20: 4 byte
```

Associated variables:

```
FUN_0012be40: param_1; FUN_0012ff63: param_2
```

```
struct_74_a {
  dword field_0x0;
  char* ptr_field_0x8;
  dword field_0x10;
  char* ptr_field_18;
  dword field_0x20;
}
```

```
struct_74_b {
  int field_0x0;
  struct_83 {
    char* ptr_field_0x0;
    dword field_0x8;
  } [2] nest_field_0x8;
}
```

```
struct_74_b {
  int field_0x0;
  struct_83 {
    char* ptr_field_0x0;
    dword field_0x8;
  } [2] nest_field_0x8;
}
```

```
struct_74_b
struct_74_a
...
```

```
Decompiled
Code
Readability:
```

```
void FUN_0012be40
    (struct_74_a* param_1)
{
  char *local_10;
  int local_18;
  void *local_20;
  int local_28;
  int local_30;
  local_30 = param_1->field_0x0;
  memcpy(&local_10, &param_1->ptr_field_0x8,
         (long)(int)local_30 << 4);
  // ...
}
```

```
struct_74_a {
  // ...
}
```

```
void FUN_0012be40
    (struct_74_b* param_1)
{
  struct_83[2] local_10;
  int local_30;
  local_30 = param_1->field_0x0;
  memcpy(local_10, param_1->nest_field_0x8,
         (long)local_30 << 4);
  // ...
}
```

```
struct_74_b {
  // ...
}
```

Figure 4: An example of TypeForge demonstrates the complete process of recovering a complex structure from a stripped binary.

```
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;
  // ...
}
```

Figure 5: Simplified code of functions from a real-world project decompiled by Ghidra.

where var_{left} and var_{right} are variables inferred by the decompiler, as shown in Figure 5. In such expressions, var_{right} serves as a base pointer to a composite type, $offset$ indicates the position of a specific member within that type, and the size of this member corresponds to the size of var_{left}. These expressions provide essential type hints for recovering the layout of composite data types. **O2:** More critical type hints for recovering relationships can be collected by further analyzing and combining these expressions. For example, line 10 of function `Fun_02d3` in Figure 5 reveals that `local_10` points to a composite type that contains a member at offset `0x30` with a size of `0x8`, matching the type of `local_18`. Line 11 further reveals that `local_18` also points to a composite type with a member at offset `0x8`. The expressions from Lines 10 and 11 can be combined to form `*(*(local_10 + 0x30) + 0x8)`, revealing that `local_10`'s composite type contains a pointer-type member at offset `0x30` which references another composite data type with a member at offset `0x8` through a single-level pointer. Similarly, expressions like $(var+offset)$ appearing as arguments at callsites typically indicate that the composite type corresponding to var contains a nested structure at that offset. These expressions can also be used to construct type alias information without performing costly and inaccurate value-set analysis. Notably, many of these revealing expressions do not appear explicitly in the program statements. Therefore, we must systematically uncover these implicit expressions to collect sufficient type hints for constructing type constraints. **O3:** Most variables with direct data flow relationships inferred by the decompiler typically share the same data type, regardless of differences in function call contexts, execution paths, or program points. Although typecasting and unions in programs can violate this observation, their occurrence is relatively rare. Therefore, by effectively identifying these exceptional cases (Section 3.2.2), we can avoid costly flow-sensitive and context-sensitive analyses, significantly reducing analysis time overhead.

Based on these observations, we designed TFG and associated algorithms to significantly enhance the efficiency and accuracy of composite type synthesis. Next, we provide the complete definition of TFG, followed by algorithms for building the whole-program TFG.

Definition 1. *Let TFG be a directed graph represented as a tuple* $G = (N, E, L)$*, where:*

- N *is a set of Nodes. Each* $n \in N$ *corresponds to a Nested*

Member Access Expression (NMAE, formally defined later) generated during intra-procedural analysis.

- *$E \subseteq N \times N \times T$ is a set of directed edges, where $T = \{dataflow, member, typealias\}$ denotes the edge types. For any edge $(n_i, n_j, t) \in E$, the type t indicates a specific relationship between node n_i and node n_j. Specifically, a $dataflow$ edge represents a direct data flow relationship, such as an assignment or parameter passing in function calls; a $member$ edge indicates a membership where the source node represents a composite type and the target node represents a member of that composite type; and a $typealias$ edge indicates that two nodes have identical types, even though there is no explicit data flow relationship between them.*
- *$L : E \rightarrow \mathbb{Z}_{\geq 0}$ is a labeling function, defined exclusively on $member$ edges. For a $member$ edge $(n_i, n_j, member)$, the label $L(n_i, n_j)$ provides a non-negative integer offset, specifying the position of the member represented by the target node within the composite data type represented by the source node.*

Table 1 presents the recursive definition of NMAE, which represents each node within the TFG. In Table 1, $expr$ represents an NMAE, which can be a variable, a binary expression combining two $expr$ components, or an expression formed by applying an unop operation to another $expr$. Specifically, **binop** includes binary operators that may involve pointer arithmetic, while **unop** encompasses pointer dereference and reference operations. NMAEs that include **unop** represent data stored at or loaded from the memory locations denoted by its inner NMAE. Here, var denotes a constant or a variable inferred by the decompiler, including local variables on the stack, global variables, heap variables, and function parameters stored in registers.

TABLE 1: Recursive Definition of NMAE

$expr$	$::=$	$expr$ **binop** $expr$ $\mid$ **unop** $expr$ $\mid$ var
var	$::=$	$DecompiledVar \mid Constant$
$DecompiledVar$	$::=$	$v_{stack} \mid v_{global} \mid v_{heap} \mid v_{param}$
binop	$::=$	$+,\ -,\ \times,\ \ldots$
unop	$::=$	$*,\ \&$

It is worth noting that mainstream decompilers [13], [14], [25] can recover primitive types from source code with extremely high accuracy, while composite types and their pointers are frequently misidentified. TFG addresses this limitation by leveraging decompiler-inferred primitive types and related expressions to recover composite data types, thereby ensuring the reliability of the results. Another key advantage of TFG is that it does not require replacing the decompiler's underlying algorithms with other complex dynamic or static analyses, making it extensible to any decompiler capable of primitive type inference, significantly enhancing its practical applicability.

Intra-procedural analysis. Intra-procedural analysis aims to uncover and collect as many explicit and implicit type hints as possible within a single function by analyzing decompiled code. These type hints typically include the variables associated with composite data types, the offsets and sizes of members within composite data types, and other types that may be referenced by pointers or nested within the structure. Figure 6 shows a TFG constructed from the two functions shown in Figure 5. Nodes marked in purple and blue represent the results of the intra-procedural analysis on functions `Fun_0413` and `Fun_02d3`, respectively (temporarily ignoring the edges labeled as "Inter"). Taking the TFG of `Fun_02d3` as an example, based on the statement `local_10 = param_1`, our analysis can construct a $dataflow$ edge from node `param_1` to node `local_10`. Furthermore, the expression `*(local_10 + 0x30)` on line 10, matches the access pattern for structure members; therefore, there is a $member$ edge labeled `0x30` from `local_10` to `*(local_10 + 0x30)`. Since the value of this member is assigned to `local_18`, there is a $dataflow$ edge from `*(local_10 + 0x30)` to `local_18`, implying that the member's type is consistent with that of `local_18`.

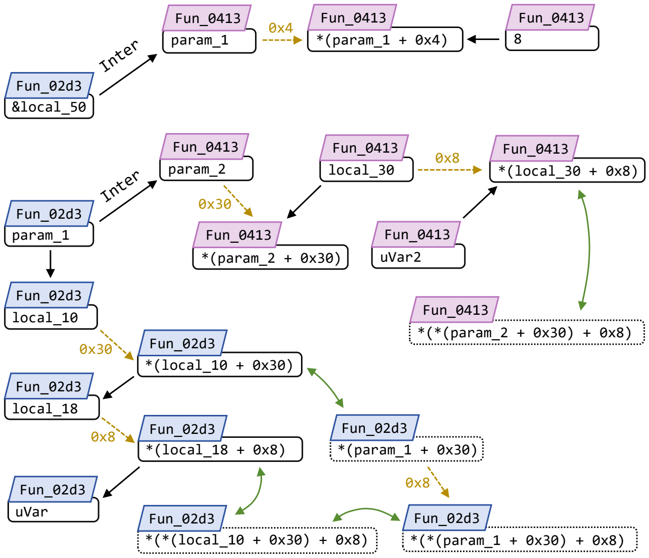


Figure 6: An example of TFG contains two functions, where black arrows represent direct data flow relationships, and yellow dashed arrows represent membership within composite data types, with the constants on the edges indicating the members' position. Green arrows indicate that the connected NMAE nodes share the same type due to aliasing.

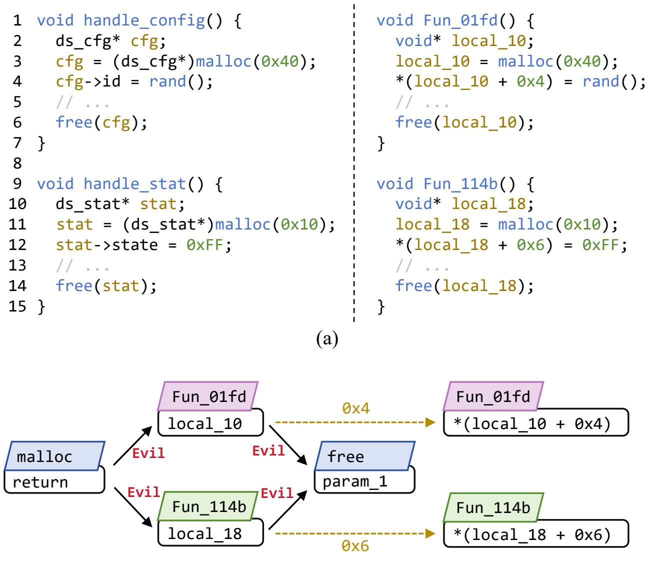
Besides collecting explicit expressions present in program statements, TypeForge can also automatically infer and create new implicit NMAE nodes in the TFG based on known expressions, as shown in Algorithm 1. Algorithm 1 maintains a mapping from variables to sets of NMAE nodes (line 3), which represents the collection of NMAE whose types match that of the variable. It then iteratively processes each statement in the decompiled code (line 4). For each statement, the algorithm generates new NMAEs based on the NMAEs associated with the input variable and the effect of the statement (line 8), updating both the expression set for the output variable and the TFG accordingly (lines 10-11). A noteworthy aspect

Algorithm 1: Intra-procedural analysis for building TFG

