NotDec: WebAssembly Decompilation With Inter-Procedural Type Recovery

Jikai Wang[†]
Huazhong University of Science and Technology
Wuhan, China
warrenwjk@gmail.com

Ningyu He[*]
The Hong Kong Polytechnic University
Hong Kong SAR, China
ningyu.he@polyu.edu.hk

Tianming Liu[†]
Huazhong University of Science and Technology
Wuhan, China
tmliu@hust.edu.cn

Junhai Wang[†]
Huazhong University of Science and Technology
Wuhan, China
junhaiwang@hust.edu.cn

Haoyu Wang[*†]
Huazhong University of Science and Technology
Wuhan, China
haoyuwang@hust.edu.cn

Abstract

With WebAssembly widely supported in browsers, containers, IoT devices, and serverless platforms and increasingly adopted as a universal low-level bytecode standard, auditing its hidden vulnerabilities and malicious intentions has become critical. Decompiling existing WebAssembly modules can help security researchers and end users understand binary behavior, but current tools suffer from verbose result, poor readability, and limited type recovery.

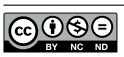
We present NotDec, an advanced WebAssembly decompilation framework. NotDec extends the WebAssembly type checking algorithm to lift bytecode into an SSA-based IR, applies the inter-procedural type recovery algorithm Retypd with pointer and numeric value differentiation methods to recover complex data structures, and leverages Memory SSA alongside semantics-preserving structured control-flow analysis to emit readable, semantically consistent C code.

NotDec achieves 100% recompilation success rate on all 5,241 Juliet samples and all Howard dataset programs, significantly outperforming baselines including Ghidra (45.95% success rate). On type recovery accuracy, NotDec recovers 85.33% of struct member accesses in real-world programs, vastly exceeding Ghidra's 9.24%. While the full inter-procedural version faces scalability challenges on large binaries, the intra-procedural variant NotDec$_F$ demonstrates superior efficiency, consuming less than half of Ghidra's memory and up to 97% less execution time on unoptimized binaries.

CCS Concepts

• **Security and privacy → Software reverse engineering**.

[*]Ningyu He and Haoyu Wang are the corresponding authors.
[†]The full name of the authors' affiliation is Hubei Key Laboratory of Distributed System Security, Hubei Engineering Research Center on Big Data Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology.



ICSE '26, Rio de Janeiro, Brazil

ACM ISBN 979-8-4007-2025-3/26/04
https://doi.org/10.1145/3744916.3787762

Keywords

Reverse Engineering, Binary Analysis, Decompilation

ACM Reference Format:
Jikai Wang, Ningyu He, Tianming Liu, Junhai Wang, and Haoyu Wang. 2026. NotDec: WebAssembly Decompilation With Inter-Procedural Type Recovery. In *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE '26), April 12–18, 2026, Rio de Janeiro, Brazil.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3744916.3787762

1 Introduction

WebAssembly is a fast and compact low-level bytecode standard designed to boost web performance. It serves as a compilation target for languages such as C/C++, Rust, and Go [30]. As of March 2024, 99% of monitored web browsers support it. Beyond the web, WebAssembly is also widely used in containers, IoT, and serverless platforms, offering security isolation and performance benefits [34].

However, WebAssembly code still faces serious security risks. Studies show that memory safety flaws from source languages (e.g., C) persist after conversion to WebAssembly [35] and can be exploited by attackers [23]. Moreover, over 50% of websites using WebAssembly deploy it for malicious activities like cryptocurrency mining [25].

To address security risks from WebAssembly, researchers and users often need to audit existing modules. However, the source of WebAssembly code may be untrustworthy and is often not visible to clients. While some program analysis methods, such as symbolic execution [19], can partially recover semantics, they have a high usage barrier and are limited by issues like path explosion. Therefore, decompilation offers a more practical solution, allowing typical users to understand and analyze the internal behavior of WebAssembly modules.

Decompiling WebAssembly binaries is challenging. First, WebAssembly's operand stack's design complicates static code analysis, and its strict pre-execution type-checking mechanism limits code modification, necessitating conversion to a more analyzable intermediate representation (IR) to facilitate subsequent analysis and transformation. Second, to produce readable output, recovering high-level types such as custom structs from low-level code remains difficult. Because it requires inferring types from pointer

usage patterns of untyped linear memory, a task that remains a known limitation of current WebAssembly decompilers. Finally, to produce readable C code enriched with meaningful high-level types, it is essential to carefully preserve semantic equivalence throughout the code transformation.

