

USENIX

THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

RecStruct: Recovering Nested Struct Types from Stripped Binaries via Stack-Driven Unification

Yuxin Chen, Zhiyang Fang, Shiyi Wu, Yixin Xu, and Yuhang Wang, *Sichuan University;* Xiaokang Yin, *Information Engineering University;* Junfeng Wang, *Sichuan University*

https://www.usenix.org/conference/usenixsecurity26/presentation/chen-yuxin

This paper is included in the Proceedings of the 35th USENIX Security Symposium.

August 12–14, 2026 • Baltimore, MD, USA

ISBN 978-1-939133-58-8

Open access to the Proceedings of the 35th USENIX Security Symposium is sponsored by



جامعة الملك عبدالله للعلوم والتقنية

King Abdullah University of Science and Technology

ARTIFACT EVALUATED
USENIX
AVAILABLE

RecStruct: Recovering Nested Struct Types from Stripped Binaries via Stack-Driven Unification

Yuxin Chen
Sichuan University

Zhiyang Fang[†]
Sichuan University

Shiyi Wu
Sichuan University

Yixin Xu
Sichuan University

Yuhang Wang
Sichuan University

Xiaokang Yin
Information Engineering University

Junfeng Wang
Sichuan University

Abstract

Recovering composite data types from stripped binaries is critical for reverse engineering and vulnerability analysis. Existing approaches either rely on heavyweight decompilers with limited type reasoning, or employ expensive constraint solvers that fail to scale. While some methods can handle nested structures, they struggle with large real-world binaries due to prohibitive computational costs. We present RecStruct, a fast type recovery framework that reconstructs nested structures without dependence on any decompiler. RecStruct lifts binaries directly, performs flow-insensitive symbolic execution to reduce memory aliases, and introduces Stack-Driven Unification—a novel union-find algorithm that maintains an offset stack during traversal to merge types only when their nesting depths align. This enables precise recovery of recursive structures. We implement an end-to-end pipeline and evaluate it on 418 binaries comprising 744,770 functions, including benchmarks from prior work and large-scale programs such as QEMU and OpenSSL. Experimental results demonstrate competitive accuracy compared to existing tools, with substantial improvements in efficiency and scalability, enabling analysis of binaries that were previously intractable.

1 Introduction

Recovering struct layouts from stripped binaries is a key challenge in binary reverse engineering, where high-level type information is irrevocably lost during compilation [3, 7, 10, 27]. *Struct layout* (the memory organization of composite data types) is particularly valuable for reverse engineers: accesses to composite types are nearly twice as frequent as those to primitive variables [43], and struct layouts directly reveal how programs organize and access complex data, which is essential for vulnerability discovery and malware analysis [11, 16, 30, 37, 39, 45]. Yet recovering them remains a prominent challenge [39, 45].

[†]Corresponding author: fangzhiyang@scu.edu.cn

The core difficulty is that field accesses are scattered across multiple functions [37, 44] and nested pointer relationships further complicate the analysis. We identify three key challenges (illustrated in Section 2): *cross-function aggregation* of partial field views, *nested struct discovery* through pointer dereferences, and *inner struct aggregation* where nested fields accessed in different functions must be recognized as belonging to the same dereferenced type.

Since recovering exact source-level types is fundamentally impossible [5], we target *memory layout* recovery (the offsets, sizes, and pointer-based nesting relationships of struct fields as realized in the binary) rather than source-level type reconstruction. Our correctness and completeness claims are scoped to this abstraction.

Existing approaches (spanning constraint-based analysis [3, 27, 45], data-flow methods [43], dynamic techniques [22, 39], and learning-based prediction [10, 37, 44]) have made significant progress on type recovery (see Section 6). However, *automated cross-function type aggregation* and *multi-level nested struct recovery* remain open challenges: existing methods either require manual intervention for cross-function propagation, or lack principled mechanisms for tracking pointer chains through nested struct hierarchies.

Our approach draws inspiration from how experienced reverse engineers manually recover struct types [42, 43]: they observe local *base* + *offset* patterns, trace data flow across function boundaries to collect field information, and integrate scattered observations into a global layout. This workflow reveals a fundamental property: *type equivalence propagates transitively across function calls*, but must respect field offsets to correctly handle nested structures. A pointer dereference yields a different (but related) type that must be tracked separately.

Based on these insights, we propose **RecStruct**, a practical system that automates this inductive workflow through two key technical innovations. First, rather than relying on heavy-weight whole-program pointer analysis, we employ *lightweight symbolic execution* to extract memory access patterns. By confining symbolic analysis to intra-procedural data

flow and tracking only base pointer propagation, we effectively avoid the path-explosion problem [2, 38], making our approach highly scalable.

Second, and most crucially, we reframe the cross-function type aggregation problem as a *graph reachability* problem rather than a constraint solving problem. While standard union-find algorithms can efficiently merge equivalent types, they fail to handle nested structures where a pointer field at offset O points to a completely different type. To address this, we introduce a novel *Stack-Driven Union-Find* algorithm. By maintaining an offset stack during graph traversal, our algorithm precisely tracks type equivalence through arbitrary levels of pointer indirection (e.g., `struct A` contains a pointer to `struct B`, which contains a pointer to `struct A`). This avoids the exponential blowup and makes the analysis tractable in practice.

In summary, this paper makes the following contributions:

- We formalize struct recovery as a *layout aggregation* problem on a **Type Distribution Graph (TDG)**, where each node represents a local struct view from a single function, enabling global layouts to emerge from local evidence without whole-program pointer analysis.
- We propose the *Stack-Driven Union-Find* algorithm, which extends the classic union-find data structure to support offset-sensitive equivalence. This allows us to handle recursive and deeply nested structures efficiently, guaranteeing termination and correctness under our layout abstraction.
- We implement RecStruct as a standalone Rust-based tool that operates directly on binary code. Evaluation on 7 real-world datasets demonstrates that RecStruct outperforms state-of-the-art tools by up to 4.9× in layout accuracy and 15× in nested structure recognition, while achieving 60–250× speedup. RecStruct also scales to large-scale binaries (e.g., QEMU, OpenSSL) where existing tools fail due to timeouts or memory exhaustion.

2 Motivation and Problem Formalization

2.1 Motivation Example

To illustrate our approach, we use the running example shown in Figure 1. The program defines two structures: `Outer` (containing a pointer `p` to `Point` and two integers `i1`, `i2`) and `Point` (containing `x`, `y`, `z`). The challenge lies in the fact that function `Func1` accesses only `p->x`, `p->y`, and `i1`, while `Func2` accesses `p->z` and `i2`. Recovering the complete definitions requires aggregating partial views from disjoint functions and correctly identifying the nested relationship between `Outer` and `Point`.

```
struct Point { int x, y, z; };
struct Outer { struct Point *p; int i1, i2; };
void Func1(struct Outer *a) {
    a->p->x = 1; a->p->y = 2; a->i1 = 1;
}
void Func2(struct Outer *b) {
    b->p->z = 3; b->i2 = 2;
}
int main() { /* ... */ Func1(v1); Func2(v1); }
```

Figure 1: Running example exhibiting partial structure views and nested pointers. Functions `Func1` and `Func2` access disjoint fields of `Outer` and `Point`, requiring cross-function aggregation.

2.2 Limitations of Existing Approaches

Existing approaches struggle with the above example in different ways. Industrial tools such as IDA Pro [35] and Ghidra [1] can infer partial layouts within each function, e.g., `{0: ptr(8), 8: int(4)}` from `Func1` and `{0: ptr(8), 12: int(4)}` from `Func2`, but merging them into the complete `Outer` definition requires *manual* cross-function reasoning. Constraint-based methods [3,45] automate global solving, yet face scalability issues and struggle with nested structures: distinguishing `Outer` from `Point` while linking them through the pointer at offset 0 requires tracking dereference chains that flat constraint systems do not natively support. Data-flow approaches such as TypeForge [43] model each *expression* as a graph node, enabling inter-procedural propagation but causing graph explosion in real-world binaries. Learning-based methods [10, 37, 44] are bounded by their training vocabulary and cannot generalize to unseen struct types; LLM-based variants further incur high per-function costs and non-deterministic outputs.

These limitations motivate our design: lightweight symbolic execution to avoid the scalability bottleneck of Value Set Analysis (VSA), *layout*-granularity graph nodes to control graph size, and a Stack-Driven Union-Find algorithm to precisely handle nested structures. All these are achieved without training data or manual intervention.

2.3 Problem Formalization

Formally, let B be a stripped binary consisting of functions $F = \{f_1, \ldots, f_n\}$. We assume that these functions operate on memory objects which are instances of a set of underlying types $\mathcal{T}$. Our objective is to reconstruct $\mathcal{T}$ based on the memory access patterns observed in F. The set $\mathcal{T}$ encompasses both primitive types and structure types, where a structure type $T \in \mathcal{T}$ is defined by a collection of fields $\{(offset, size, \tau)\}$ with $\tau \in \mathcal{T}$. We address this challenge by decomposing the recovery process into two phases: (1) **Layout Collection**: We map each function f to a set of *Local Layouts* $\mathcal{L}_f$. Each layout $l \in \mathcal{L}_f$ aggregates a set of field ac-

cess constraints observed on a specific memory object within f, serving as a partial hypothesis of the underlying type. (2) **Layout Aggregation**: We construct an equivalence relation $\sim$ on the universe of layouts $\mathbb{L} = \bigcup_f \mathcal{L}_f$. This step determines the *mergeability* of partial views: l_i and l_j are merged ($l_i \sim l_j$) if they are linked by inter-procedural data flow or satisfy structural consistency, and must be distinguished otherwise (e.g., due to conflicting field offsets or sizes).