```
   Input: $D$: Decompiled code of function $f$
   Output: $G_f$: Type Flow Graph of function $f$
1  $\mathcal{S} \leftarrow$ ExtractStatements($D$)
2  $\mathcal{V} \leftarrow$ ExtractInferredVariables($D$)
3  $\mathcal{E} \leftarrow \{v \mapsto \{$NewExpr$(v)\} \mid v \in V\}$
4  foreach statement $s \in \mathcal{S}$ do
5      $v_{in} \leftarrow$ GetInputVars($s$)
6      $v_{out} \leftarrow$ GetOutputVar($s$)
7      foreach expression $e \in \mathcal{E}[v_{in}]$ do
8          $e' \leftarrow$ VisitStatement($e, s$)
9          if $e' \neq \bot$ then
10             $\mathcal{E}[v_{out}] \leftarrow \mathcal{E}[v_{out}] \cup \{e'\}$
11             $G_f \leftarrow$ UpdateTFG($G_f, e', e, s$)
12         end
13     end
14 end
15 return $G_f$
```

of the algorithm is that every NMAE associated with the input variable is used to generate new NMAEs (line 7), which enables the discovery of additional implicit type hints. Specifically, our algorithm searches for other known expressions that share the same type with the input NMAE's base and uses them to replace the base, thereby creating new NMAE nodes. For example, when processing the statement on line 10 of the function `Fun_02d3` in Figure 5, the input variable `local_10` is associated with a NMAE set containing both `local_10` and `param_1` due to the assignment at line 9. Consequently, an implicit NMAE `*(param_1 + 0x30)` is generated, indicating that the composite type corresponding to `param_1` has a member at offset `0x30` (these implicit nodes are represented with dashed boundaries in Figure 6). Furthermore, these generated NMAE can also be used to build alias relationships. For instance, since `param_1` and `local_10` have a dataflow relationship and share the same type, both `*(param_1 + 0x30)` and `*(local_10 + 0x30)` represent members of that type and are therefore aliases. It can be further inferred that `*(*(local_10 + 0x30) + 0x8)` and `*(*(param_1 + 0x30) + 0x8)` are also aliases.

The design of TFG enables TypeForge to avoid the need for costly context-sensitive and flow-sensitive analysis, allowing both decompilation and intra-procedural analysis to be efficiently executed in parallel. During intra-procedural analysis, TypeForge records dataflow facts of callsite arguments and function return values as connecting information. After all function-level TFGs are built, TypeForge connects these individual graphs by adding $dataflow$ edges based on this connecting information, ultimately forming the whole-program TFG. For example, in Figure 6, the TFG of `Fun_02d3` integrates with that of `Fun_0413` via $dataflow$ edges connecting the callsite arguments in the caller to the parameters in the callee, with "Inter" labels are used to mark these inter-procedural connections.



```
void handle_config() {
  ds_cfg* cfg;
  cfg = (ds_cfg*)malloc(0x40);
  cfg->id = rand();
  // ...
  free(cfg);
}

void handle_stat() {
  ds_stat* stat;
  stat = (ds_stat*)malloc(0x10);
  stat->state = 0xFF;
  // ...
  free(stat);
}
```

```
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)

Figure 7: A simple example illustrating the challenges in constraint construction. (a) shows a snippet of source code and the corresponding decompiled code from a real-world open-source project that includes typecasting. (b) shows a portion of the TFG for this project.

Leveraging the design of TFG and NMAE, TypeForge can efficiently collect both explicit and implicit type hints, supporting the construction of more sufficient and accurate constraints for composite data types.

3.2.2. Conflict-Aware Type Hint Propagation. To construct composite type constraints based on the whole-program TFG, a straightforward strategy is to assume that all nodes connected by $dataflow$ or $typealias$ edges in the TFG share the same type, then group these nodes together and merge their type hints, similar to the approach used by ReSym [19]. However, due to the existence of typecasting and unions in programs, and the fact that decompilers typically cannot recognize them, nodes with direct dataflow relationships in the TFG do not always mean they have consistent types. Although typecasting and unions are relatively rare in programs, if not identified, many spurious type hints will propagate incorrectly in the whole-program TFG, leading to highly inaccurate constraint construction.

Figure 7 illustrates this situation with a simple example. Figure 7(a) shows source code (left) and its corresponding decompiled code (right) from an open-source project. The functions `handle_config` and `handle_stat` both call `malloc` and `free` for memory allocation and deallocation, with the pointers returned by `malloc` being cast to the corresponding structure pointers `ds_cfg*` and `ds_stat*`. However, due to the absence of type information in stripped binaries, existing decompilers are unable to recover typecasting from the source code, as illustrated in the right side of Figure 7(a). As a result, the TFG constructed from this decompiled code adds $dataflow$ edges between the return value of `malloc` and both `Fun_01fd:local_10` and `Fun_114b:local_18`, as shown in Figure 7(b). The $dataflow$ edges (marked as "Evil") in Figure 7(b) suggest that `Fun_01fd:local_10` and `Fun_114b:local_18` share the same type. In reality, these two variables have different types, and these edges are introduced by typecasting in the source code. Without identifying and removing these "evil edges", the type hints for `local_10` (corresponding to `ds_cfg*`) and `local_18` (corresponding to `ds_stat*`) would be incorrectly conflated and propagated to other nodes, resulting in numerous spurious members appearing in the composite type constraints.

Similarly, variables involving unions can also introduce "evil edges" in TFG, causing type hints to propagate incorrectly throughout the whole program. Although these cases are relatively rare, even a single "evil edges" can cause significant inaccuracies in the resulting type constraints.

Observations. We make the following key observations to identify and remove "evil edges" in the whole-program TFG. **O1:** Different composite data types have distinct sizes, which can typically be collected at callsites of functions such as `malloc`, `calloc`, and `memset` (e.g., in Figure 7(a), `ds_cfg` has a size of `0x40`, while `ds_stat` has a size of 0x10). By collecting these critical constants and associating them as attributes of NMAE, we can propagate these size information throughout the whole-program TFG. Since "evil edges" often lead to conflicting size information at nodes (e.g., in Figure 7(b), `malloc:return` and `free:param_1` might simultaneously have sizes of both `0x40` and `0x10`), they can be effectively identified. **O2:** Different composite data types have distinct member layouts. For example, in Figure 7(b), `Fun_01fd:local_10` has a 4-byte integer member at offset `0x4`, while `Fun_114b:local_18` has a 1-byte member at offset `0x6`. When "evil edges" erroneously merge these type hints, the resulting member layout conflicts serve as a reliable signal for identifying such edges.

Algorithm. Based on the above observations, we design Conflict-Aware Type Hint Propagation to identify and remove "evil edges" in the whole-program TFG, and ultimately construct accurate constraints for composite data types. The Conflict-Aware Type Hint Propagation faces two key challenges during implementation. First, due to the prevalence of wrapper functions for memory management in programs, it is often impossible to directly collect constants representing sizes from callsites of functions like `malloc`. To address this, we design a *Backward Constant Tracking* algorithm that traces constant usage backward from key callsite's arguments in the TFG, enabling identification of wrapper functions and attribution of size properties to NMAE nodes. Another challenge is how member type hints should propagate within the whole-program TFG to accurately capture conflicts, as insufficient member type hints could lead to missed conflicts during detection. To address this challenge, we observe that within each single program execution path, all member accesses to a composite data type must be valid and conflict-free, as invalid access would otherwise result in runtime errors. Therefore, we first identify source nodes in the whole-program TFG, and utilize the $dataflow$ edges to traverse all possible program execution paths. We then collect all member type hints from each source node's possible execution paths and construct layouts, which are then used to detect conflicts between layouts and identify "evil edges".