To address WebAssembly's operand stack and type checking challenges, we extend the type checking algorithm [8] to convert WebAssembly into an SSA-based IR, enabling the use of existing mature code analysis and optimization methods. To address the challenges of type recovery in linear memory, we introduce a state-of-the-art native binary type recovery algorithm Retypd [27]. To bridge the gap in applying type recovery algorithms, we propose the PNDiff Graph, which distinguishes pointer values from numeric values, producing precise pointer constraints for type recovery. To maintain semantic consistency during C code generation, we employ Memory SSA [6] to ensure memory reference expressions match actual memory contents. and use a semantics-preserving structuring algorithm to convert the CFG into control flow statements, preserving result semantics.

This work. In this work, we propose NotDec, a WebAssembly decompiler that supports custom structure recovery. It comprises three modules. The WebAssembly Code Lifting module parses WebAssembly code and converts it into SSA-based IR. The Optimization and Type Recovery module optimizes the code and generates a mapping from values in the IR to high-level C language types. The Control Flow Structuring and C Code Generation module transforms the intermediate representation with high-level type information into C code.

In our evaluation using the Juliet dataset (5,241 samples) and the Howard dataset (five large-scale programs), we demonstrate both efficiency and effectiveness. For efficiency (§4.1), while the full inter-procedural NotDec faces performance issues on large binaries (e.g., timing out in some cases), its intra-procedural variant NotDec$_F$ achieves less time and memory consumption compared to Ghidra, particularly on unoptimized binaries. For effectiveness (§4.2), our approach excels in type recovery, successfully reconstructing 85.33% of struct member accesses in the Howard dataset, vastly outperforming Ghidra, which only recovers 9.24%. An ablation study (§4.3) shows that code optimization combined with low-level pattern matching effectively reduces redundancy in binary code. The type recovery module contributes to the recovery of all of struct types and their associated accesses. Furthermore, deconstructing the stack into concrete variables yields additional reduction in LoC.

Contribution. In this work, our major contributions are:

- **WebAssembly Decompiler.** We present a novel WebAssembly decompiler that supports recovery of user-defined (custom) structure types and produces re-compilable high-level code.
- **Decompilation System Design.** We integrate mature compiler optimization techniques into decompilation and combine them with type-recovery algorithms (§3.2.1). Furthermore, we model each function's stack in linear memory as a single large structure and leverage compiler optimizations to decompose it into individual stack variables (§3.2.3). Besides, we introduce Memory SSA to resolve whether memory reads have been updated when transforming SSA-based intermediate representations back into non-SSA code (§3.3).
- **Efficiency.** NotDec achieves 15.9× faster execution and 4.8× lower memory consumption than Ghidra on the Juliet dataset, with its intra-procedural variant NotDec$_F$ scales effectively to large binaries, consuming less than half of Ghidra's memory and up to 97% less execution time on unoptimized code.
- **Effectiveness.** NotDec is the only tool achieving 100% recompilation success across all evaluated binaries and recovers 85.33% of struct member accesses compared to Ghidra's 9.24%, demonstrating substantial improvements in code readability and semantic reconstruction.
- **Open Source**: NotDec is available at https://github.com/NotDec/NotDec.

2 Background & Challenges

2.1 Background

WebAssembly. WebAssembly (Wasm) was initially created to improve web application performance through its lightweight and compact binary format. Its loading, validation, and execution speeds far exceed those of traditional JavaScript, while running in a secure sandbox and working in conjunction with JavaScript [30]. Today, WebAssembly has evolved into a general-purpose low-level bytecode standard that supports various programming languages such as C/C++, Rust, and Go [20]. To abstract various memory management methods of different languages, WebAssembly retains a naive array-like linear memory structure.

Structured Control Flow & Type Checking. WebAssembly features unique structured control flow and type-checking mechanisms. Unlike traditional assembly code that allows arbitrary jumps, WebAssembly uses `block`, `loop`, and `end` to explicitly mark the nested levels of the code. Control flow jumps are only permitted from within a nested block to the end of the block (for `block` blocks) or to the beginning (for `loop` blocks) [17].