3 Approach

In this paper, we propose RecStruct, a framework designed to recover nested structure types from stripped binaries. Unlike traditional graph-based recovery methods [43, 45], RecStruct innovatively utilizes *local structure layouts* as fundamental graph nodes to significantly reduce complexity. We formalize the type recovery problem as a two-stage process: *local field collection* followed by *global layout aggregation*. This decomposition enables the use of lightweight, flow-insensitive symbolic execution for scalable field extraction, coupled with a novel stack-driven graph traversal algorithm to handle the intricacies of nested and recursive data structures.

3.1 System Overview

The RecStruct pipeline, illustrated in Figure 2, operates in four stages that progressively raise the abstraction level from binary instructions to high-level C definitions.

- **Semantic Extraction (§3.2).** we employ **DAG-based symbolic execution** for semantic extraction. We observe that structure field accesses are largely path-insensitive: the set of fields a function accesses rarely depends on complex control flow conditions. This allows us to use a scalable, flow-insensitive analysis that avoids the path explosion problem inherent in traditional symbolic execution.
- **TDG Construction (§3.3).** we model type relationships using a TDG. Type equivalence is an equivalence relation, which is naturally represented by connected components in a graph. However, the presence of pointer dereferences introduces asymmetric relationships that simple connectivity cannot capture. The TDG explicitly distinguishes between *reference edges* (representing type equivalence) and *dereference edges* (representing pointer relationships).
- **Stack-Driven Type Propagation (§3.4).** We introduce **Stack-Driven Union-Find** to solve the nested structure recovery problem. A naive graph traversal would conflate a container structure with its fields if not carefully constrained. By maintaining an *offset stack* during traversal, we track the current "nesting depth" of pointers, ensuring that we only unify layouts that exist at the same structural level.
- **Type Synthesis (§3.5).** The final stage synthesizes concrete C struct definitions from the identified type groups. A key challenge is *conflict resolution*: different functions may access the same memory offset with inconsistent sizes or interpretations. We address this with a **priority-based merging** strategy that ranks field observations by their semantic confidence: actual data accesses outweigh pointer dereferences, which in turn outweigh reference-only edges.

3.2 Stage 1: Semantic Extraction

The first stage transforms raw binary instructions into *Semantic Logs* (SemaLogs), a structured intermediate representation (IR) that captures type-relevant memory operations. We introduce this intermediate layer not merely because assembly is low-level, but because raw instruction streams and standard IRs contain excessive redundancy and complex memory aliasing information that obscures structural relationships. SemaLog distills these into a concise form focused solely on data flow and memory layout.

3.2.1 Lightweight Symbolic Execution

To generate SemaLogs, we employ a lightweight, flow-insensitive symbolic execution engine operating on the IR. Unlike traditional symbolic execution that aims for path completeness and constraint solving, our approach focuses on efficient value propagation.

State and Execution Model. The execution state consists of a register file and a symbolic memory map, both mapping locations to symbolic expressions. We construct a Directed Acyclic Graph (DAG) from the function's Control Flow Graph (CFG) by ignoring back-edges, which avoids path explosion. The execution proceeds in a topological order based on the DAG. Dropping back-edges means that loop bodies are analyzed only once rather than iterated to a fixed point. This is sound for our purpose because structure field accesses are almost always loop-invariant: a loop that dereferences `p->field` on every iteration still reveals the same layout information as a single iteration. The primary effect of discarding back-edges is to prevent repeated propagation of identical symbolic values around loop headers, which would add cost without improving field discovery. In the rare cases where a loop introduces a new base pointer on each iteration (e.g., linked-list traversal `p = p->next`), the single-iteration analysis still captures the dereference at the relevant offset, sufficing to establish the pointer field in the recovered layout.

State Merging and Conflict Resolution. A key feature is our aggressive state merging strategy. At control flow join points, we merge incoming states. If a register or memory location holds different symbolic values from different predecessor paths, we introduce a *fresh symbolic variable* to represent the convergence. Unlike SSA-based Phi functions that preserve path conditions, this approach discards path constraints to prioritize scalability over precision, allowing us to resolve conflicts without expensive Satisfiability Modulo Theories (SMT) solving. To further ensure scalability, we

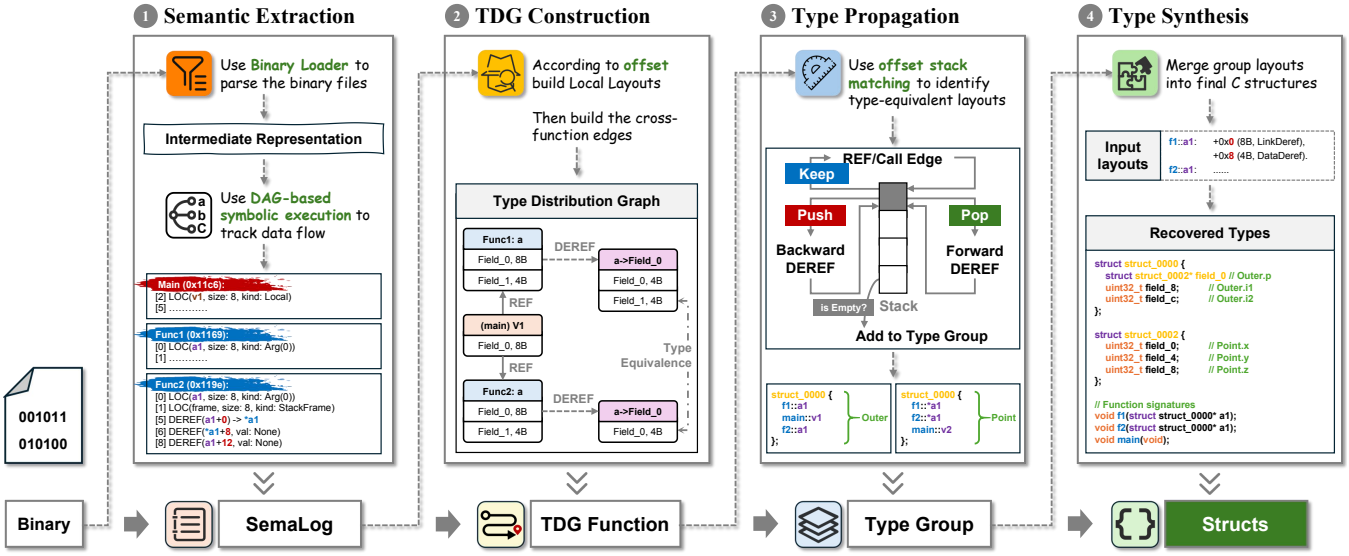


Figure 2: RecStruct's four-stage pipeline for struct recovery. Stage 1 extracts SemaLogs via DAG-based symbolic execution. Stage 2 constructs per-function type layouts and builds the TDG. Stage 3 applies Stack-Driven Union-Find to identify type-equivalent layouts across function boundaries. Stage 4 merges layouts and synthesizes C struct declarations.

limit the depth of symbolic expressions (default 8 levels), concretizing expressions that exceed this complexity threshold. Since structure field accesses and memory addressing calculations are typically linear, the engine is limited to algebraic simplification and value tracking, achieving high scalability while retaining sufficient precision for reconstructing complex pointer chains.

3.2.2 SemaLog Semantics

The analysis produces a sequence of SemaLogs, categorized into five unified types: **Loc** establishes symbolic bases for memory regions such as function arguments and stack frames; **Deref** records memory accesses at specific offsets, uniformly handling both data reads and pointer dereferences; **Call** captures function invocations and argument passing; **Ref** tracks value aliasing and data flow continuity.

Notation Note: In our traces, `DEREF(addr) -> val` denotes a load operation producing symbolic value `val`, while `DEREF(addr, val: V)` denotes a store operation writing value `V` to `addr`.

In our running example, Figure 2 (Stage 1) illustrates the SemaLogs generated for `Func2`, which accesses `b->p->z` and `b->i2`. Logs (0) and (1) declare base locations: the function argument `a1` and the stack frame. Log (5) is a Link Dereference reading `a1+0` with size 8 (pointer width), producing `*a1`, which corresponds to loading the pointer `b->p`. Log (6) writes to `*a1+8` with size 4, corresponding to `b->p->z` (offset 8 within `Point`). Log (8) writes to `a1+12` with size 4, corresponding to `b->i2`. Logs (5) and (8) both access the base symbol `a1`. These accesses will be aggregated to form the local layout for `a1`. We evaluate this running example in detail in Section 4.2, and provide a comprehensive end-to-end walkthrough, covering all SemaLog entries, dereference classification, and resulting local layouts, in Appendix A.3.

3.3 Stage 2: Type Distribution Graph Construction

The second stage constructs the TDG, which serves as the global substrate for our type inference algorithms. While semantic extraction provides a local view of each function, the TDG integrates these views into a cohesive whole, enabling us to reason about type relationships across the entire binary.

3.3.1 Graph Definition

Formally, we define the TDG as a directed multigraph $G = (V, E)$. The vertex set V consists of *Local Structure Layouts*, where each layout represents a specific symbolic variable within a specific function (e.g., "argument 0 of Func1"). We treat primitive types as a special case of local layout, and stack frames as a special memory layout. The edge set E contains two distinct types of edges. *Reference edges* (E_{ref}) connect layouts that are structurally equivalent; for example, passing a variable as an argument to a function creates a reference edge from the caller's variable layout to the callee's argument layout. *Dereference edges* (E_{deref}) represent pointer relationships: a dereference edge labeled with offset o connects a

layout L_1 to L_2 if L_2 is the result of dereferencing L_1 at offset o.

3.3.2 Construction Process

RecStruct builds the TDG in three phases. First, it pre-scans to identify active variables. Second, it iterates through all SemaLogs to create vertices. For every LOC log, a corresponding Local Layout node is instantiated.