Algorithm 2: Conflict-Aware Type Hint Propagation

```
Input: Whole-program TFG $\mathcal{G}$
Output: Variable-Constraint Mapping $\mathcal{M}$
1  $\mathcal{A} \leftarrow \{a \mid a \in \mathcal{G}.\textsc{GetSizeSensitiveParams}()\}$
2  foreach argument node $a \in \mathcal{A}$ do
3      $(expr_r, \sigma) \leftarrow$ $\textsc{BackwardTrackConstants}(\mathcal{G}, a)$
4      $\mathcal{G} \leftarrow \mathcal{G} \setminus \textsc{RemoveWrapperEdges}(expr_r)$
5      $expr_r.\textsc{SetSize}(\sigma)$
6  end
7  $E_{evil} \leftarrow \textsc{MultiSourceSizePropagation}(\mathcal{G})$
8  $\mathcal{G} \leftarrow \mathcal{G} \setminus E_{evil}$
9  foreach source node $expr_s \in \mathcal{G}.\textsc{GetSources}()$ do
10     $\mathcal{L} \leftarrow \textsc{BuildLayoutFromPaths}(\mathcal{G}, expr_s)$
11     $\mathcal{G} \leftarrow \textsc{UpdateLayout}(\mathcal{G}, expr_s, \mathcal{L})$
12 end
13 $E_{evil} \leftarrow \textsc{MultiSourceLayoutPropagation}(\mathcal{G})$
14 $\mathcal{G} \leftarrow \mathcal{G} \setminus E_{evil}$
15 $\mathcal{G} \leftarrow \textsc{AddInterTypeAlias}(\mathcal{G})$
16 $\mathcal{M} \leftarrow \textsc{BuildConstraintMap}(\mathcal{G})$
17 return $\mathcal{M}$
```

Algorithm 2 outlines the complete process of Conflict-Aware Type Hint Propagation. This algorithm consists of three phases: first, it uses composite type sizes to detect conflicts and remove "evil edges"; second, it employs layout information for more refined conflict detection and edge removal; finally, it adds inter-procedural $typealias$ edges and builds a mapping from decompiler-inferred variables to composite type constraints.

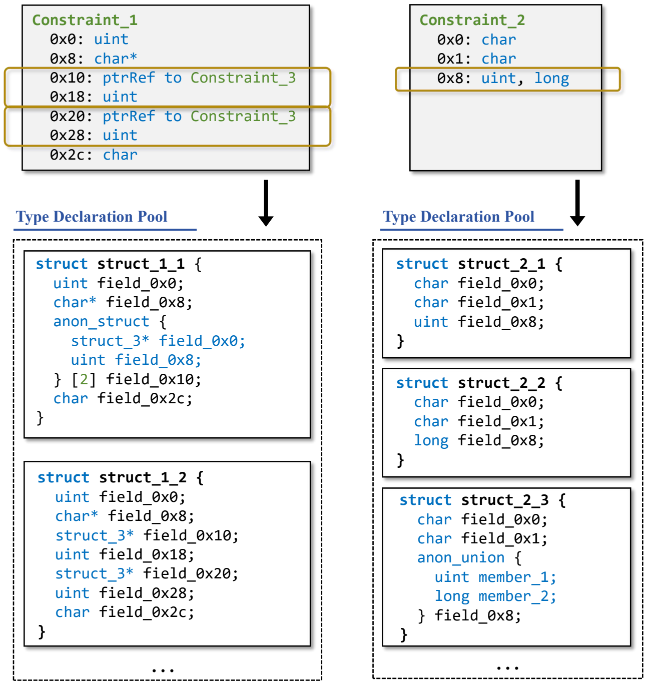
Specifically, In the first phase, Algorithm 2 begins by collecting size-related parameters from functions such as `malloc` and `memset` (line 1), then performs Backward Constant Tracking on the TFG to identify which constant arguments at callsites ultimately propagate to these size-sensitive parameters (line 3). These size values are then added as node attributes to the callsite's receiver nodes (line 5). Subsequently, based on these size attributes, a BFS-based multi-source size propagation is employed to detect conflicts and identify and remove "evil edges" (lines 7-8). However, not all "evil edges" can be detected based solely on size information, as there exist cases where sizes cannot be calculated or where composite types have identical sizes but different internal layouts. Therefore, in the second phase, Algorithm 2 further employs layout information for more refined conflict detection. The algorithm first constructs layouts based on source nodes and their paths in the TFG (lines 10-11), then, these layout specifications are propagated along paths from sources to connected nodes. Some nodes will contain conflicting layouts from different sources, enabling the identification and removal of additional ”evil edges” (lines 13-14). Finally, after eliminating the impact of “evil edges”, the algorithm adds inter-procedural *typealias* edges to TFG nodes and constructs the final composite type constraints (lines 15-16), including member layouts, pointer reference, and nesting relationships. Furthermore, the decompiler-inferred variables are then mapped to these constraints, clearly establishing the correspondence between variables and their respective composite type constraints.

3.2.3. Type Declaration Synthesis. Most existing approaches typically only recovers a single composite data type declaration directly from member offsets and sizes. However, due to the ambiguity caused by the lack of type information in stripped binaries, the same constraints may lead to multiple potential data type declarations. Through a comprehensive analysis, we identify the following key factors contributing to this ambiguity in recovering composite data types.

- *F1: Loss of Type Boundaries Due to Flattening.* During compilation, structures and arrays allocated on the stack or nested within other data types are often flattened into separate variables, removing the clear boundaries that originally distinguished related data types. This flattening obscures whether a variable is part of a larger structure or array, or if it stands alone as an independent entity.
- *F2: Ambiguity in Determining Whether a Pointer References a Single Object or an Array.* For example, if a node in TFG has four *member* edges, each member with a size of 1 byte, it may be ambiguous whether the node represents a pointer to a `char` array or a pointer to a structure with four 1-byte members. Similarly, a `struct_A*` pointer could point to either a single `struct_A` instance or an array of `struct_A`, with the latter potentially leading to constraints that incorrectly include multiple repeated members.
- *F3: Absence of Type Casting and Union Information.* As mentioned in Section 3.2.2, Algorithm 2 is used for detecting conflicts and removing “evil edges” in TFG. However, for nodes associated with these edges, it remains difficult to determine whether they represent unions or other types involving type casting.

To address the ambiguities caused by the above-mentioned factors, we observe that by strategically combining, selecting, and splitting members within constraints according to certain conventions, multiple potential composite data type declarations can be synthesized, with the correct one often among them. For example, when member constraints of a composite data type exhibit repetitive patterns, this suggests the corresponding type could be either a single structure or a flattened array of structures. Based on these insights, we designed the *adaptive sliding window algorithm* to identify and leverage these characteristics within constraints, guiding the combination, selection, and splitting of member constraints to synthesize a comprehensive series of candidate composite data type declarations.

Specifically, the algorithm employs an iterative scanning process, with each iteration referred to as a *Pass*. During each *Pass*, the algorithm identifies characteristics within the constraints and uses them to synthesize a series of composite data type declarations. The window size is then dynamically adjusted, and the process continues until the window spans the entire constraint.


```
Constraint_1
  0x0: uint
  0x8: char*
  0x10: ptrRef to Constraint_3
  0x18: uint
  0x20: ptrRef to Constraint_3
  0x28: uint
  0x2c: char
```

```
struct struct_1_1 {
  uint field_0x0;
  char* field_0x8;
  anon_struct {
    struct_3* field_0x0;
    uint field_0x8;
  } [2] field_0x10;
  char field_0x2c;
}
```
```
struct struct_1_2 {
  uint field_0x0;
  char* field_0x8;
  struct_3* field_0x10;
  uint field_0x18;
  struct_3* field_0x20;
  uint field_0x28;
  char field_0x2c;
}
```

(a)

```
Constraint_2
  0x0: char
  0x1: char
  0x8: uint, long
```

```
struct struct_2_1 {
  char field_0x0;
  char field_0x1;
  uint field_0x8;
}
```
```
struct struct_2_2 {
  char field_0x0;
  char field_0x1;
  long field_0x8;
}
```
```
struct struct_2_3 {
  char field_0x0;
  char field_0x1;
  anon_union {
    uint member_1;
    long member_2;
  } field_0x8;
}
```

(b)

Figure 8: Synthesize type declarations from constraints.

Figure 8 illustrates the working mechanism of adaptive sliding window algorithm through two intuitive examples. The *Pass* in Figure 8(a) employs a sliding window to identify duplicate member constraints at offsets `0x10` and `0x20`, indicating the potential presence of a flattened nested structure. As a result, two candidate composite data types `struct_1_1` and `struct_1_2` are created. Similarly, the *Pass* in Figure 8(b) detects a conflicting member constraint at offset `0x8`, where the member constraint could be satisfied by either a `uint` or a `long` type. Consequently, three possible composite data types are created: `struct_2_1` with `uint` as the type of `field_0x8`, `struct_2_2` with `long`, and `struct_2_3`, a union containing both `uint` and `long`, all synthesized composite data type declarations are stored in the Type Declaration Pool.

We currently design three specific *Passes* to target the ambiguities caused by the above factors. The first *Pass* focuses on recovering the boundaries of flattened data types and their nested relationships, the second *Pass* addresses conflicting member constraints, and the third *Pass* is dedicated to resolving the types of nodes in the TFG that are associated with “evil edges”. According to our evaluation in Section 5.3, these three *Passes* are sufficient to recover the layout and relationships of the majority of composite data types. Furthermore, our algorithm is easy to implement and integrates additional *Passes* if needed.

3.3. Readability-Guided Selection

3.3.1. Retyping. TypeForge first retrieves all composite type declarations from the Type Declaration Pool corresponding to each constraint. It then utilizes the *Variable-Constraint Mapping* constructed by Algorithm 2 to assign different type declarations to variables through the decompiler's retype interface, thereby generating multiple variants of the decompiled code. For each constraint, this process ultimately forms a Code-Type Map (CTMap), represented as $CTMap = \{\langle D_1 : Type_1\rangle, \langle D_2 : Type_2\rangle, \ldots, \langle D_n : Type_n\rangle\}$, where $Type_i$, $i = 1, 2, \ldots, n$ represent different composite data type declarations synthesized from the same constraint, and D_i represents the set of decompiled code collected after applying $Type_i$ through the decompiler's retype interface. Each D_i within the $CTMap$ can be expressed as $D_i = \{d_{f_1}, d_{f_2}, \ldots, d_{fm}\}$, where the subscript f_i denotes the functions in the stripped binary and d_{f_i} represents the decompiled code for that function. For efficiency, we only collect functions that access ambiguous members within the composite data type. Notably, for any D_i and D_j within the same $CTMap$, the functions they contain are identical and can be represented as $D_i = \{d_{f_1}, d_{f_2}, \ldots, d_{fm}\}$ and $D_j = \{d'_{f_1}, d'_{f_2}, \ldots, d'_{fm}\}$. Here, each d_{fi} and d'_{fi} represent different variants of decompiled code for the same function, with the only difference being the assignment of different type declarations to the same variables within the functions, resulting in structural changes in the decompiled code.