WebAssembly has an operand stack from which all instructions read operands and push the results onto the stack. There is also a type checking algorithm [8] that ensures that the values obtained from the operand stack are always of the correct type. Additionally, each block declares parameter and return types. At the beginning and the end of each block, if the types the remained operands in the stack mismatch with the parameter or the return type, the type check will fail. To this end, the WebAssembly module will be rejected during loading, thereby preventing the code from running.

WebAssembly Decompilation. Existing academic WebAssembly decompilers have limited capabilities and often produce code lacking high-level types. WaDec [31] converts WebAssembly bytecode to C code using fine-tuned large language models, but it suffers from hallucinations and cannot ensure semantic consistency or even syntactic correctness. SnowWhite [24] infers high-level types for parameters and return values based on sequence generation models, yet it only recognizes common data structure patterns (e.g., FILE descriptors) and cannot handle custom ones. We also underline that both learning-based methods also lack interpretability.

As for industrial tools, IDA's Hex-rays decompiler [5] supports loading WebAssembly but does not support decompilation. Ghidra [7] can support WebAssembly decompilation by installing

a community plugin (ghidra-wasm-plugin). There are also some decompilation tools that merely generate "low-level" C code, such as wasm2c from WABT (The WebAssembly Binary Toolkit) and wasmdec[1] (the highest-starred "decompiler" on GitHub). Their output is typically hard to read, characterized by excessive `goto` statements and pointer operations expressed as numeric calculations—only cast to pointer types during memory accesses. These limitations significantly reduce code readability and hinder further analysis.

2.2 Challenge & Solution

There are three main challenges in decompiling WebAssembly code.
Challenge #1: Handling the operand stack of WebAssembly for better code analysis and optimization. The operand stack obscures use-def relationships between instructions, complicating code analysis. Additionally, WebAssembly's type-checking mechanism restricts stack usage, making code transformations difficult [8]. For example, if instructions are inserted without considering stack balance, it can easily lead to misalignment of values on the operand stack, causing the code to fail type checking and thus become invalid [14]. Effective analysis and transformation are essential for decompilation, posing a significant challenge.
Challenge #2: Recovering local variables and custom data structures. WebAssembly's operand stack cannot take addresses or support aggregate types, so compilers often maintain another call stack in the untyped linear memory. This results in a significant loss of type information for local variables, requiring their high-level types to be inferred from stack memory access patterns, similar to traditional binary reverse engineering. Moreover, the code may contain complex custom structures type (such as recursive types), and the usage of types may be distributed across different functions.
Challenge #3: Reuse memory load expression to make code concise while ensuring semantic consistency. If a memory value is modified between its load and subsequent use, the load no longer corresponds to the current variable value but to its prior state, necessitating a temporary variable. Ensuring that load expressions map to the correct value, especially when they may represent variable accesses, is a key challenge.
Our Solution. For **Challenge #1**, we lift WebAssembly bytecode to an SSA-based IR, allowing existing compilation optimizations and analyses to be applied (§ 3.1). Leveraging the static determinism of the operand stack, we translate instructions directly into a register-based SSA form, preserving original semantics.

For **Challenge #2**, we applied the state-of-the-art binary type recovery algorithm, Retypd [27] (§ 3.2.2). Additionally, to distinguish between pointer and numeric types to extract more constraints on types, we proposed PNDiff graph and apply a set of inference rules.

For **Challenge #3**, we employ Memory SSA to determine if a memory location remains unchanged after its last load. A temporary variable is introduced only when a potential modification is detected, thereby balancing correctness with readability (§3.3).

3 Design Overview

The architecture and workflow of our proposed decompilation process is shown in Figure 1. Similar to a compiler's architecture, our approach is divided into three parts: (*i*) *Front End.* The front end takes a WebAssembly code file as the input, in which the WebAssembly code lifting module is responsible for converting it into an SSA-based intermediate representation (*e.g.,* LLVM IR); (*ii*) *Middle End.* The middle end is mainly responsible for optimization and type recovery, outputting the optimized IR and high-level type information for each value in the IR; and (*iii*) *Back End.* In the back end, the control flow structuring and C code generation module is responsible for converting the IR with high-level type information into a more readable piece of C code, which will be output finally.

3.1 Front End: Code Lifting

As noted in **Challenge #1**, mandatory type verification restricts WebAssembly programs, making direct transformation and optimization on WebAssembly instructions difficult. To facilitate better code analysis and transformation, we convert WebAssembly code into a register-based Static Single Assignment (SSA) Intermediate Representation (IR) format. The conversion to SSA is also mentioned in the design rationale document [9], and implemented by V8 and SpiderMonkey [30]. We underline that SSA is widely used in compiler optimizations and in decompilers (like Ghidra's P-Code IR [39]). Converting to SSA enables the application of a wide range of existing and mature code analysis and optimization algorithms.