During this process, we distinguish between *Link Dereferences* (LinkDref) and *Data Dereferences* (DataDref) derived from SemaLog's generic DEREF. A LinkDref represents a pointer access that leads to another structure node, while a DataDref represents access to a primitive value. We differentiate them based on two strict criteria: (1) the dereferenced value must be symbolic (i.e., it has a corresponding layout node), and (2) the access size must match the machine pointer width. If either condition is not met, the access is classified as a DataDref.

DEREF logs contribute fields to these layouts; if the log represents a Link Dereference, it also prompts the creation of a target layout node if one does not yet exist.

In the third phase, RecStruct instantiates edges. CALL logs generate Reference edges connecting call sites to function definitions. Link Dereferences generate Dereference edges connecting container structures to their members. Figure 2 (Stage 2) visualizes the TDG for our running example. The local layouts for `Func1::a` and `Func2::a` are distinct nodes, but both possess Dereference edges at offset 0 pointing to their respective `Point` layouts. Crucially, the `main` function connects to both `Func1` and `Func2` via Reference edges, effectively "gluing" the graph together and enabling the propagation of type information between the disjoint functions.

3.4 Stage 3: Stack-Driven Type Propagation

This stage identifies which Local Layouts in the TDG represent the same underlying type. A naive approach might simply find connected components in the graph, but this is insufficient for nested structures. In our example, `Outer` contains a pointer to `Point`. These two types are connected in the graph via a Dereference edge, but they are clearly distinct types. Simple connectivity would erroneously merge them.

3.4.1 Offset Stack Semantics

To solve this, we introduce the concept of an *offset stack* to track the "nesting level" during graph traversal. The stack represents the sequence of pointer offsets traversed from a starting node. An empty stack implies the current node is at the same structural level as the start node (i.e., they are type-equivalent). A stack containing $[0,4]$ implies we are currently investigating a type reachable by dereferencing offset 0 of the start type, and then offset 4 of the resulting type.

Algorithm 1 Stack-Driven Type Grouping

Require: Starting layout L_{start}, TDG $G = (V, E)$
Ensure: Set of type-equivalent layouts

worklist $\leftarrow \{([], L_{start})\}$
group $\leftarrow \emptyset$
while worklist $\neq \emptyset$ **do**
  $(S, L) \leftarrow$ worklist.pop()
  if $(S, L) \in$ visited **then continue**
  visited.add((S, L))
  if S is empty $\wedge$ L is not StackFrame **then**
    group $\leftarrow$ group $\cup \{L\}$
  for each incoming edge (L_{src}, o, L) type T **do**
    if $T =$ DEREF $\vee$ $(T =$ REF $\wedge\, o \neq 0)$ **then**
      worklist.add($(S \cdot [o], L_{src})$)
    if $T =$ REF $\wedge\, o = 0$ **then**
      worklist.add((S, L_{src}))
  for each outgoing edge (L, o, L_{dst}) type T **do**
    if $T =$ REF $\wedge\, o = 0$ **then**
      worklist.add((S, L_{dst}))
    if S.top() == o **then**
      worklist.add($(S$.pop(), $L_{dst})$)
return group

We define type equivalence based on reachability with an empty stack. Two layouts L_1 and L_2 are equivalent if there exists a valid path between them such that the net change to the offset stack is zero. This enforces the *DEREF Consistency Rule*: if two container types are equivalent, their fields at the same offset must also be equivalent.

3.4.2 Stack-Driven Union-Find Algorithm

We implement this logic using a modified union-find algorithm, detailed in Algorithm 1. The algorithm maintains a worklist of `(stack, layout)` pairs. It starts exploring from a node with an empty stack. When traversing a **Dereference edge** with offset o in the forward direction (container $\rightarrow$ member), we push o from the stack. Conversely, traversing a Dereference edge backward (member $\rightarrow$ container) pops o from the stack, but only if the top of the stack matches o. **Reference edges** are traversed freely without modifying the stack, as they represent type identity.

This algorithm offers a scalable approximation for unifying nested structures, prioritizing practical effectiveness over theoretical completeness (recovering all relations expressible within our abstraction). By enforcing stack consistency during graph traversal, it robustly mitigates the risk of merging a structure with one of its constituent fields, a frequent source of imprecision in alias analysis. The algorithm naturally handles recursive data structures through its visited-state tracking, which prevents infinite loops while still allowing type information to propagate through deep pointer chains.

This allows RecStruct to correctly distinguish and reconstruct complex, self-referential data types that simpler connectivity-based methods would often conflate.

3.5 Stage 4: Type Synthesis

The final stage synthesizes concrete C structure definitions from the identified type groups. Each group contains a collection of Local Layouts that have been determined to represent the same type. The synthesis process must merge these layouts into a single, consistent definition.

Conflict resolution is a primary challenge here. Different functions may access the same memory offset in inconsistent ways due to compiler optimizations or limited visibility (e.g., one function reads a byte, another reads a full word). RecStruct resolves these conflicts using a priority-based merging strategy. We assign a priority score to each field observation: *Data Dereferences* (actual memory reads) are given the highest priority (10^5 weight). *Link Dereferences* (pointer dereferences) have medium priority (10^3), and properties inferred from *Reference edges* have the lowest priority (10^1).

The merging algorithm sorts all candidate fields by priority and greedily selects non-overlapping fields. Crucially, when a higher-priority Data Dereference wins over a Link Dereference at the same offset (e.g., recognizing an `int` field instead of a pointer), we preserve the pointer type information from the lower-priority candidate if available. This ensures we do not lose structural information (like "points to struct X") even when the concrete access pattern dominates. Finally, we simplify single-field structs at offset 0 into primitive types (degenerate types) unless they are self-referential, further cleaning up the output.

3.6 Implementation

We implement RecStruct as a standalone binary analysis framework in Rust, comprising approximately 7,900 lines of code. A key design decision is that our tool operates independently of existing reverse engineering platforms such as IDA Pro or Ghidra. Instead, we develop our own binary lifting infrastructure that directly parses ELF binaries and disassembles machine code into an intermediate representation suitable for symbolic analysis. This self-contained architecture eliminates external dependencies, simplifies deployment, and enables tighter integration between the disassembly, symbolic execution, and type inference components. Moreover, RecStruct exports recovered type information in standard formats (e.g., C header files), allowing it to be seamlessly integrated into existing reverse engineering workflows and toolchains such as IDA Pro and Ghidra.

Methodologically, RecStruct requires only two inputs: the binary itself (in ELF format) and function boundaries. Function boundary identification is orthogonal to our contribution and can be supplied by any existing tool (e.g., IDA, Ghidra, or signature-based heuristics); The symbolic analysis operates on a low-level P-code-style intermediate representation that we lift directly from machine code, without invoking any external decompiler or its high-level output (e.g., decompiled C, inferred variables).

4 Evaluation

We evaluate RecStruct on 7 real-world datasets to demonstrate its effectiveness in recovering structure layouts and compare it against state-of-the-art (SoTA) methods TypeForge [43], Trex [5], and ReSym [44]. Our evaluation addresses five research questions:

- **RQ1:** Can RecStruct effectively recover complex structures and nested relationships?
- **RQ2:** How accurate is RecStruct compared to SoTA methods?
- **RQ3:** How efficient is RecStruct compared to other tools?
- **RQ4:** Can RecStruct scale to large-scale binaries?
- **RQ5:** What is the contribution of each component to the overall performance?

4.1 Experiment Setup

Datasets. We evaluate on 7 datasets covering diverse program scales and complexities, categorized by average functions per binary:

Table 1: Benchmark datasets categorized by average functions per binary.

Scale	Dataset	Description
Small	coreutils-O3	GNU coreutils, -O3
	coreutils-O2	GNU coreutils, -O2
	coreutils-O1	GNU coreutils, -O1
Medium	utils	grep, gzip, lighttpd, wget
	binutils	GNU binary tools
Large	qemu	User and system emulators
	openssl	Cryptographic library

Baselines. We compare RecStruct against three SoTA tools representing distinct technical approaches: **Trex** [5], representing the constraint-based program analysis approach; **TypeForge** [43], representing the graph-based data flow analysis approach; and **ReSym** [44], representing the Large Language Model (LLM) based approach.

Metrics. We assess performance using two sets of metrics: *Primary Metrics (2 Tasks):*

- **Task 1 - Layout Accuracy:** For correctly identified structure pointers, we evaluate the accuracy of the recovered memory layout. Since stripped binaries lack meaningful field names, and type names may vary, we represent each structure as a set of $(offset, size)$ tuples corresponding to its fields. We calculate the intersection between the ground truth field set and the predicted field set. Precision is defined as the ratio of matched fields to total predicted fields, and Recall as the ratio of matched fields to total ground truth fields. This metric strictly enforces structural correctness, ensuring that fields are identified at the correct offsets with the correct sizes.

- **Task 2 - Structure Nesting Recognition:** This task evaluates the recovery of inter-structure relationships, specifically pointer fields linking structures. We employ a path-based method: starting from each structure pointer parameter, we use DFS to generate nesting paths, where each step is a tuple $(offset, pointer_level)$ representing a pointer field's position and indirection level. DFS is bounded to depth 4 and terminates upon encountering cycles to ensure tractability. We compute Precision, Recall, and F1 by comparing path sets between ground truth and predictions.

Hierarchical Scoring (Trex-style): To provide a granular comparison with Trex, we also report hierarchical property scores: IsDefined → IsCPointer → CPointerLevel → IsCStruct → SignIgnoredCPrimitive → CPrimitive. Notably, this metric does not quantify the accuracy of structure field layouts or inter-structure relationships: it only checks whether the type *is* a struct, not whether its internal layout is correct.