To assess the readability of decompiled code variants in $CTMap$ while simultaneously considering both code syntax and semantics and selecting the best-fit composite data type declaration, the most straightforward and cost-effective approach is to directly score the decompiled code using a large language model (LLM). However, the absence of reference standards in decompiled code can lead to subjective and inconsistent outcomes. Additionally, the inherent randomness of LLMs can introduce fluctuations and biases in the results, further compromising the accuracy and reliability of our selections. Therefore, it is a challenge to *ensure reliable and objective results without incurring significant overhead.*

3.3.2. LLM-assisted Double-elimination. To address the above challenge, we reference and design an *LLM-assisted double-elimination mechanism* [20] that pairs decompiled code variants for comparative readability assessment. Compared to direct scoring, this mechanism introduces a comparative reference, making our assessment more objective and allowing TypeForge to determine which decompiled code variant exhibits stronger readability.

Figure 9 provides a simple example illustrating how this mechanism works. Initially, all $CTPair : \langle D_i : Type_i\rangle$ within the $CTMap$ corresponding to each constraint start in the winner's bracket (Green border). During each round, if decompiled code in a $CTPair$ is judged to have lower readability, it moves to the loser's bracket (Yellow border). It is eliminated from the competition if it loses again in the loser's bracket. To illustrate their comparison process, we mark the comparison process of $\langle D_2 : Type_2\rangle$ and $\langle D_3 : Type_3\rangle$ in green and yellow. Ultimately, the last remaining $CTPair$ in the winner's bracket faces the last remaining $CTPair$ from the loser's bracket to determine the final winner. To mitigate the impact of the LLM's inherent randomness, each comparison round is executed in parallel t times (with $t = 3$ in our evaluation), and the $CTPair$ with the highest number of wins is selected. The expected number of comparisons in our double-elimination mechanism is approximately $2n - 2$, where n is the total number of $CTPairs$. This is significantly lower than the n^2 comparisons required for exhaustive pairwise comparisons.

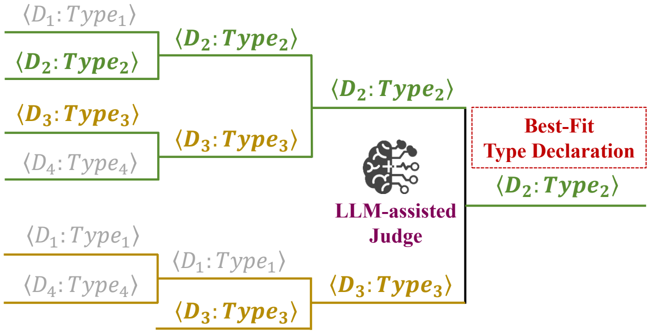

$\langle D_1 : Type_1\rangle$
$\langle D_2 : Type_2\rangle$
$\langle D_2 : Type_2\rangle$
$\langle D_3 : Type_3\rangle$
$\langle D_4 : Type_4\rangle$
$\langle D_3 : Type_3\rangle$
$\langle D_2 : Type_2\rangle$
Best-Fit Type Declaration
$\langle D_2 : Type_2\rangle$
LLM-assisted Judge
$\langle D_1 : Type_1\rangle$
$\langle D_4 : Type_4\rangle$
$\langle D_1 : Type_1\rangle$
$\langle D_3 : Type_3\rangle$
$\langle D_3 : Type_3\rangle$

Figure 9: An intuitive example demonstrating the LLM-assisted double-elimination mechanism in TypeForge.

Specifically, in each round of $CTPair$ comparison, when comparing $\langle D_i : Type_i\rangle$ and $\langle D_j : Type_j\rangle$, where $D_i = \{d_{f_1}, d_{f_2}, \ldots, d_{fm}\}$ and $D_j = \{d'_{f_1}, d'_{f_2}, \ldots, d'_{fm}\}$, TypeForge construct a set of code pairs as follows: $CodePairs = \{\langle d_{f_1}, d'_{f_1}\rangle, \langle d_{f_2}, d'_{f_2}\rangle, ..., \langle d_{fm}, d'_{fm}\rangle\}$, and use this $CodePairs$ as input for the LLM-assisted Double-Elimination process.

Syntactic Clarity Metric. To ensure objectivity and accuracy in readability assessment as much as possible, we collect a set of syntactic features related to composite data types and calculate a syntactic clarity metric to assist the LLM in making more accurate judgments. These features are derived from variables, expressions, and statements associated with member accesses to composite data types. For example, decompiled code without well-recovered types often contains numerous local variables, pointer arithmetic, direct memory accesses, and typecasting. In contrast, code with well-recovered types typically has fewer of these hard-to-read operations, instead showing an increase in the use of member access operators `.` and `->`, as well as the subscript operator `[]`. We count their occurrences and follow a similar method in R2I [23] to compute the syntactic clarity metric.

Prompt Design. TypeForge interacts with the LLM using the prompt shown in the box below, where d_{f_i} and d'_{f_i} represent the respective decompiled code variants of the same function, and $metric_{f_i}$ and $metric'_{f_i}$ contain the corresponding syntactic clarity metrics.

Based on the LLM's response, TypeForge judges the relative readability of the two decompiled code snippets. This process continues until the final $\langle D_i : Type_i\rangle$ is selected, with $Type_i$ being considered the Best-Fit composite data type declaration.

It is worth mentioning that since each constraint corresponds to an independent Type Declaration Pool and $CTMap$, our LLM-assisted readability-guided selection can be executed in parallel for efficient processing, further reducing time costs.

> 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:
> $\langle d_{f_1},\ metric_{f_1}\rangle\ \langle d'_{f_1},\ metric'_{f_1}\rangle$
> $\ldots$
> $\langle d_{f_m},\ metric_{f_m}\rangle\ \langle d'_{f_m},\ metric'_{f_m}\rangle$

4. Implementation

The prototype of TypeForge is implemented as a Ghidra Extension with over 12,000 lines of code. The *TFG-based Type Synthesis* consists of approximately 11,000 lines of Java code, about 1,000 lines of Python code are used for *Readability-Guided Selection*. TypeForge uses Ghidra [13] as the foundational analysis framework for disassembly, decompilation, and retype functionality because it is a widely used open-source binary analysis tool with a robust API and strong community support. Notably, Ghidra supports the analysis of stripped binaries across multiple architectures using its intermediate representation, PCode [26], which forms the foundation for TypeForge's analysis and enables TypeForge to analyze binaries from various architectures. Furthermore, due to the design of TFG being independent of any specific decompiler, TypeForge could utilize any decompiler capable of primitive type inference (such as IDA Pro and Binary Ninja) as its underlying analysis framework. For *Readability-Guided Selection*, TypeForge utilizes Tree-sitter [27] to calculate the syntactic clarity metric and automates interactions with LLMs using LangChain [28]. Our default model is OpenAI's GPT-4o mini [29], which supports up to 128k tokens.

5. Evaluation

5.1. Experiment Setup

Platform. All our experiments were conducted on a personal computer running Arch Linux with 8 processors Intel(R) Core i7-12700 CPU @ 2.10GHz and 24GB of memory. Ghidra version is 11.0.3.

Dataset. Previous studies [9], [18], [19] have built large and complex datasets for recovering composite data types from binaries, as the effectiveness of learning-based methods heavily relies on the quality of the training set. In contrast, one of the key advantages of TypeForge is that its performance primarily depends on accurately synthesized type declarations and LLM-assisted readability assessments. Since most existing LLMs are already trained on extensive code datasets, TypeForge can achieve decently robust performance without requiring additional training. Consequently, we only need a moderately sized dataset to evaluate TypeForge's effectiveness. To facilitate comparison with existing work, we use the same evaluation dataset as OSPREY [16], which includes GNU coreutils [30] and several larger, more complex real-world binary programs from the Howard [31] dataset. Our dataset contains over 10,000 functions and more than 11,000 variables that hold composite data types.

Metrics. Similar to previous work [16], [18], [19], we evaluate the TypeForge's effectiveness in *Composite Data Type Identification* and *Layout Recovery*. We also evaluate the effectiveness of *Relationship Recovery* between composite data types, such as structure pointer references, structure nesting, and unions, which are common in real-world open-source projects. Below, we will provide a detailed explanation of our metrics.

• ***Composite Data Type Identification:*** In this step, we evaluate the precision and recall of TypeForge in predicting whether variables are of composite data types within a stripped binary. We define V_{gt} as the ground-truth set of variables corresponding to composite data types, including structs and pointers to structs, unions and pointers to unions, and arrays, extracted from debug information. V_{infer} is defined as the set of all variables that TypeForge infers as corresponding to composite data types, while V^T_{infer} represents the correctly inferred variables. The recall, precision, and F1 score of *Composite Data Type Identification* are defined as:

$$Rec_{var} = \frac{V^T_{infer}}{V_{gt}}, \quad Pre_{var} = \frac{V^T_{infer}}{V_{infer}}$$
$$F1_{var} = \frac{2 \times Rec_{var} \times Pre_{var}}{Rec_{var} + Pre_{var}}$$

All subsequent F1 scores are defined similarly based on their precision and recall values.