We first build use-def relationships for all instructions by extending the specification's type checking algorithm [8] described by the specification, transforming the operand-stack-based form into a register-based SSA form. Then, we continue the process according to instruction types:

- **Variable Instructions.** For WebAssembly's global and local variables of simple types, *i.e.,* i32, i64, f32, and f64. We allocate mutable global or local memory. Instructions like `local.get` and `global.set` become loads and stores to that memory.
- **Memory Instructions.** We map linear memory to a global byte array of equal size and transform all memory access operations like *i32.load* into accesses to this array. This memory variable will later be split during type analysis (see § 3.2.3).
- **Control Instructions.** Structured control flow is converted to a standard control flow graph. For `br_table`, we resolve all possible targets and emit a `switch` instruction in the IR.
- **Table/Element Section & Indirect Call.** In WebAssembly, indirect call locates the target function via a table (defined in the element section) storing function references. We create a corresponding global function pointer array for each function table in the element section. When encountering a `call_indirect`, we access this global array by index, retrieve the function pointer, and then generate a function pointer call.
- **Vector Instructions.** We create corresponding SIMD instructions in the IR according to its semantics.

3.2 Middle End: Optimization & Type Recovery

As shown in Figure 1, the middle end takes the lifted IR file as input, performs optimization and type recovery, and outputs an optimized IR along with a mapping from IR values to high-level types. The detailed architecture (Figure 2) involves three main steps: *pre-optimization and low-level pattern recognition*, *full-context-sensitive polymorphic type inference*, and *stack SROA and post-optimization.*

[1] https://github.com/wwwg/wasmdec

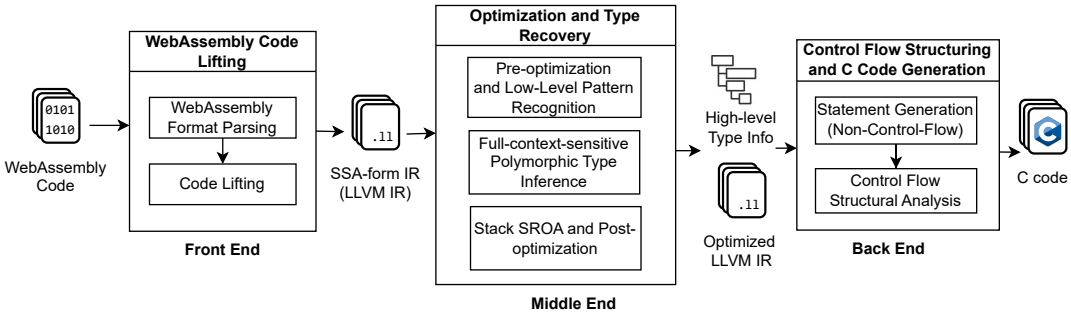


Figure 1: The overview of our WebAssembly decompilation method.

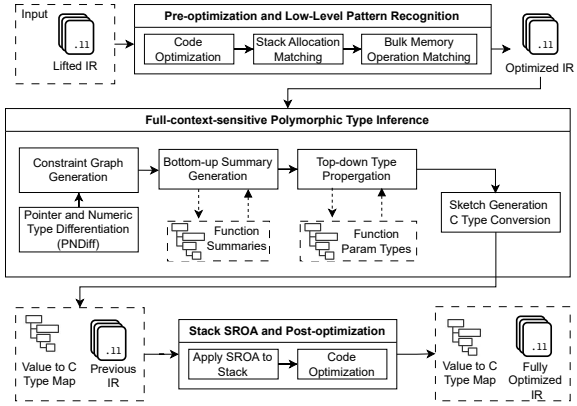


Figure 2: The main process of type recovery.

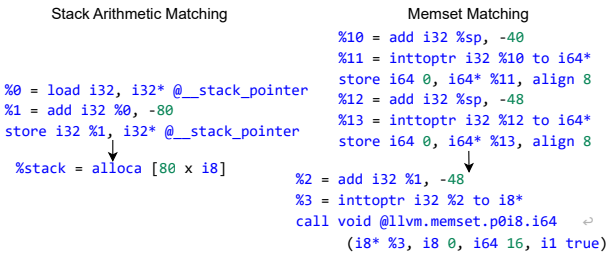


Figure 3: Examples shows the conversion of instructions for stack allocation matching and memset matching.