Ground Truth. We extract ground truth from DWARF debug information embedded in unstripped binaries. As noted by prior work [5], recovering exact source-level type definitions from binaries is fundamentally impossible due to information loss during compilation. Raw DWARF describes source-level types, which may diverge from the actual binary memory layout due to compiler transformations (e.g., anonymous struct expansion, typedef resolution, padding insertion). We process DWARF information to normalize these source-level artifacts into binary-layout-aligned type definitions, producing ground truth that reflects the practical goal of reverse engineering. We exclude functions with incomplete DWARF information or those absent in the stripped binary (e.g., inlined functions). To ensure reproducibility, we release both pre-extracted ground truth files in YAML format and extraction scripts.

Evaluation Scope. We focus our evaluation on function parameters to ensure a fair comparison. Prior works receive significant variable-level information for free: Trex [5] takes ground-truth variable locations and sizes as input, and TypeForge [43] operates on Ghidra's decompiler-inferred variables. In contrast, RecStruct is *decompiler-agnostic*: it works directly on a low-level intermediate representation (P-code) lifted from machine code, without relying on ground-truth variable boundaries or decompiler-provided abstractions. Evaluating local variables under these asymmetric conditions would conflate *variable identification* with *type recovery*, unfairly penalizing RecStruct for lacking free variable information. Function parameters, identified through standardized calling conventions, provide a decompiler-independent anchor for fair alignment. Moreover, since cross-function structural information (the primary target of aggregate type recovery) must propagate through parameters, they are the semantically meaningful evaluation point for inter-procedural type inference. An empirical analysis confirms that function parameters cover 82–87% of all actively used structure types; the remaining stack-only types are exclusively anonymous nested types or function-local helpers that never cross function boundaries (Appendix A.1).

Environment. All experiments run on a server with 32 vCPUs and 46GB RAM running Ubuntu 24.04, using Python 3.12.3 and Rust 1.93.0.

4.2 RQ1: Qualitative Analysis of Structure Recovery

We conducted a qualitative case study using `old-case`, a custom micro-benchmark exhibiting multi-level pointer nesting (e.g., `A` → `B` → `C`), cyclic dependencies (`D` → `A`), and diverse field types. The ground truth comprises four interdependent structures (`A`, `B`, `C`, `D`) with 19 fields total. RecStruct recovered 100% of the fields and all five pointer relationships, including the cyclic reference `D` → `A`. Figure 3 compares RecStruct's output against the ground truth, demonstrating high-fidelity recovery of both field layouts and nested pointer types.

We further established a comparative baseline using IDA Pro 9.2, representing the state of the art in interactive binary analysis. We simulated an expert analyst's workflow: the binary was first decompiled to C pseudocode, followed by invoking IDA's "Create new structure type" functionality on key variables to infer layout definitions. This manual, semi-automated approach contrasts with RecStruct's fully automated pipeline, where the binary is processed directly without human intervention.

Table 2 presents the per-field recovery results on the `old-case` micro-benchmark. A field is marked as ✓ if the tool correctly identified both its offset and size; ✗ indicates the field was either missed entirely or only recovered as a raw offset without proper typing. RecStruct achieved 100% field recovery (19/19 fields), while IDA Pro missed approximately half (10/19). The failures predominantly occurred in fields accessed via nested pointers (e.g., `b2`, `c`, `d` in Struct B; `c1`–`c4` in Struct C), where IDA's function-local scope prevented type constraint propagation across function boundaries.

A critical distinction emerged in recovering deep nested references. The test case features an access chain `a->b->c->c1`, traversing three structure layers. RecStruct correctly propagated type constraints through this chain, identifying interme-

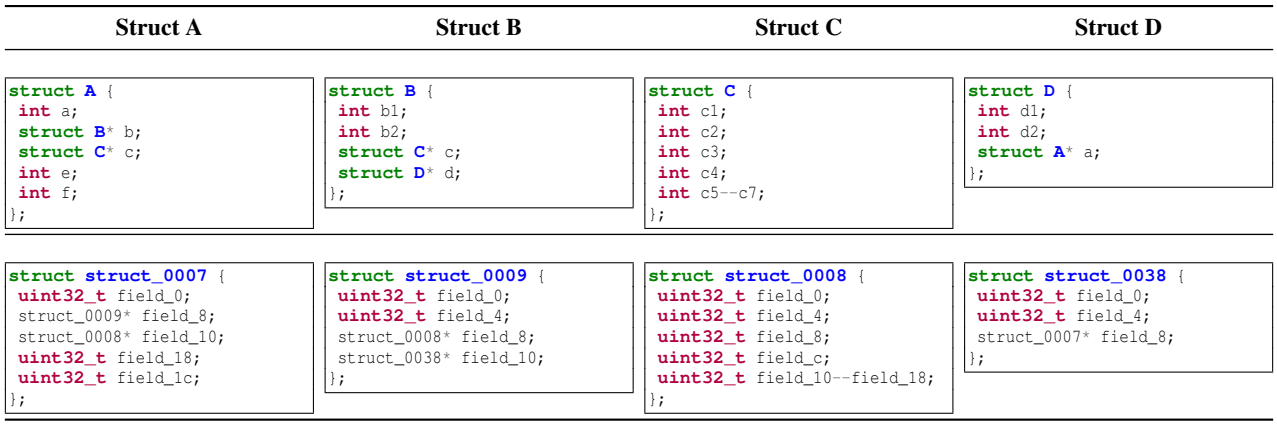


Figure 3: Comparison of Ground Truth (top row) and RecStruct's recovered definitions (bottom row). RecStruct correctly infers field layouts and pointer types: `struct_0007` matches `struct A`, with `field_8` correctly typed as `struct_0009*` (i.e., `struct B*`). The cyclic reference D → A is preserved as `struct_0038.field_8` → `struct_0007`.

diate fields as pointers to their respective types (e.g., `field_8` of `struct_0007` typed as `struct_0009*`). IDA frequently failed to resolve these, reducing semantic pointer accesses to raw arithmetic offsets (e.g., `**(v1 + 8) + 8`), obscuring the program's logical topology. Due to IDA's function-local scope, it also inferred disjoint structure definitions for the same runtime type across different functions (e.g., identifying `struct A` as distinct types `struct_a1`, `struct_a1_1`, etc.), necessitating significant manual reconciliation.

Table 2: Structure Field Recovery Status (✓: Correctly Identified, ✗: Missed/Offset Only)

Structure	Field (Offset)	RecStruct	IDA Pro
Struct A	a (0x00)	✓	✓
	b (0x08)	✓	✓
	c (0x10)	✓	✓
	e (0x18)	✓	✓
	f (0x1C)	✓	✓
Struct B	b1 (0x00)	✓	✓
	b2 (0x04)	✓	✗
	c (0x08)	✓	✗
	d (0x10)	✓	✗
Struct C	c1–c4 (Indirect)	✓	✗
	c5–c7 (Direct)	✓	✓
Struct D	d1 (0x00)	✓	✓
	d2 (0x04)	✓	✗
	a (0x08)	✓	✓

4.3 RQ2: Accuracy Comparison

We evaluate RecStruct's accuracy against TypeForge, Trex, and ReSym across the two primary tasks: Layout Accuracy (Task 1) and Nesting Recognition (Task 2). Tables 3 and 4 present the detailed Precision, Recall, and F1 scores for all tools on representative datasets.

Table 3: Task 1 - Layout Accuracy (P/R/F1, %) on representative datasets. Best F1 per dataset in **bold**.

Dataset	RecStruct			TypeForge			Trex			ReSym		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
coreutils-O1	49.94	36.54	**42.21**	80.38	10.98	19.32	39.68	5.08	9.01	36.90	4.46	7.95
coreutils-O2	25.90	19.21	**22.06**	78.07	11.76	20.44	35.47	4.36	7.76	48.53	9.48	15.86
coreutils-O3	24.48	18.59	21.13	76.57	12.56	**21.59**	34.98	4.15	7.42	50.00	8.38	14.36
utils	10.55	53.91	17.64	63.43	12.36	**20.68**	44.41	3.21	5.98	35.24	11.63	17.49

Table 4: Task 2 - Structure Nesting Recognition (P/R/F1, %) on representative datasets. Best F1 per dataset in **bold**.

Dataset	RecStruct			TypeForge			Trex			ReSym		
	P	R	F1	P	R	F1	P	R	F1	P	R	F1
coreutils-O1	49.01	10.12	**16.77**	84.11	3.37	6.48	89.20	0.94	1.87	87.80	0.22	0.43
coreutils-O2	38.84	16.15	**22.81**	93.67	3.03	5.88	92.94	0.58	1.15	93.24	1.92	3.76
coreutils-O3	32.29	16.76	**22.06**	91.81	3.20	6.19	93.02	0.59	1.16	85.13	1.68	3.29
utils	3.37	11.69	**5.23**	41.39	1.49	2.87	60.00	0.15	0.30	15.75	1.05	1.97

For Layout Accuracy (Task 1), RecStruct achieves the highest F1 on the coreutils-O1 and coreutils-O2 datasets, with the O1 score approximately 2.2× higher than TypeForge and up to 4.7× higher than Trex. On coreutils-O3 and utils, TypeForge achieves the best F1. ReSym achieves moderate performance on coreutils-O2 (15.86% F1) but exhibits inconsistent results across other optimization levels.

For Nesting Recognition (Task 2), RecStruct achieves the highest F1 across all datasets, with 1.8–3.9× higher scores than TypeForge and up to 19.8× higher than Trex. The large gap over Trex is expected, as Trex does not model nesting relationships in its design. The baselines achieve high precision but very low recall on this task, indicating they detect few nesting relationships overall. ReSym shows variable performance, achieving 3.76% F1 on coreutils-O2 but lower scores on other optimization levels.

We also report performance using the Trex-style hierarchical scoring introduced in Section 4.1, as shown in Table 5.

Table 5: Trex-style hierarchical scoring (Average Score out of 6.0) across datasets.