• ***Layout Recovery:*** In this step, we focus on evaluating the recovery of struct layouts, as most composite data types in stripped binaries are structs and their pointers. Similar to the approach in previous work [19], we evaluate the layout based on the offsets and sizes of each member.

For a variable $v \in V_{infer}$ inferred by TypeForge, we define its Layout as a set of pairs:

$$L^v_{infer} = \{\langle offset_0, size_0\rangle,\ \ldots,\ \langle offset_n, size_n\rangle\}$$

Each pair represents the offset and size attribute of a member in the struct. For example, in Figure 4, the layout for `struct_74_a` is represented as $L^{param\_1}_{infer} = \{\langle 0x0, 0x4\rangle, \langle 0x8, 0x8\rangle, \langle 0x10, 0x4\rangle, \langle 0x18, 0x8\rangle, \langle 0x20, 0x4\rangle\}$. Similarly, we define the ground truth layout for a variable as L^v_{gt}. Finally, the recall and precision for layout recovery are defined as follows:

$$Rec_L = \frac{\sum_{v \in V} |L^v_{infer} \cap L^v_{gt}|}{\sum_{v \in V} |L^v_{gt}|},$$
$$Pre_L = \frac{\sum_{v \in V} |L^v_{infer} \cap L^v_{gt}|}{\sum_{v \in V} |L^v_{infer}|}$$

• ***Relationship Recovery:*** In source code, relationships between composite data types are easily identifiable by their type names. However, these type names are removed during compilation, making it necessary to develop a reliable and accurate approach to evaluate the recovery of relationships between structures. We define T_{infer} as the set of all composite data types inferred by TypeForge. The relationships between these types are represented as a set of 5-tuples: $Relations_{infer} = \{\langle t_{src}, offset, t_{dst}, RelationType, ptrLevel\rangle, \ldots, \langle ...\rangle\}$. Where t_{src} and $t_{dst} \in T_{infer}$, $RelationType$ specifies the type of relationship (including structure pointer reference, structure nesting, and union), and $ptrLevel$ indicates the pointer reference level. We similarly define the ground truth relationships as $Relations_{gt}$. Since type names are unavailable, we use layout to determine the correctness of relationship recovery. Specifically, for each 5-tuple in $Relations_{infer}$, we consider the relationship correctly recovered if the layouts of t^{infer}_{src} and t^{infer}_{dst} match those in $Relations_{gt}$ and the $offset$, $RelationType$, and $ptrLevel$ also align with $Relations_{gt}$. Finally, the recall and precision for *Relationship Recovery* are defined as follows:

$$Rec_r = \frac{|Relations_{infer} \cap Relations_{gt}|}{Relations_{gt}},$$
$$Pre_r = \frac{|Relations_{infer} \cap Relations_{gt}|}{|Relations_{infer}|}$$

5.2. Effectiveness

Table 2 presents the results of TypeForge across different optimization levels for the three aforementioned metrics on our evaluation dataset. We first analyze the data for each metric in detail and then measure the runtime of each module within TypeForge.

The data show that TypeForge achieves its strongest overall performance at optimization level O2 for Composite Data Type Identification, with an overall precision of 74.9%, a recall of 82.9% and an F1 score of 78.8%. This is likely because, under O2 optimization, many non-composite local variables are merged or removed by the compiler, thereby reducing interference with TypeForge's analysis. Overall, TypeForge performs consistently well in Composite Data Type Identification across all optimization levels, with minimal performance variation.

TABLE 2: TypeForge's Performance Metrics (Precision, Recall, and F1 score (%)) across optimization levels for Composite Data Type Identification, Layout Recovery and Relationship Recovery

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
Relationship Recovery	Precision	47.9	55.4	**94.2**	66.9
	Recall	**50.3**	28.9	30.8	22.2
	F1 Score	**49.1**	37.9	46.4	33.3

TABLE 3: TypeForge's Time Consuming Analysis at optimization level O0

Project	LoC	TFG-based Type Synthesis	Readability-Guided Selection		Total
			Retyping	Double-elimination	
coreutils	4.8k	1.7s	4.1s	8.4s	14.2s
gzip	10k	5.1s	14.7s	20.4s	40.2s
grep	27k	3.9s	1.2s	4.1s	9.2s
wget2	43k	6.1s	1.7s	5.2s	13.0s
lighttpd	69k	17.6s	21.9s	44.2s	83.7s

In the context of Layout Recovery, TypeForge achieves its optimal performance at optimization level O0, with an overall precision of 97.9%, a recall of 82.2%, and an F1 score of 89.4%, as shown in Table 2. At higher optimization levels (O1–O3), TypeForge maintains high precision while exhibiting gradually declining recall for layout recovery, which aligns with our expectations: under aggressive compiler optimizations, composite type members are often merged or flattened, thereby introducing challenges for layout reconstruction. Specifically, the *Conflict-Aware Type Hint Propagation* adopts a conservative strategy to prevent type hints from propagating along "evil edges". However, at higher optimization levels, many conflicts detected by this algorithm are compiler-induced and indistinguishable, which may inadvertently remove legitimate data flow edges. This results in failed propagation of valid type hints that would otherwise propagate correctly. Furthermore, the lower recall rate can be attributed to two additional key factors: (1) **Not all composite data type members are accessed within the program.** For members that are never accessed within the program, TypeForge cannot collect any information about them, a common limitation faced by all data flow analysis tools. (2) **The same pointer may be interpreted as different composite data types on different locations in the program.** For example, if structure A is nested as the first member of structure B, a pointer to structure B may be interpreted as pointing to structure A through upcasting at certain program locations. We refer to this situation as *Pointer Type Ambiguity*, which can result in mismatches between the actual type that the pointer references (as inferred by TypeForge) and the ground truth collected from debugging symbols.

Table 2 further demonstrates that TypeForge recovers relationships with a precision of 47.9% and a recall of

TABLE 4: Comparison with SOTA approaches on three metrics in Precision (P), Recall (R) and F1 score (F1) (%), as well as the analysis time consumed by each approach.

	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**

50.3%, and an F1 score of 49.1% at optimization level O0, achieving its best performance across optimization levels O0-O3. Compared to previous tasks, relationship recovery is the most challenging. This difficulty arises because relationship recovery not only relies on accurate layout recovery but also requires accurately handling the complex pointer relationships within the program. Additionally, under higher optimization levels, complex nested relationships are often flattened by compilers, lose their original boundary information, making recovery even more difficult. Nevertheless, relying on *TFG-based Type Synthesis* and *Readability-Guided Selection*, TypeForge could still recover a significant portion of these complex relationships between structures, something that current state-of-the-art approaches have been unable to achieve. A detailed comparison with state-of-the-art approaches [9], [16], [18], [19] will be provided in Section 5.3.

Table 3 presents TypeForge's average time consumption at optimization level O0 across several projects in our dataset. LoC represents lines of Code, with the projects arranged from smallest to largest based on LoC. Overall, *TFG-based Type Synthesis*, which builds constraints and synthesizes possible type declarations, incurs the lowest time consumption, primarily due to the efficient design of TFG and Conflict-Aware Type Hint Propagation. Notably, our program analysis technique achieves significantly lower runtime overhead than existing state-of-the-art methods while maintaining high accuracy (requiring only 17.6 seconds on lighttpd), demonstrating its potential for future adaptation to other binary analysis tasks. The time consumption in *Readability-Guided Selection* is directly correlated with the number of type declarations generated in stage one. As this number increases, the retyping process during decompilation requires more time, and our double-elimination mechanism produces more code pairs for LLM-assisted assessment. Nevertheless, the overall time overhead remains considerably lower than existing approaches, as detailed in Section 5.3.

TypeForge can be applied to various downstream security analysis tasks, including decompilation, vulnerability detection, indirect call recovery, and exploitation. To further demonstrate the effectiveness of TypeForge, we applied TypeForge to two important downstream tasks: decompiled code optimization and vulnerability detection, with detailed results in the Appendix A.

5.3. Comparison with State-of-the-Arts

We compare TypeForge against state-of-the-art (SOTA) approaches, including DIRTY [9], OSPREY [16], ReSym [19], and TYGR [18]. Our evaluation uses the coreutils dataset provided by OSPREY (different from Section 5.2's coreutils dataset), which has been widely adopted by other related works. For DIRTY, we selected the “DIRTY-Multitask” model, which has demonstrated superior performance among their available models. Since OSPREY is not publicly available, its authors generously provided us with their results on the dataset. Additionally, as ReSym's artifacts are not yet fully available, we utilized the published results on the same dataset with permission from the authors.

Table 4 shows the effectiveness of these approaches on *Composite Data Type Identification*, *Layout Recovery, and Relationship Recovery*, and we mark approaches that either fail to complete the task or do not provide results with a dash. In *Composite Data Type Identification*, TYGR performs the best among existing approaches, followed by DIRTY. OSPREY performs worse primarily because it cannot recover register-based variables, including function parameters and some local variables. TypeForge, however, significantly outperforms the existing best approach, TYGR, with a 32.4% improvement in F1 score. Notably, TYGR's functionality focuses on using a GNN-based classifier that determines whether a variable's type is a struct pointer (`struct*`) or a stack-allocated struct, which means their “Struct Accuracy” metric evaluates the same capability as our Composite Data Type Identification metric.