3.2.1 Pre-optimization and Low-level Pattern Recognition. We first apply code optimization to eliminate redundancies in the lifted code (especially from unoptimized binaries). Additionally, reducing the size of the code can increase the efficiency of subsequent analyses. As for the pattern recognition, it focuses on two key tasks: identifying stack allocations and recognizing common memory operations such as `memcpy` and `memset`. For stack allocation, we replace low-level stack pointer manipulations with explicit stack space allocations, making stack memory visible to type analysis and enabling correct type inference for local variables. As for `memcpy` and `memset`, compilers often translate them in WebAssembly into sequences of fixed-size (e.g., 8-byte) load and store instructions. If left unhandled, these patterns can mislead type recovery by suggesting incorrect data layouts or member structures. Recognizing

```c
struct linkedBuf {
  char buf[50]; struct linkedBuf* next;
};
struct linkedBuf*
getEnd(struct linkedBuf* L) {
  while(L->next) { L = L->next; } return L;
}
void printBuf(struct linkedBuf* L) {
  puts(L->buf);
}
int main() {
  struct linkedBuf B1 = {};
  struct linkedBuf *E = getEnd(&B1);
  printBuf(E); return 0;
}
```

(a) Original code to be compiled to WebAssembly.

```c
// Decompilation result of our tool
struct struct_3328 {
    char field_3329[52] /* at offset: 0 */ ;
    struct struct_3328 *field_3330 /* at offset: 52 */ ;
};
static struct struct_3328 *
getEnd(struct struct_3328 *_arg_0) {
    struct struct_3328 *stack__4_0 = _arg_0;
    do
        stack__4_0 = stack__4_0->field_3330;
    while (stack__4_0->field_3330 != (void *)0);
    return stack__4_0->field_3330;
}
// Decompilation result from Ghidra + ghidra-wasm-plugin
int getEnd(int param1) {
  int local_4;
  for (local_4 = param1; *(int *)(local_4 + 0x34) != 0;
    local_4 = *(int *)(local_4 + 0x34)) {
  }
  return local_4;
}
```

(b) Decompilation result of `getEnd` using our tool and Ghidra.

Figure 4: Sample code that requires inter-procedural type analysis.

and abstracting them back into high-level operations helps preserve accurate type information.

These steps are implemented as a sequence of transformation passes over the intermediate representation (IR). We perform pattern recognition after code optimization because compiler optimizations often rewrite related operations into standardized, *i.e., canonical*, forms. This simplifies pattern matching by reducing variations we need to handle. For stack pointer operations, we look not only for the typical stack pointer decrement at the beginning of a function (the entry block) but also for dynamic stack allocations that occur elsewhere in the code—such as those generated by C's `alloca` function. For `memset` and `memcpy`, we scan for sequences of consecutive memory stores to the same region, and replace them with a single intrinsic `memset` or `memcpy` call. For example, as shown on the left side of Figure 3, three instructions identified as adjusting the stack pointer are transformed into a single, clear stack allocation statement. On the right, a sequence of consecutive store instructions are collapsed into a single `memset` call.

3.2.2 Full-context-sensitive Polymorphic Type Inference. To improve code readability, we need to analyze the high-level type information within the code. The input for this stage is the optimized IR, and after analysis, the output will be a mapping of each value in the IR to its corresponding high-level C type.

Due to the fact that WebAssembly's linear memory is untyped and byte-addressable, its type recovery closely resembles that of native binary code. Therefore, we attempt to introduce a state-of-the-art binary type recovery algorithm, Retypd [27], into WebAssembly.

Table 1: Inference rules for addition and subtraction.