Dataset	RecStruct	TypeForge	Trex	ReSym
coreutils-O1	2.16	3.32	2.73	**3.64**
coreutils-O2	2.17	3.35	2.91	**3.89**
coreutils-O3	2.18	3.35	2.91	**3.82**
utils	3.61	2.97	2.79	**3.78**

Under this metric, ReSym achieves the highest scores, largely because it operates on decompiled pseudocode that already contains the decompiler's inferred type information. This metric heavily weights primitive type correctness and only checks whether the type *is* a struct without validating field offsets or sizes, explaining the contrast with Tables 3 and 4 where RecStruct leads when structural fidelity is properly evaluated.

We further analyze the robustness of these tools against compiler optimizations in Figure 4. RecStruct demonstrates balanced performance across optimization levels, maintaining its lead over all baselines in both tasks. The relatively stable performance suggests that our approach is robust to the structural transformations introduced by compiler optimizations.

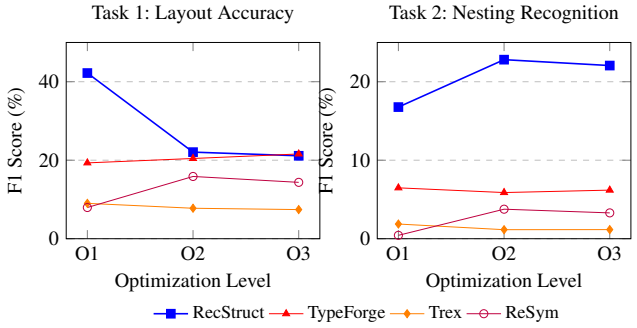


Figure 4: Impact of optimization levels (coreutils) on performance. Comparison of RecStruct against baselines.

4.4 RQ3: Time Efficiency

We compare the time efficiency of RecStruct against baselines by measuring end-to-end wall-clock time from input to output. For fairness, we measure baseline tools (TypeForge, Trex) including their Ghidra initialization overhead, as RecStruct is similarly measured on its complete pipeline. For ReSym, we exclude the one-time model loading latency to focus on per-binary inference time. Table 6 and Figure 5 present the results.

The results demonstrate a dramatic efficiency advantage for RecStruct, which achieves speedups of **62–76**× over TypeForge, **93–135**× over Trex, and **71–255**× over ReSym. This

Table 6: Runtime comparison across tools and datasets.

Dataset	RecStruct	TypeForge	Trex	ReSym
Coreutils-O1	1.0m	1.3h	2.3h	1.2h
Utils	9.5s	9.8m	14.7m	40.5m

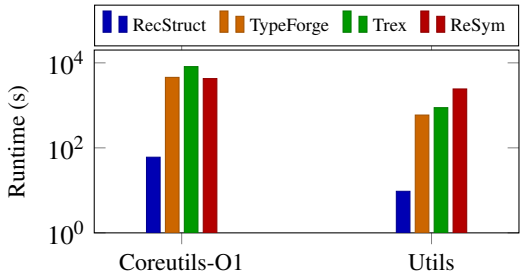


Figure 5: Total runtime comparison (log scale). Lower is better.

performance advantage stems from two complementary factors. First, RecStruct operates directly on low-level P-code lifted from the binary, whereas baseline tools incur substantial overhead from Ghidra's decompilation pipeline: both TypeForge and Trex require Ghidra to export decompiled output, variable information, and type annotations, a process that dominates their total runtime. Second, RecStruct's algorithmic design favors efficiency: our union-find-based constraint propagation runs in amortized $O(\alpha(n))$ time, while Trex employs iterative constraint solving across its six-stage analysis pipeline, and ReSym requires repeated LLM inference passes over each function. This efficiency is critical for enabling practical, interactive analysis of large-scale software systems.

4.5 RQ4: Scalability Analysis

Scalability is a prerequisite for practical binary analysis in real-world settings. To assess the scalability of RecStruct and the baselines, we evaluated them on three large-scale datasets: `binutils`, `qemu`, and `openssl`, employing a strict 2-hour timeout per binary on the hardware environment described in Section 4.1.

The experiment revealed a sharp contrast in capability. RecStruct successfully completed the analysis for all binaries in `binutils` and `openssl`, and all but one in `qemu` (with `qemu-system-aarch64` timing out). Conversely, both Trex and TypeForge failed to complete analysis for any of these datasets, succumbing to either the 2-hour timeout or Out-Of-Memory (OOM) errors. This result underscores the limitations of heavyweight constraint solving techniques when applied to binaries with complex inter-dependencies.

Table 7 details RecStruct's performance on these datasets. The absolute F1 scores are lower than on smaller benchmarks; our error analysis (§4.6) attributes this primarily to pervasive `void*` usage in these codebases, which creates type-erasing

bridges that induce over-unification, not binary size per se. Notably, partitioned analysis often yields better precision (Figure 7), consistent with practical workflows where analysts focus on a module or subsystem. These results demonstrate RecStruct's *scalability* to large binaries while highlighting that whole-program precision remains an open challenge. For example, on the `qemu` dataset, which contains over 843,000 structure parameters, RecStruct successfully recovered 2.58 million fields and identified over 415,000 valid nesting paths. The Trex-style scores, ranging from 2.75 to 3.45 out of 6.0, further indicate that RecStruct maintains a reasonable baseline of type property agreement even at this scale.

Table 7: RecStruct Performance on Large-scale Datasets. Task 1/2 F1 in %, Trex-style score out of 6.0. Baselines (TypeForge, Trex) failed to complete within 2-hour timeout.

Dataset	Task 1 F1	Task 2 F1	Trex-style
openssl	4.84	2.38	3.32
qemu	3.26	2.03	2.75
binutils	15.00	2.45	3.45

We further analyze the timing characteristics of these large-scale runs in Table 8. Despite the immense complexity of the target binaries (spanning up to 15,000 functions per binary), RecStruct completes analysis within practical time frames, demonstrating that the approach scales effectively to large-scale systems.

Table 8: RecStruct timing analysis on large-scale datasets.

Dataset	Bins	Avg Time	Total Time	Avg Funcs
openssl	1	1,312s	0.36h	15,316
qemu	61	1,173s	19.9h	11,652
binutils	27	260s	1.95h	2,560

4.6 Error Analysis on Large-Scale Binaries

While RecStruct demonstrates strong scalability (RQ4), the relatively low F1 scores on large-scale datasets reveal important limitations. On QEMU, our analysis achieves only 3.26% F1 for layout recovery and 2.03% for nesting recognition. We acknowledge that these aggregate metrics indicate substantial room for improvement, and provide a detailed error analysis to understand the underlying causes.

The primary source of precision loss is *type over-unification*. Large-scale systems like QEMU extensively use generic functions with `void*` parameters, which serve as type-erasing bridges that connect otherwise unrelated structures. Our flow-insensitive unification propagates constraints through these generic interfaces, merging distinct types into oversized "super-structs." For instance, structures like `CPUState` (15,056 bytes, 60+ fields) and `ARMCPU` (95,200 bytes) were collapsed together, resulting in a single unified type (`struct_16507`) assigned to 5,120 parameters with 142 fields spanning 3,758 bytes.

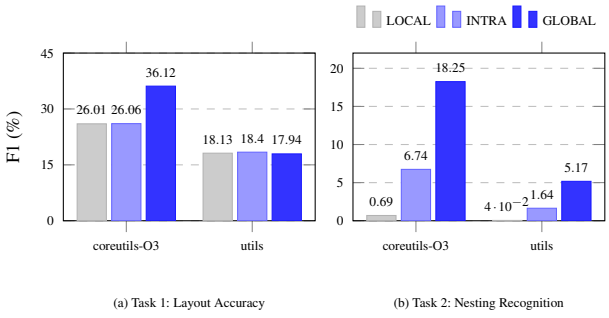
Despite these limitations, a manual inspection of per-parameter results reveals that RecStruct still provides meaningful recovery for a substantial subset of parameters. We find 6,816 parameters achieve perfect recovery, all ground truth fields correctly identified with no spurious fields. Of these, 2,837 involve structures with 3 or more fields, and 218 involve structures with 10 or more fields, demonstrating that the approach succeeds even for complex types when the codebase does not exhibit pathological unification patterns. Additionally, 103,152 parameters achieve at least 50% recall, providing useful partial layouts as starting points for manual refinement.

4.7 RQ5: Ablation Studies

We examine the contribution of key components and parameters to RecStruct's performance.

4.7.1 Impact of Global Type Unification

RecStruct's core innovation lies in its use of the TDG and Stack-Driven Union-Find algorithm to propagate type constraints globally. To quantify this contribution, we progressively expand the analysis scope across three configurations: **LOCAL** (no union-find, purely local inference), **INTRA** (union-find within function boundaries only), and **GLOBAL** (full inter-procedural unification). Figure 6 visualizes the impact on both evaluation tasks.



(a) Task 1: Layout Accuracy (b) Task 2: Nesting Recognition

Figure 6: Impact of analysis scope on structure recovery. (a) Task 1 shows moderate improvement (+10 pts on coreutils-O3). (b) Task 2 exhibits dramatic gains (26×) as analysis expands from LOCAL to GLOBAL.

From Fragments to Topology. The results tell a clear story of progressive improvement as analysis scope expands. Starting from LOCAL, where each function is analyzed in isolation, nested structure recovery is nearly impossible: Task 2 F1 is merely 0.69% on `coreutils-O3` and 0.04% on `utils`. This is expected, as a pointer chain like $A \to B \to C$ is rarely

traversed within a single function. LOCAL analysis sees only disconnected fragments: one function accesses $A \to B$, another accesses $B \to C$, but the connection between them is invisible.