In *Layout Recovery*, TypeForge achieves a 39.2% higher F1 score than ReSym and a 1.3% higher F1 score than OSPREY, which are two of the best performers among existing approaches. Our recall is slightly lower than OSPREY due to the same reason previously mentioned: the algorithm we use to eliminate “evil edges” may also remove valid ones, which prevents some type hints from being properly propagated. However, the benefit of this algorithm is a significantly higher accuracy. As shown in the figure, TypeForge achieves an accuracy of 96.9%, nearly 10% higher than OSPREY. TypeForge's high precision enables it to be effectively applied in downstream security tasks with a very low false positive rate. Another noteworthy detail is that due to TYGR's lack of inter-procedural analysis, its "struct member type prediction" functionality works only for stack-allocated structs and cannot recover layouts for struct pointers, which constitute the majority of composite types. As a result, TYGR achieves poor performance in our Layout Recovery metric, with a recall of only 5.3% and a precision of 31.3%.

In *Relationship Recovery*, we mark approaches that cannot complete the task or provide data as a dash. Since DIRTY can select complete composite data type declarations from its training dataset based on decompiled code, we fully parse its results to meet the evaluation criteria of Relationship Recovery. However, DIRTY still performs poorly in this task, with no relationships accurately recovered. In contrast, TYPEFORGE successfully recovers relationships between composite data types with 78.5% precision and 66.6% recall, significantly outperforming DIRTY.

We also compared the analysis time. To ensure fairness, we start timing each approach from the initial analysis phase. For DIRTY, we calculated the total time for decompilation, model inference, and subsequent analysis. TYGR requires extracting data flow information from stripped binaries as input to the model, so we calculated the total time for data flow analysis and model inference. Since OSPREY is not publicly available to run in our hardware environment, we adopted the execution times reported in the OSPREY paper and marked them (†) accordingly in Table 4. Nevertheless, based on hardware performance differentials, we can provide an optimistic estimate of OSPREY's time cost, projecting an average execution time of approximately 200 seconds in our hardware environment. The results in Table 4 show that TYPEFORGE has the second-lowest analysis overhead, with DIRTY having the lowest. However, TYPEFORGE achieves significantly better overall performance than DIRTY. Additionally, even when compared to the optimistic time estimate for the best-performing existing approach, OSPREY, TYPEFORGE still only requires 3.8% of OSPREY's analysis time.

TYPEFORGE performs strongly in composite data type recovery, achieving high accuracy with minimal time overhead. However, since TYPEFORGE leverages LLMs, we also estimated its token usage. On the coreutils dataset, TYPEFORGE consumes an average of 180,000 tokens per binary, costing approximately $0.12 using the gpt-4o-mini [29] model. TYPEFORGE has demonstrated significant potential in addressing various real-world reverse engineering tasks.

5.4. Ablation Study

To demonstrate the importance of each component in TYPEFORGE's design, we conducted two sets of ablation studies: (1) We evaluate the importance of *Readability-Guided Selection* (Section 3.3.2) by comparing it with a simple heuristic approach that directly synthesizes final type declarations deterministically from constraints. (2) We evaluate the significance of *Conflict-Aware Type Hint Propagation* by comparing it with a method that propagates type hints across all nodes with $dataflow$ edges.

TABLE 5: Ablation study demonstrating the effectiveness of each component

	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**
TypeForge$_{\text{fast}}$	66.2	**69.3**	**67.7**	55.8	73.4	63.4
TypeForge$_{\text{evil}}$	79.3	87.7	83.2	87.6	**15.5**	**26.9**

Without Readability Assessment. We implemented a straightforward heuristic that deterministically generates final type declaration from constraints, and we refer to this method as TYPEFORGE$_{fast}$. Table 5 shows the performance of TYPEFORGE compared with TYPEFORGE$_{fast}$. As can be seen, without readability assessment, precision drops significantly, primarily due to the increase in erroneous type and member identification.

Propagate Type Hints Directly. We denoted the method that indiscriminately propagates type hints to all nodes with $dataflow$ edges as TYPEFORGE$_{evil}$, and compared it with TYPEFORGE. The results are also shown in Table 5. It is evident that TYPEFORGE$_{evil}$ performs significantly worse than TYPEFORGE does, particularly in *Layout Recovery* because due to the presence of typecasting and union, TYPEFORGE$_{evil}$ tends to merge type hints from unrelated composite data types, resulting in numerous false positives and conflicts.

6. Discussion

Limitations. In this paper, we propose a two-stage comprehensive-selection approach, TYPEFORGE, to accurately and efficiently recover composite data types from stripped binary files by simulating the workflow of reverse engineering experts. However, TYPEFORGE has the following limitations. (1) The algorithm used in the *Conflict-Aware Type Hint Propagation* to identify "evil edges" caused by type casting and unions may result in some false positives, leading to the removal of some valid $dataflow$ edges as well. Specifically, when Conflict-Aware Type Hint Propagation propagates size and layout information to detect conflicts, the algorithm cannot guarantee that detected "evil edges" always correspond to unions or typecasting in the source code. However, to prevent incorrect propagation of type hints, we adopt a conservative strategy of removing all edges detected as causing conflicts. This conservative approach results in some recovered composite data types may be missing certain members, as valid $dataflow$ edges may be inadvertently removed. (2) Approximately 75% of TYPEFORGE's time overhead is spent on *Readability-Guided Selection* using LLMs. This occurs because current LLMs require more time to generate output for longer prompts.

Future Work. We believe the most promising direction for future work is to incorporate fuzzing concepts to further enhance TYPEFORGE. Currently, TYPEFORGE generates composite data type declarations through a search process that does not utilize feedback from the readability of decompiled code. In the future, we plan to focus on designing a feedback mechanism inspired by fuzzing and developing more efficient and accurate readability assessment methods. These improvements aim to optimize both the generation and selection of composite data type declarations. By incorporating this feedback-driven approach, it would be possible to explore a broader range of potential composite data type declarations, ultimately achieving better recovery results.

7. Related Work

7.1. Binary Type Inference

Binary type inference is critical for many security analysis tasks, such as decompilation [7], [8], malware analysis [10], [11], [12], vulnerability detection [2], [3], [4], and root cause analysis [32], [33]. Over the past decade, significant works in this area include TIE [15], REWARDS [34], ReTyped [35], OSPREY [16], DIRTY [9], and ReSym [19]. TIE, ReTyped, and OSPREY are static approaches that rely on costly binary analysis algorithms. TIE constructs a lattice-based type system to support binary type inference, while ReTyped establishes a more sound type system for precisely recovering polymorphic and recursive types. OSPREY achieves more accurate type inference through binary dependency analysis [24] and probabilistic analysis. However, being limited by their underlying data flow analysis algorithms, these approaches incur substantial computational and time overhead during inter-procedural analysis, making them less practical for real-world reverse engineering tasks. REWARDS uses dynamic analysis to infer composite data types by tracking variables to their final sink points. However, it is limited by coverage and does not support the recovery of user-defined structures. DIRTY and ReSym are learning-based approaches that use models to predict composite data types and layouts. However, DIRTY is restricted to types present in its training set, while ReSym is constrained by token limits in model input, making it applicable only to smaller functions. In this paper, we propose TypeForge, which utilizes a novel graph structure *Type Flow Graph* and *Conflict-Aware Type Hint Propagation* to efficiently and accurately construct composite data type constraints. Based on these constraints, TypeForge further employs *Readability-Guided Selection* to address the inherent ambiguity in composite type inference. Compared to previous approaches, TypeForge demonstrates superior efficiency and accuracy, making it well-suited for real-world reverse engineering tasks.

7.2. LLM for Program Analysis

Large Language Models (LLMs), such as GPTs [29] and LLaMA [36], have demonstrated impressive results in various code-related tasks, including code understanding [37], [38] and code generation [39], [40]. As LLMs continue to evolve, many researchers are exploring their application in program analysis, particularly in areas like decompilation [6], [19], vulnerability detection [1], [2], program repair [41], [42], and fuzz testing [43], [44]. However, due to the need for a global view and detailed semantic capture of the entire binary program, LLMs are not well-suited for directly recovering composite data types from stripped binaries. In this paper, inspired by the use of LLMs for reference-free text quality evaluation in NLP [45], we leverage LLMs to assess the readability of decompiled code and use this method to select the best-fit composite data type from a series of candidates.

8. Conclusion

In this paper, we present TypeForge, a novel approach for automatic recovery of composite data types from stripped binaries, which employs a two-stage synthesis-selection strategy to emulate the workflow of experts. We evaluate TypeForge against state-of-the-art approaches, and the experimental results show it achieves high F1 scores of 81.7% and 88.2% in Composite Data Type Identification and Layout Recovery, respectively, overall outperforming existing state-of-the-art methods. Furthermore, TypeForge is capable of recovering relationships between composite data types, with an F1 Score of 72.1%, a task that the majority of existing approaches cannot handle. Additionally, TypeForge is far more efficient, requiring just about 3.8% of the time needed by the best existing approach. These results demonstrates TypeForge's ability to address the challenges encountered in real-world reverse engineering scenarios effectively.

Acknowledgments

We thank the Shepherd and reviewers for their constructive feedback. The authors are supported in part by NSFC (62302497, U24A20236, 92270204), CAS Project for Young Scientists in Basic Research (Grant No. YSBR-118).

References

[1] Y. Liu, C. Zhang, F. Li, Y. Li, J. Zhou, J. Wang, L. Zhan, Y. Liu, and W. Huo, "Semantic-enhanced static vulnerability detection in baseband firmware," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 173:1–173:12.

[2] J. Zhao, Y. Li, Y. Zou, Z. Liang, Y. Xiao, Y. Li, B. Peng, N. Zhong, X. Wang, W. Wang, and W. Huo, "Leveraging semantic relations in code and data to enhance taint analysis of embedded systems," in *USENIX Security Symposium*, 2024.

[3] P. Liu, Y. Zheng, C. Sun, C. Qin, D. Fang, M. Liu, and L. Sun, "FITS: inferring intermediate taint sources for effective vulnerability analysis of iot device firmware," in *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, T. M. Aamodt, M. M. Swift, and N. D. E. Jerger, Eds. ACM, 2023, pp. 138–152.

[4] H. Han, J. Kyea, Y. Jin, J. Kang, B. Pak, and I. Yun, "Queryx: Symbolic query on decompiled code for finding bugs in COTS binaries," in *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*. IEEE, 2023, pp. 3279–312 795.

[5] Z. Gao, C. Zhang, H. Liu, W. Sun, Z. Tang, L. Jiang, J. Chen, and Y. Xie, “Faster and better: Detecting vulnerabilities in linux-based iot firmware with optimized reaching definition analysis,” *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[6] P. Hu, R. Liang, and K. Chen, “Degpt: Optimizing decompiler output with llm,” *Proceedings 2024 Network and Distributed System Security Symposium*, 2024.

[7] Y. Cao, R. Liang, K. Chen, and P. Hu, “Boosting neural networks to decompile optimized binaries,” in *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*. ACM, 2022, pp. 508–518.

[8] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. Le Goues, “A taxonomy of C decompiler fidelity issues,” in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024.

[9] Q. Chen, J. Lacomis, E. J. Schwartz, C. L. Goues, G. Neubig, and B. Vasilescu, “Augmenting decompiler output with learned variable names and types,” in *USENIX Security Symposium*, 2022.

[10] Y. Chen, S. Lin, S. Huang, C. Lei, and C. Huang, “Guided malware sample analysis based on graph neural networks,” *IEEE Trans. Inf. Forensics Secur.*, vol. 18, pp. 4128–4143, 2023.

[11] D. Corlatescu, A. Dinu, M. Gaman, and P. Sumedrea, “Embersim: A large-scale databank for boosting similarity search in malware analysis,” in *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, A. Oh, T. Naumann, A. Globerson, K. Saenko, M. Hardt, and S. Levine, Eds., 2023.

[12] F. Barr-Smith, X. Ugarte-Pedrero, M. Graziano, R. Spolaor, and I. Martinovic, “Survivalism: Systematic analysis of windows malware living-off-the-land,” in *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*. IEEE, 2021, pp. 1557–1574.

[13] “Ghidra,” https://ghidra-sre.org/, 2023.

[14] “hex-rays,” https://hex-rays.com, 2023.

[15] J. Lee, T. Avgerinos, and D. Brumley, “Tie: Principled reverse engineering of types in binary programs,” in *Network and Distributed System Security Symposium*, 2011.

[16] Z. Zhang, Y. Ye, W. You, G. Tao, W.-C. Lee, Y. Kwon, Y. Aafer, and X. Zhang, “Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary,” *2021 IEEE Symposium on Security and Privacy (SP)*, pp. 813–832, 2021.

[17] G. Balakrishnan and T. Reps, “Wysinwyx: What you see is not what you execute,” *ACM Trans. Program. Lang. Syst.*, vol. 32, no. 6, aug 2010.

[18] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé, M. Naik, Y. Shoshitaishvili, R. Wang, and A. Machiry, “Tygr: Type inference on stripped binaries using graph neural networks,” in *USENIX Security Symposium*, 2024.

[19] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, “Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries,” in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 4554–4568.

[20] H. Aziz, S. Gaspers, S. Mackenzie, N. Mattei, P. Stursberg, and T. Walsh, “Fixing balanced knockout and double elimination tournaments,” *Artif. Intell.*, vol. 262, pp. 1–14, 2018.

[21] T. McCabe, “A complexity measure,” *IEEE Transactions on Software Engineering*, vol. SE-2, no. 4, pp. 308–320, 1976.

[22] A. Gussoni, A. D. Federico, P. Fezzardi, and G. Agosta, “A comb for decompiled c code,” *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*, 2020.

[23] H. Eom, D. Kim, S. Lim, H. Koo, and S. Hwang, “R2i: A relative readability metric for decompiled code,” *Proc. ACM Softw. Eng.*, vol. 1, pp. 383–405, 2024.

[24] Z. Zhang, W. You, G. Tao, G. Wei, Y. Kwon, and X. Zhang, “Bda: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation,” *Proceedings of the ACM on Programming Languages*, vol. 3, pp. 1 – 31, 2019.

[25] “binary-ninja,” https://binary.ninja/, 2023.

[26] N. Naus, F. Verbeek, D. Walker, and B. Ravindran, “A formal semantics for p-code,” in *Verified Software. Theories, Tools and Experiments - 14th International Conference, VSTTE 2022, Trento, Italy, October 17-18, 2022, Revised Selected Papers*, ser. Lecture Notes in Computer Science, A. Lal and S. Tonetta, Eds., vol. 13800. Springer, 2022, pp. 111–128.

[27] “Tree-sitter,” https://tree-sitter.github.io/tree-sitter/, 2024.

[28] “Langchain,” https://github.com/langchain-ai/langchain, 2024.

[29] “gpt-4o-mini,” https://openai.com/, 2024.

[30] “Coreutils,” https://www.gnu.org/software/coreutils/, 2024.

[31] A. Slowinska, T. Stancescu, and H. Bos, “Howard: A dynamic excavator for reverse engineering data structures,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011.

[32] D. Xu, D. Tang, Y. Chen, X. Wang, K. Chen, H. Tang, and L. Li, “Racing on the negative force: Efficient vulnerability root-cause analysis through reinforcement learning on counterexamples,” in *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, D. Balzarotti and W. Xu, Eds. USENIX Association, 2024.

[33] W. Guo, D. Mu, X. Xing, M. Du, and D. Song, “DEEPVSA: facilitating value-set analysis with deep learning for postmortem program analysis,” in *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, N. Heninger and P. Traynor, Eds. USENIX Association, 2019, pp. 1787–1804.

[34] Z. Lin, X. Zhang, and D. Xu, “Automatic reverse engineering of data structures from binary execution,” in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3rd March 2010*. The Internet Society, 2010.

[35] M. Noonan, A. Loginov, and D. Cok, “Polymorphic type inference for machine code,” in *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2016, pp. 27–41.

[36] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” *CoRR*, vol. abs/2302.13971, 2023.

[37] D. Nam, A. Macvean, V. J. Hellendoorn, B. Vasilescu, and B. A. Myers, “Using an LLM to help with code understanding,” in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 2024, pp. 97:1–97:13.

[38] J. T. Liang, C. Yang, and B. A. Myers, “A large-scale survey on the usability of ai programming assistants: Successes and challenges,” *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*, pp. 616–628, 2023.

[39] L. Zhong and Z. Wang, “Can llm replace stack overflow? a study on robustness and reliability of large language model code generation,” in *AAAI Conference on Artificial Intelligence*, 2023.

[40] G. Sandoval, H. A. Pearce, T. Nys, R. Karri, S. Garg, and B. Dolan-Gavitt, “Lost at c: A user study on the security implications of large language model code assistants,” in *USENIX Security Symposium*, 2022.

[41] C. Xia and L. Zhang, “Less training, more repairing please: revisiting automated program repair via zero-shot learning,” *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022.

[42] H. Joshi, J. Cambronero, S. Gulwani, V. Le, I. Radicek, and G. Verbruggen, “Repair is nearly generation: Multilingual program repair with llms,” Aug 2022.

[43] Asmita, Y. Oliinyk, M. Scott, R. Tsang, C. Fang, and H. Homayoun, “Fuzzing busybox: Leveraging llm and crash reuse for embedded bug unearthing,” *ArXiv*, vol. abs/2403.03897, 2024.

[44] Y. Deng, C. Xia, H. Peng, C. Yang, and L. Zhang, “Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models,” Dec 2022.

[45] Y. Chen, R. Wang, H. Jiang, S. Shi, and R.-L. Xu, “Exploring the use of large language models for reference-free text quality evaluation: A preliminary empirical study,” *ArXiv*, vol. abs/2304.00723, 2023.

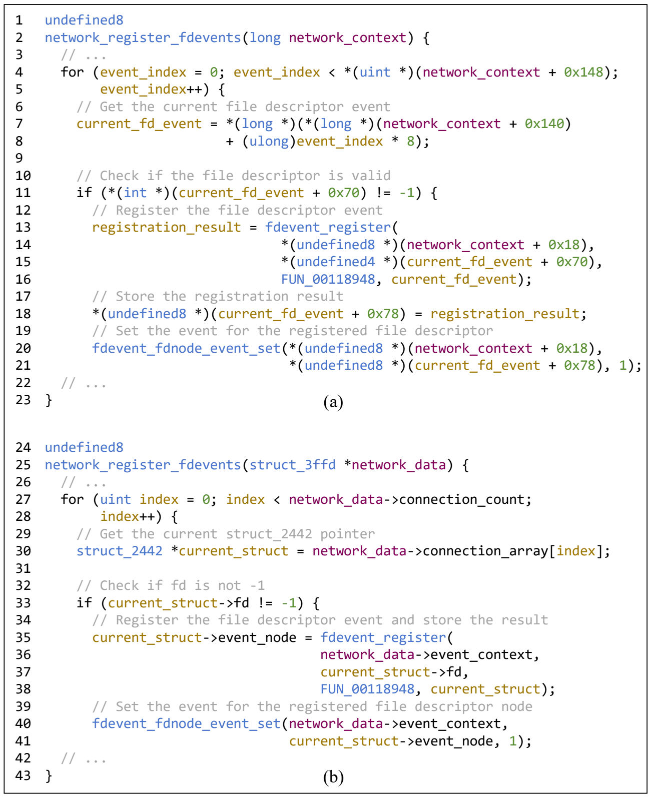
Appendix A. Downstream Security Analysis Tasks

A.1. Decompiled Code Optimization

A common downstream security analysis task supported by TypeForge is optimizing decompiled code to enhance its readability and accuracy. Widely used decompilers, such as IDA Pro [14], Ghidra [13], and Binary Ninja [25], often struggle to recover composite data types, particularly user-defined composite data types. This limitation results in decompiled code with poor readability, which negatively impacts the efficiency and accuracy of reverse engineers' analyses.