Intra-procedural Consistency: A Stepping Stone. Enabling INTRA-procedural unification provides a meaningful but limited boost. On `coreutils-O3`, Task 2 F1 rises to 6.74%, a 10× improvement over LOCAL. This gain comes from ensuring type consistency within each function's local scope, allowing the system to correctly link dereference chains that appear within the same function body. However, the ceiling is low: most real-world nesting relationships span multiple functions.

Global Unification: The Decisive Factor. The full GLOBAL configuration unlocks RecStruct's true potential. Task 2 F1 reaches 18.25% on `coreutils-O3`, a further 2.7× improvement over INTRA, and a total **26**× improvement over LOCAL. By constructing the inter-procedural TDG and applying Stack-Driven Union-Find across function boundaries, RecStruct can finally link the fragments: when function `f1` passes a pointer of type B to function `f2`, the TDG records this relationship, enabling the system to unify `f1`'s view of $A \to B$ with `f2`'s view of $B \to C$, reconstructing the complete topology $A \to B \to C$. Task 1 also benefits from global analysis, improving by 10 percentage points as partial field views are aggregated into complete definitions.

4.7.2 Incremental Batch Analysis

Full-program analysis maximizes constraint recovery but may be impractical for large binaries. To study the trade-off between scope and fidelity, we conducted an incremental analysis on `openssl` (15,316 functions), partitioning the binary into batches of 500–5,000 functions and merging results via global renaming.

The results, presented in Figure 7, reveal a significant trade-off between local precision and global topology recovery. Most notably, restricting the analysis scope yielded substantial performance gains: analyzing the binary in batches of 1,000 functions completed in just 78 seconds, a 17× speedup compared to the 1,312 seconds required for full-program analysis. More surprisingly, this batch-based approach improved the Layout Accuracy (Task 1) F1 score by nearly 5×, reaching 24.82% compared to 4.84% for the full analysis. This counter-intuitive improvement suggests that in large, monolithic binaries like OpenSSL, global unification can sometimes propagate erroneous constraints too aggressively, causing unrelated types to merge. Partitioning the analysis effectively acts as a regularization mechanism, isolating type constraints and preventing the over-generalization that can degrade field-level precision.

However, this locality comes at a cost to semantic reconstruction. As shown in Figure 7, the Nesting Recognition (Task 2) F1 score increases monotonically with batch size,

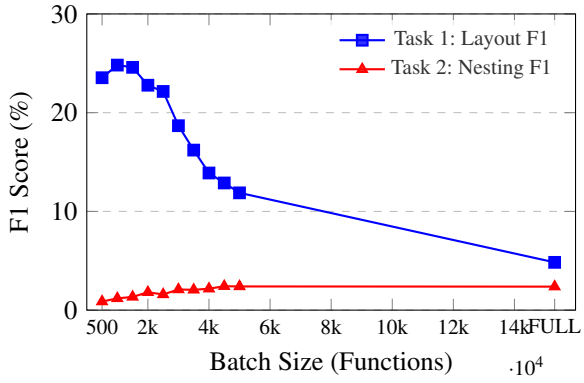


Figure 7: Impact of analysis scope (batch size) on structure recovery metrics for OpenSSL. Smaller batches enhance field layout precision (Task 1) by reducing constraint noise, while larger scopes are necessary to recover inter-procedural nesting relationships (Task 2).

peaking at the full-program analysis. This trend confirms that recovering deep pointer chains (e.g., $A \to B \to C$) relies heavily on inter-procedural evidence that spans across function boundaries. When the analysis is partitioned, the critical links connecting these structures are severed, inhibiting the reconstruction of the complete type graph. These findings indicate that while batch processing offers an efficient strategy for recovering individual structure layouts, holistic full-program analysis remains essential for reconstructing complex, interconnected data models.

4.8 Summary

Our evaluation shows that RecStruct significantly outperforms all baselines, achieving up to 4.9× higher F1 score in layout accuracy and up to 15.7× higher F1 score in nesting recognition compared to constraint-based approaches. These results validate our thesis from Section 1: struct recovery should be treated as a *layout aggregation* problem, effectively addressed by lifting instruction-level traces into SemaLogs and enforcing offset-sensitive unification via Stack-Driven Union-Find. In summary:

- **RQ1:** RecStruct faithfully recovers complex and recursive structures, preserving both field layouts and multi-level pointer-induced nesting relationships.
- **RQ2:** RecStruct consistently achieves higher accuracy than TypeForge, Trex, and ReSym on both layout recovery and nesting recognition.
- **RQ3:** RecStruct is substantially more time-efficient, enabling practical analysis where prior tools incur prohibitive overhead.

- **RQ4:** RecStruct scales to large real-world binaries (e.g., QEMU and OpenSSL), while prior baselines frequently hit timeouts or memory limits.

- **RQ5:** Ablation confirms the necessity of Stack-Driven Union-Find and highlights a controllable trade-off between layout precision and global topology via batch analysis.

5 Discussion

The evaluation results validate our core thesis: effective structure recovery requires a global, inter-procedural perspective. By transforming type inference into a graph connectivity problem solved via Stack-Driven Union-Find, RecStruct overcomes the fragmentation inherent in function-local analyses. Where prior tools view a linked list node passed between functions as disjoint tuples, our global constraint propagation reconstructs unified recursive type definitions. This graph-based approach also yields decisive scalability advantages over SMT-based and LLM-based methods, enabling practical analysis of massive binaries like `qemu` and `openssl` by avoiding the combinatorial explosion of path-sensitive symbolic execution.

However, our ablation studies reveal a trade-off between analysis scope and precision. While full-program analysis is essential for recovering deep nesting, it risks "constraint pollution": erroneous alias analysis or spurious pointer connections can propagate across the TDG, causing aggressive type merging. The observation that smaller batch sizes can improve field-level accuracy suggests global unification is a double-edged sword: it connects valid disjoint views but also propagates noise.

Our approach handles these type-obscuring patterns with varying success. *Type casts* are largely transparent: the underlying dereference still records access offset and width, so layout evidence is preserved unless the cast erases structure entirely (e.g., byte-wise access through `char*`). *Unions* are partially recoverable: RecStruct detects overlapping fields at the same offset but cannot infer tag-to-variant semantics, which requires value-sensitive reasoning beyond our layout-driven abstraction. *Generic types* are the primary failure mode: type-erasing interfaces (e.g., `void*`-based containers) cause all callers' layouts to unify into oversized groups via the TDG's reference edges, producing conflated structures. Our error analysis confirms this as the dominant source of over-unification, consistent with the batch-analysis finding (§4.7) that limiting propagation scope mitigates such pollution. Additionally, our abstraction does not differentiate primitive types of identical width (e.g., `int` vs. `float`). Future work could integrate generic container detection and operational semantics to address both limitations.

Downstream Integration. RecStruct exports recovered types as C header files importable by IDA Pro and Ghidra, directly improving decompiled output readability. We position it as an analyst-facing tool: approximate type information already provides substantial value for manual reverse engineering, whereas fully automated binary rewriting would require stronger precision guarantees that we leave to future work.

In conclusion, RecStruct demonstrates that lifting abstraction from local instruction sequences to global type graphs enables scalable and accurate reverse engineering. Extending this model to handle noise robustly and support C++ polymorphism remains future work.

6 Related Work

We discuss related work in four categories: industrial tools, static analysis, dynamic analysis, and learning-based approaches.

Industrial Reverse Engineering Tools. IDA Pro [35] and Ghidra [1] provide struct recovery via heuristic-based pattern matching within individual functions, but rely heavily on *manual effort* to propagate types across function boundaries [25]. Open-source platforms such as angr [2, 38] and B2R2 [21] offer programmatic analysis APIs but similarly lack automated inter-procedural struct recovery. Language-specific tools such as GoReSym [24] exploit runtime metadata available in Go binaries but are inapplicable to C/C++ targets. In contrast, RecStruct is *fully automated*—our TDG connects type observations across functions without expert annotation.

Static Analysis Methods. Caballero and Lin [7] provide a comprehensive survey of type inference on executables. TIE [3] pioneered constraint-based type recovery using Value Set Analysis (VSA), though it faces scalability challenges. ElWazeer et al. [14] improved scalability for variable and data type detection in binary rewriting. Retypd [27] introduced polymorphic type inference with algebraic subtyping. OSPREY [45] extends constraint-based recovery with probabilistic inference. BTIGhidra [28] implements inter-procedural type constraint propagation as a Ghidra extension. TypeForge [43] constructs a Type Flow Graph (TFG) for inter-procedural composite type synthesis. TRex [5] focuses on behavior-capturing structural types rather than recovering original source types. RecStruct differs in graph granularity: TFG nodes represent *expressions*, whereas TDG nodes represent *local layouts*, reducing complexity. Our *Stack-Driven Union-Find* precisely tracks nesting levels, correctly handling recursive structures that constraint-based methods struggle with.

Dynamic Analysis Methods. REWARDS [22] introduced the *type sink* concept, propagating types from system calls and library functions via dynamic taint analysis. Howard [39] recovers data structures by observing memory access patterns during execution—treating memory as a struct if accessed like one. DSIbin [34] extends Howard's approach by combining dynamic structure identification with type excavation for C/C++ binaries. Tupni [12] and Bossert et al. [6] apply similar dynamic techniques to reverse engineer input formats and protocol structures. These dynamic methods achieve high precision but are limited by code coverage; RecStruct performs purely static analysis and achieves comparable accuracy without requiring program execution.

Learning-Based Approaches. Debin [18] combines Extremely Randomized Trees with Conditional Random Fields to jointly predict variable names and types. CATI [9] and TypeMiner [23] apply machine learning to infer variable types from binary features. DIRTY [10] uses Transformers to predict types from decompiled code but cannot recover user-defined structs outside its training vocabulary. DRAGON [40] further predicts variable types with learned confidence estimates. TYGR [46] employs GNNs for intra-procedural type inference but does not recover struct members. StateFormer [33] learns instruction semantics via generative state modeling for fine-grained type recovery. Learning-based methods have also been applied to function name prediction [13, 17, 20] and variable name recovery [26, 31]. ReSym [44] combines LLMs with Prolog-based reasoning to recover struct declarations with field names. HyRES [37] integrates Datalog-based static analysis with LLM inference for semantic-enhanced structure aggregation. Recent work further applies LLMs to decompilation enhancement [19, 41], decompilation structuring [4], and general binary analysis assistance [8, 29]. Learning-based methods are limited by training data, while LLM-based methods (ReSym, HyRES) incur high computational costs and non-deterministic results. RecStruct requires *no training data*, provides *deterministic analysis*, and achieves 60–250× speedup while supporting full nested structure recovery.

Object-Oriented Recovery. OOAnalyzer [36] uses logic programming to recover C++ classes and methods from compiled executables. MARX [32] uncovers class hierarchies through virtual function table analysis, and DeClassifier [15] infers directed class inheritance relationships from optimized binaries. These approaches target C++ class semantics, whereas RecStruct focuses on recovering C-style struct layouts.

7 Conclusion

We presented RecStruct, a framework that transforms binary structure recovery into a graph connectivity problem through interprocedural type evidence aggregation. Our evaluation demonstrates that RecStruct consistently outperforms existing approaches and scales to large real-world programs where prior methods fail. Future work includes generic container detection to prevent type pollution and enriching primitive type inference through operational semantics.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China under Grant U24B20147, in part by the Sichuan Key Research and Development Program under Grant 2025ZDZX0059, 2026YFHZ0121, 2024ZDZX0044, and 2024ZHCG0195, and in part by the Henan Province Science Foundation for Youths (No. 262300422592). We thank the anonymous reviewers for their constructive feedback and the artifact evaluation committee for their thorough assessment.

Ethical Considerations

We conduct a stakeholder-based ethics analysis following the USENIX Security guidelines and the principles outlined in The Menlo Report.

Stakeholders. We identify three primary stakeholder groups: (1) *security researchers and reverse engineers*, who benefit from automated structure recovery for vulnerability analysis, malware understanding, and legacy code comprehension; (2) *software developers and vendors*, whose compiled binaries may be analyzed using our tool; and (3) *society at large*, which is affected by the downstream security implications of improved binary analysis capabilities.

Ethical Principles and Potential Harms. *Beneficence:* RecStruct's primary purpose is to assist *defensive* security tasks. Accurate type recovery improves the efficiency of vulnerability discovery, incident response, and software auditing—activities that benefit the security of software systems and their users. *Dual-use risk:* As with all program analysis tools, RecStruct could theoretically be used to facilitate unauthorized reverse engineering of proprietary software or to aid exploit development. However, RecStruct recovers only structural type layouts (field offsets and sizes), not semantic information such as variable names, algorithms, or business logic. This significantly limits its utility for intellectual property theft compared to existing, widely available decompilers (e.g., IDA Pro, Ghidra). Furthermore, our tool does not discover vulnerabilities; it provides type context that must be combined with substantial manual expertise.

Mitigations. Our evaluation exclusively uses publicly available, open-source software (GNU coreutils, binutils, grep, gzip, lighttpd, wget, QEMU, OpenSSL) that is freely distributable. We do not analyze any proprietary, commercial, or access-restricted binaries. We do not introduce any new attack techniques, exploit any vulnerabilities, or perform experiments on live systems.

Decision. We believe the benefits of this research—enabling more efficient defensive binary analysis and advancing the state of the art in automated type recovery—substantially outweigh the marginal dual-use risks, which are no greater than those posed by existing open-source reverse engineering tools. We therefore conclude that conducting and publishing this research is ethical.

Open Science

All artifacts necessary to evaluate the contributions of this paper are publicly available:

https://doi.org/10.5281/zenodo.20628819

The repository contains the following artifacts:

- **RecStruct Source Code**: The complete Rust implementation of RecStruct, including the symbolic execution engine, TDG construction, Stack-Driven Union-Find algorithm, and type synthesis pipeline. Build instructions are provided in the repository `README.md`.
- **Evaluation Datasets**: Pre-compiled binaries used in our evaluation (coreutils, binutils, grep, gzip, lighttpd, wget, qemu, openssl). Each binary is provided in two forms: (1) with DWARF debug symbols for ground truth extraction, and (2) stripped for use as input to RecStruct. Pre-extracted ground truth files in YAML format are included alongside re-extraction scripts.
- **Evaluation Scripts**: End-to-end scripts for reproducing all experiments reported in the paper, including ground truth extraction, metric computation (Task 1 Layout Accuracy, Task 2 Nesting Recognition, and Trex-style hierarchical scoring), and baseline comparison pipelines.
- **Baseline Tool Configurations**: Configuration files and wrapper scripts used to run TypeForge, Trex, and ReSym under identical experimental conditions.

References

[1] National Security Agency. Ghidra: Software reverse engineering framework. https://ghidra-sre.org/, 2019. [Online; accessed 2025-11-13].

[2] angr Project. angr: A binary analysis framework. https://github.com/angr/angr, 2016. [Online; accessed 2025-11-12].

[3] Thanassis Avgerinos. Tie: Principled reverse engineering of types in binary programs. In *Proceedings of the 18th Annual Network and Distributed System Security Symposium (NDSS'11)*, 2011.

[4] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. Ahoy sailr there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 361–378, 2024.

[5] Jay Bosamiya, Maverick Woo, and Bryan Parno. Trex: Practical type reconstruction for binary code. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 6897–6915, 2025.

[6] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 51–62, 2014.

[7] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys (CSUR)*, 48(4):1–35, 2016.

[8] Guoqiang Chen, Huiqi Sun, Daguang Liu, Zhiqi Wang, Qiang Wang, Bin Yin, Lu Liu, and Lingyun Ying. Recopilot: Reverse engineering copilot in binary analysis, May 2025. arXiv:2505.16366 [cs].

[9] Ligeng Chen, Zhongling He, and Bing Mao. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 88–98, Valencia, Spain, June 2020. IEEE.

[10] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[11] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. Selectivetaint: Efficient data flow tracking with static binary rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1665–1682, 2021.

[12] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402, 2008.

[13] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, November 2020.

[14] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 51–60, 2013.

[15] Rukayat Ayomide Erinfolami and Aravind Prakash. Declassifier: Class-inheritance inference engine for optimized c++ binaries. In *Proceedings of the 2019 ACM*

Asia Conference on Computer and Communications Security, pages 28–40, 2019.

[16] Qian Feng, Rundong Zhou, Chengcheng Xu, Yao Cheng, Brian Testa, and Heng Yin. Scalable graph-based bug search for firmware images. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*, pages 480–491, 2016.

[17] Han Gao, Shaoyin Cheng, Yinxing Xue, and Weiming Zhang. A lightweight framework for function name reassignment based on large-scale stripped binaries. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 607–619, Virtual Denmark, July 2021. ACM.

[18] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, Toronto Canada, October 2018. ACM.

[19] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. In *Proceedings 2024 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2024. Internet Society.

[20] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, Los Angeles CA USA, November 2022. ACM.

[21] Minkyu Jung, Soomin Kim, HyungSeok Han, Jaeseung Choi, and Sang Kil Cha. B2r2: Building an efficient front-end for binary analysis. In *Proceedings 2019 Workshop on Binary Analysis Research*, San Diego, CA, 2019. Internet Society.

[22] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, pages 1–1, 2010.

[23] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. Typeminer: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 288–308. Springer, 2019.

[24] Mandiant. Goresym: Go symbol recovery tool. https://github.com/mandiant/GoReSym, 2021. [Online; accessed 2025-11-12].

[25] Mandiant. Applying function types to structure fields in ida. https://cloud.google.com/blog/topics/threat-intelligence/function-types-structure-fields-ida, 2024. [Online; accessed 2025-11-12].

[26] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. Direct: a transformer-based model for decompiled identifier renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, pages 48–57, 2021.

[27] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 27–41, 2016.

[28] Trail of Bits. Binary type inference in Ghidra. https://blog.trailofbits.com/2024/02/07/binary-type-inference-in-ghidra/, February 2024. [Online; accessed 2025-11-12].

[29] OpenAI. Gpt-4o: A multimodal ai model for text, audio, and vision. https://openai.com/index/hello-gpt-4o/, 2024. [Online; accessed 2025-04-23].

[30] Bindu Madhavi Padmanabhuni and Hee Beng Kuan Tan. Buffer overflow vulnerability prediction from x86 executables using static analysis and machine learning. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 450–459. IEEE, 2015.

[31] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, Adam Doupé, Chitta Baral, and Ruoyu Wang. "len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4069–4087, San Francisco, CA, USA, May 2024. IEEE.

[32] Andre Pawlowski, Moritz Contag, Victor van der Veen, Chris Ouwehand, Thorsten Holz, Herbert Bos, Elias Athanasopoulos, and Cristiano Giuffrida. Marx: Uncovering class hierarchies in c++ programs. In *NDSS*, 2017.

[33] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software*

Engineering Conference and Symposium on the Foundations of Software Engineering, pages 690–702, Athens Greece, August 2021. ACM.

[34] Thomas Rupprecht, Xi Chen, David H. White, Jan H. Boockmann, Gerald Lüttgen, and Herbert Bos. Dsibin: Identifying dynamic data structures in c/c++ binaries. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 331–341, October 2017.

[35] Hex-Rays SA. Ida pro: The interactive disassembler. https://www.hex-rays.com/ida-pro/, 2025. [Online; accessed 2025-11-12].

[36] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover c++ classes and methods from compiled executables. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 426–441, Toronto Canada, October 2018. ACM.