In recent years, research efforts such as DeGPT [6] have aimed to enhance the readability of decompiled code. However, these approaches typically focus on renaming variables and functions or adding comments, without addressing the recovery of composite data types. Consequently, their effectiveness in optimizing decompiled output remains limited. To illustrate this, we use a stripped binary from real-world ASUS router firmware as an example. Figure 10(a) shows the results of applying the state-of-the-art optimization approach, DeGPT, to Ghidra's decompiled code for a network preprocessing function in this program, without recovering composite data types. While variable and function names are partially recovered and some comments are added, the code's syntax and semantics remain unclear. It still contains numerous explicit type casts and complex memory access expressions, such as `*(long *)(network_context + 0x140)` and `(ulong)event_index * 8`, making it challenging for reverse engineers to fully understand the code's purpose.

Next, we use TypeForge to recover the composite data types in this program, and then reprocess the decompiled code with the recovered types as input for DeGPT's optimization, as shown in Figure 10(b). As illustrated, the readability of the type-recovered decompiled code has improved significantly compared to (a). Many explicit type casts and complex memory access expressions have been correctly translated into structure member accesses, with member names also successfully recovered, resulting in clearer overall code semantics. Additionally, certain complex code structures in (a) are now optimized. For example, the complex memory access statement on lines 7-8 is simplified into the more intuitive array access of structure member on line 30, represented as `network_data->connection_array[index]`. The redundant assignment statements on lines 13-18 in (a) are also streamlined into a clearer single statement in (b), as shown on line 35.



```
1   undefined8
2   network_register_fdevents(long network_context) {
3     // ...
4     for (event_index = 0; event_index < *(uint *)(network_context + 0x148);
5          event_index++) {
6       // Get the current file descriptor event
7       current_fd_event = *(long *)(*(long *)(network_context + 0x140)
8                          + (ulong)event_index * 8);
9
10      // Check if the file descriptor is valid
11      if (*(int *)(current_fd_event + 0x70) != -1) {
12        // Register the file descriptor event
13        registration_result = fdevent_register(
14                              *(undefined8 *)(network_context + 0x18),
15                              *(undefined4 *)(current_fd_event + 0x70),
16                              FUN_00118948, current_fd_event);
17        // Store the registration result
18        *(undefined8 *)(current_fd_event + 0x78) = registration_result;
19        // Set the event for the registered file descriptor
20        fdevent_fdnode_event_set(*(undefined8 *)(network_context + 0x18),
21                                 *(undefined8 *)(current_fd_event + 0x78), 1);
22    // ...
23  }
```

(a)

```
24  undefined8
25  network_register_fdevents(struct_3ffd *network_data) {
26    // ...
27    for (uint index = 0; index < network_data->connection_count;
28         index++) {
29      // Get the current struct_2442 pointer
30      struct_2442 *current_struct = network_data->connection_array[index];
31
32      // Check if fd is not -1
33      if (current_struct->fd != -1) {
34        // Register the file descriptor event and store the result
35        current_struct->event_node = fdevent_register(
36                                     network_data->event_context,
37                                     current_struct->fd,
38                                     FUN_00118948, current_struct);
39        // Set the event for the registered file descriptor node
40        fdevent_fdnode_event_set(network_data->event_context,
41                                 current_struct->event_node, 1);
42    // ...
43  }
```

(b)

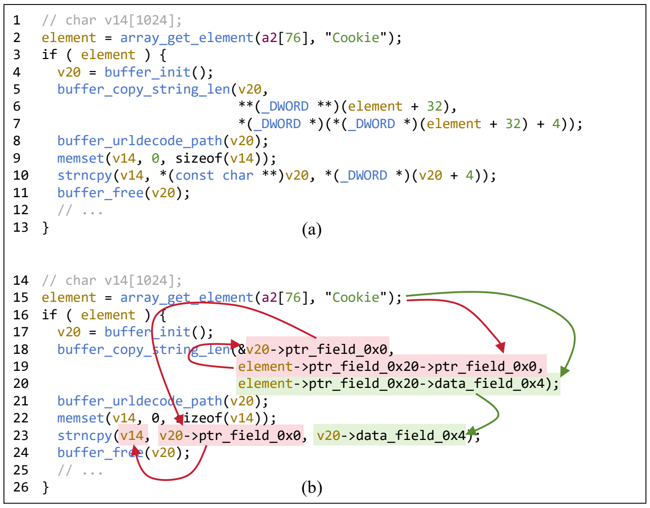
Figure 10: Comparison of DeGPT-optimized decompiled code before and after type recovery with TypeForge.

In summary, the composite data types recovered by TypeForge significantly enhance the readability of decompiled code, thereby improving the efficiency of reverse engineers in their analyses.

A.2. Vulnerability Detection

Another downstream task supported by TypeForge is static vulnerability detection. While numerous static vulnerability detection tools for source code have achieved high accuracy, their methods cannot be applied to stripped binaries or decompiled code due to the lack of composite data type information. Currently, there are also some static vulnerability detection approaches targeting stripped binaries [2], [3], [4]; however, these approaches typically operate at the instruction level and require extensive modeling of program memory, which significantly impacts their execution efficiency.

Figure 11 shows a recent real-world vulnerability from the same ASUS router firmware as an example. To preserve the anonymity of this submission, we do not include the full



```
1  // char v14[1024];
2  element = array_get_element(a2[76], "Cookie");
3  if ( element ) {
4    v20 = buffer_init();
5    buffer_copy_string_len(v20,
6                           **(_DWORD **)(element + 32),
7                           *(_DWORD *)(*(_DWORD *)(element + 32) + 4));
8    buffer_urldecode_path(v20);
9    memset(v14, 0, sizeof(v14));
10   strncpy(v14, *(const char **)v20, *(_DWORD *)(v20 + 4));
11   buffer_free(v20);
12   // ...
13 }
```

(a)

```
14 // char v14[1024];
15 element = array_get_element(a2[76], "Cookie");
16 if ( element ) {
17   v20 = buffer_init();
18   buffer_copy_string_len(&v20->ptr_field_0x0,
19                          element->ptr_field_0x20->ptr_field_0x0,
20                          element->ptr_field_0x20->data_field_0x4);
21   buffer_urldecode_path(v20);
22   memset(v14, 0, sizeof(v14));
23   strncpy(v14, v20->ptr_field_0x0, v20->data_field_0x4);
24   buffer_free(v20);
25   // ...
26 }
```

(b)

Figure 11: Decompiled code of a real-world vulnerability before and after type recovery with TypeForge.

CVE identifier. Notably, this vulnerability has already been reported to the vendor and patched, ensuring no issues are violating academic ethics.

Figure 11(a) shows the decompiled code of this vulnerability without type recovery. The function `array_get_element` stores user input from the "Cookie" into the member of a structure variable `element` (line 2). These members propagate as the program executes and eventually trigger a buffer overflow vulnerability in `strncpy` (line 10). However, when the composite data types in the stripped binary are not recovered, static analysis tools cannot determine that variable `element` is a structure pointer or understand the meaning of the memory access expressions. As a result, existing methods for analyzing stripped binaries often rely on complex memory modeling, significantly increasing performance overhead.

If the composite data types in this program are recovered, as shown in Figure 11(b), static analysis tools can easily trace the propagation path of the dangerous data. Specifically, the function `array_get_element` stores the content and length of the dangerous input "Cookie" into the structure members `element->ptr_field_0x20->ptr_field_0x0` (red arrow) and `element->ptr_field_0x20->data_field_0x4` (green arrow), respectively. Subsequently, the function `buffer_copy_string_len` propagates the content and length to the structure members `v20->ptr_field_0x0` and `v20->data_field_0x4`. At the `strncpy` call on line 23, the buffer overflow vulnerability can be detected if it is determined that both `v20->ptr_field_0x0` and `v20->data_field_0x4` are attacker-controlled.

In summary, the composite data types recovered by TypeForge greatly enhance the capabilities of existing static analysis tools for stripped binaries. By eliminating the need for overly complex memory modeling, these tools can adopt static analysis techniques initially designed for source code, thereby significantly improving their efficiency.

Appendix B.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

B.1. Summary

This paper focuses on recovering composite data types in stripped binaries. The authors propose a new graph structure to capture type information and synthesize from it possible type declarations. Then, they use an LLM-assisted double-elimination framework to select the best-fit declaration from the candidates according to the readability of the resulting decompiled code.

B.2. Scientific Contributions

- Creates a New Tool to Enable Future Science
- Addresses a Long-Known Issue
- Provides a Valuable Step Forward in an Established Field

B.3. Reasons for Acceptance

1) This paper creates a new tool to enable future science. The authors make their tool and the evaluation artifacts available to other researchers, allowing for independent confirmation and supporting future research.
2) This paper addresses a long-known issue. Binary code type inference has been studied for over 20 years (e.g., ACM CSUR 2016 survey from Caballero and Lin) and has numerous applications to software and systems security.
3) This paper provides a valuable step forward in an established field. While recent literature has improved inference accuracy with primitive types, this work advances the state of the art with composite types, proposing a graph abstraction that is new and may enable other downstream uses for the captured information.

B.4. Noteworthy Concerns

Compiler optimizations can negatively impact the accuracy of the method, leaving to future research the open question of whether—and how—their undesired effects can be mitigated in the current design.