[37] Zihan Sha, Hui Shu, Hao Wang, Zeyu Gao, Yang Lan, and Chao Zhang. Hyres: Recovering data structures in binaries via semantic enhanced hybrid reasoning. *ACM Transactions on Software Engineering and Methodology*, 35(3):1–28, 2026.

[38] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[39] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *NDSS*, 2011.

[40] Caleb Stewart, Rhonda K. Gaede, and Jeffrey H. Kulick. Dragon: Predicting decompiled variable data types with learned confidence estimates. In *Proceedings 2025 Workshop on Binary Analysis Research*, San Diego, CA, USA, 2025. Internet Society.

[41] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. Llm4decompile: Decompiling binary code with large language models. In *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487, 2024.

[42] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.

[43] Yanzhong Wang, Ruigang Liang, Yilin Li, Peiwei Hu, Kai Chen, and Bolun Zhang. Typeforge: Synthesizing and selecting best-fit composite data types for stripped binaries. In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 1–18, San Francisco, CA, USA, May 2025. IEEE.

[44] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu 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*, pages 4554–4568, Salt Lake City UT USA, December 2024. ACM.

[45] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wenchuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832, San Francisco, CA, USA, May 2021. IEEE.

[46] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. Tygr: Type inference on stripped binaries using graph neural networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, 2024.

Appendix

A.1 Representativeness of Parameter-Based Evaluation

A natural concern with our parameter-focused evaluation scope (Section 4.1) is whether it adequately represents the full landscape of structure types in a binary. To address this, we quantify the overlap between structures that appear as function parameters or return types and those found exclusively in stack frame variables.

Table 9 summarizes the results across four benchmark datasets. *Defined* lists the total number of distinct structure types recorded in the DWARF ground truth. *InParam* counts those appearing in at least one function parameter or return type. *InStack* counts those present in at least one stack frame variable. *OnlyStack* counts structure types that appear exclusively in stack variables and never in any parameter or return position. The *Ratio* column reports $\mathit{OnlyStack}/(\mathit{InParam}+\mathit{OnlyStack})$, i.e., the fraction of actively used structure types that would be invisible to a parameter-only evaluation.

Table 9: Structure type usage across function parameters and stack variables. "OnlyStack" types are those never appearing in any function signature.

Dataset	Defined	InParam	InStack	OnlyStack	Ratio
binutils	1704	567	233	85	13.0%
coreutils-O1	214	115	84	26	18.4%
coreutils-O2	215	116	83	26	18.3%
coreutils-O3	213	116	81	24	17.1%

Across all datasets, only 13–18% of structure types are stack-only. Manual inspection reveals that these types fall into two characteristic categories:

- **Anonymous nested types.** Compiler-generated names such as `anon_union_152_2_xxx` denote unnamed unions or structs embedded inside a larger named type. Because they lack standalone type identity, they cannot appear as parameter or return types and only surface when their enclosing parent structure is allocated on the stack.
- **Function-local auxiliary structures.** Types such as `Active_dir`, `F_triple`, and `Src_to_dest` are small (typically 16–24 bytes) bookkeeping structures whose lifetimes are confined to a single function. By construction, they are never passed across function boundaries.

Neither category participates in inter-procedural data flow—the primary target of aggregate type recovery. Consequently, a parameter-focused evaluation covers 82–87% of all actively used structure types, with the excluded remainder being structurally trivial or inherently invisible at function interfaces. This confirms that our evaluation scope is both representative and practical for assessing structure recovery quality.

A.2 Complexity Analysis

The Stack-Driven Union-Find algorithm (Algorithm 1) explores configurations of the form (L,S), where L is a layout node and S is the offset stack. Without bounds, the number of distinct stacks reaching a node can grow rapidly with graph branching and cycles, and thus the worst-case running time can be far above linear in $|E|$.

In our implementation, we employ a *budgeted exploration policy* with three parameters: maximum stack depth D, a hop-based time-to-live R, and a per-node expansion cap C. Under these budgets, each node is expanded at most C times, and each expansion scans incident edges, yielding a time bound of $O(C \cdot m_R)$, where m_R denotes the number of edges reachable within the TTL radius. Since D, R, and C are small constants in all experiments (default values: $D=10$, $R=10$, $C=50$), the algorithm remains efficient in practice. For a TDG with $|V|$ nodes and $|E|$ edges, the practical complexity is $O(|V| \cdot C \cdot \bar{d})$, where $\bar{d}$ is the average node degree.

A.3 SemaLog Walkthrough

We illustrate the end-to-end SemaLog extraction process using `func2` from the `old-case` micro-benchmark (Section 4.2), which exercises all four SemaLog types and multi-level pointer chains up to three dereferences deep.

```
void func2(struct A* a) {
  struct D* d = (struct D*)malloc(sizeof(struct D));
  d->d1 = 0x112233;
  d->a = a;
  a->b->d = d;
  a->b->c->c1 = 0x112233;
  a->c->c2 = 0x445566;
}
```

Figure 8: Source code for `func2` in the `old-case` benchmark. The function allocates a `struct D`, links it bidirectionally with `a` and `a->b`, and accesses fields through pointer chains up to three levels deep.

RecStruct's symbolic execution translates each binary instruction into SemaLog entries. Table 10 shows the cleaned trace for `func2` (stack-frame bookkeeping such as saving/restoring the base pointer and spilling arguments to the stack is elided for clarity). Each DEREF entry is classified as *Data* when the accessed value is concrete and the size is sub-pointer-width (4 bytes), or *Link* when the value is symbolic and pointer-width (8 bytes). Link dereferences simultaneously generate dereference edges in the TDG.

Walkthrough. Logs 0–2 establish symbolic bases: `a1` for the function parameter (a pointer to `struct A`) and `v1` for the `malloc` return value. The CALL log records the allocation with the constant argument 16 reflecting `sizeof(struct D)`.

Table 10: Cleaned SemaLog trace for `func2`. "⊥" denotes a concrete (non-symbolic) value. *Data* and *Link* indicate the dereference classification.

#	Op	Content	Class	Source
0	LOC	`a1` : Arg(0)	–	param `a`
1	CALL	`malloc(16)` → `v1`	–	alloc `D`
2	LOC	`v1` : Local	–	result `d`
3	DEREF	`v1+0` [4B] → ⊥	Data	`d->d1`
4	DEREF	`v1+8` [8B] → `a1`	Link	`d->a=a`
5	LOC	`*a1_8` : Local	–	interm.
6	DEREF	`a1+8` [8B] → `*a1_8`	Link	load `a->b`
7	DEREF	`*a1_8+16` [8B] → `v1`	Link	`b->d=d`
8	LOC	`**a1_8_8` : Local	–	interm.
9	DEREF	`*a1_8+8` [8B] → `**a1_8_8`	Link	load `b->c`
10	DEREF	`**a1_8_8+0` [4B] → ⊥	Data	`c->c1`
11	LOC	`*a1_10` : Local	–	interm.
12	DEREF	`a1+16` [8B] → `*a1_10`	Link	load `a->c`
13	DEREF	`*a1_10+4` [4B] → ⊥	Data	`c->c2`

Logs 3–4 access fields of `v1` (`struct D`). Log 3 writes a 4-byte constant to `v1+0`; since the written value is concrete (⊥) and the access width (4B) is less than the pointer size, it is classified as a *DataDeref*, recording a primitive field `d->d1`. Log 4 stores the symbolic pointer `a1` at `v1+8` with pointer-width access (8B), yielding a *LinkDeref* that simultaneously creates a TDG dereference edge `v1` $\xrightarrow{+8}$ `a1`, encoding the `d->a` pointer field.

Logs 5–10 demonstrate multi-level dereference chains. The symbolic executor automatically introduces *intermediate layout nodes*: `*a1_8` names the result of dereferencing `a1` at offset 8 (i.e., `a->b`, yielding a pointer to `struct B`), and `**a1_8_8` names a further dereference of `*a1_8` at offset 8 (i.e., `a->b->c`, a pointer to `struct C`). The naming convention is: `*base_off` for dereferencing `base` at offset `off` (in hex). Log 10 writes a 4-byte constant to `**a1_8_8+0`—this captures the three-level access chain `a->b->c->c1` as a DataDeref, while the preceding LinkDerefs (logs 6, 9) create TDG edges encoding the pointer topology: `a1` $\xrightarrow{+8}$ `*a1_8` $\xrightarrow{+8}$ `**a1_8_8`.

Logs 11–13 record the independent two-level chain `a->c->c2`: a LinkDeref from `a1+16` producing intermediate node `*a1_10`, followed by a DataDeref at `*a1_10+4`.

Resulting TypeLayouts. The TDG builder aggregates these SemaLogs into five per-symbol local layouts, shown in Table 11. Each layout corresponds to a partial view of a ground-truth structure: `a1`↔`struct A` (two pointer fields at offsets 8 and 16), `*a1_8`↔`struct B` (two pointer fields at offsets 8 and 16), `v1`↔`struct D` (one data field and one pointer field), and `**a1_8_8`, `*a1_10`↔`struct C` (one data field each, at different offsets). Critically, the last two layouts are *distinct nodes* created from different pointer chains (`a->b->c` and `a->c`), yet represent the same underlying type. Stage 3's Stack-Driven Union-Find will unify them by discovering that both are reachable from `a1` through dereference paths that reduce to the same offset-stack signature.

Table 11: Local layouts produced by `func2`. Link fields list their target layout.

Layout (Kind)	Fields	Ground Truth
`a1` (Arg)	+8: 8B→`*a1_8`; +16: 8B→`*a1_10`	`struct A`
`v1` (Local)	+0: 4B data; +8: 8B→`a1`	`struct D`
`*a1_8` (Local)	+8: 8B→`**a1_8_8`; +16: 8B→`v1`	`struct B`
`**a1_8_8` (Local)	+0: 4B data	`struct C`
`*a1_10` (Local)	+4: 4B data	`struct C`