Rule	$X + Y = Z$	$X - Y = Z$
1	X: Num, Y: Num $\rightarrow$ Z: Num	X: Num $\rightarrow$ Y: Num, Z: Num
2	Z: Num $\rightarrow$ X: Num, Y: Num	Y: Num, Z: Num $\rightarrow$ X: Num
3	X: Ptr $\rightarrow$ Y: Num, Z: Ptr	Y: Num, Z: Ptr $\rightarrow$ X: Ptr
4	Y: Num, Z: Ptr $\rightarrow$ X: Ptr	Y: Ptr $\rightarrow$ X: Ptr, Z: Num
5	Y: Ptr $\rightarrow$ X: Num, Z: Ptr	X: Ptr, Z: Num $\rightarrow$ Y: Ptr
6	X: Num, Z: Ptr $\rightarrow$ Y: Ptr	X: Ptr, Y: Num $\rightarrow$ Z: Ptr
7	–	X: Ptr, Z: Ptr $\rightarrow$ Y: Num

For example, when decompiling the WebAssembly code generated from the code in Figure 4a, our tool can produce results for this sample function with the correct type information (in Figure 4b, line 2-13). In contrast, the code generated by Ghidra contains pointer arithmetic and type casting (at line 17-18), impacting the readability.

Pointer and Numeric Type Differentiation (PNDiff). Retypd cannot be directly applied as it relies on correctly identifying, among integer-typed values, those that serve as memory addresses (i.e., pointers) and their associated pointer arithmetic operations, which are essential constraints for subsequent type inference. This differentiation is critical because: (*i*) pointer addition typically changes the pointer's target type; (*ii*) constant pointer offsets often indicate accesses to structure members; and (*iii*) variable pointer offsets may suggest array indexing.

This problem is a fundamentally undecidable challenge in native binary analysis. WebAssembly faces the same issue because its memory instructions also use integer types (e.g., `i32`), making pointers syntactically indistinguishable from integers. Therefore, we introduce *PNDiff graph*, a graph-based solution that adapts established principles from native binary analysis for differentiating pointers from integers (i.e., rules in Table 1).

Specifically, the PNDiff analysis proceeds as follows:

(1) Node creation. Each program value is represented as a node in the PNDiff graph, annotated with a *PN Type*, which is one of four categories: `Ptr` (pointer), `Num` (numeric), `Unknown` (could be either), or `Non-PN` (e.g., floating-point values).
(2) Initialization. We seed the graph with known type information. For addresses used in load/store instructions are initialized as `Ptr`. Values involved in numeric-only operations (e.g., multiplication, division) are marked as `Num`.
(3) Node merging. Dataflows like assignment operations induce equivalence between source and destination values, so their corresponding nodes are merged. Similarly, equality comparisons (`==`, `!=`) imply that both operands must be of the same category (both pointers or both numbers), prompting node merging.
(4) Constraint collection. For every addition or subtraction instruction, we record a constraint involving its three operands (two inputs and one result) as a triple of PN nodes.
(5) Rule-based inference. We iteratively apply inference rules (summarized in Table 1) to propagate type information across constraints. For instance, Rule 7 for subtraction states that if both the minuend and difference are pointers, the subtrahend must be numeric—reflecting the semantics of pointer subtraction.

Due to node merging in step (iii), multiple syntactic values may map to the same PN node, establishing an equivalence relation over types. We exploit this to derive additional inference rules based on type equality, as shown in Table 2. For example, If $X = Y$ in an addition $X + Y = Z$, then both must be numeric, since adding two pointers is semantically invalid. If $X = Z$, then Y must be numeric, as pointer arithmetic preserves the pointer nature of the base operand. These equality-aware rules can further enhance type resolution.

Table 2: Additional inference rules for addition and subtraction using equality relation.

Rule	$X + Y = Z$	$X - Y = Z$
1	$X = Y$ $\rightarrow$ X : Num, Y : Num	$Y = Z$ $\rightarrow$ X : Num, Y : Num
2	$X = Z$ $\rightarrow$ Y : Num	$X = Y$ $\rightarrow$ Z : Num
3	$Y = Z$ $\rightarrow$ X : Num	$X = Z$ $\rightarrow$ Y : Num

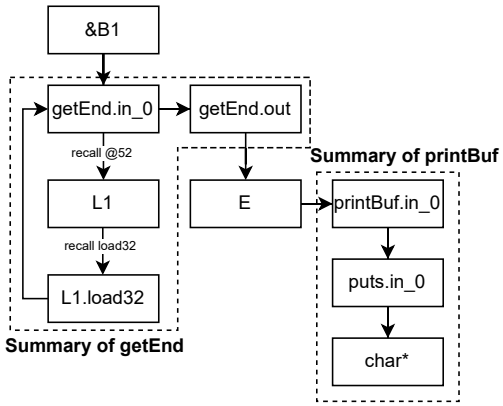


Figure 5: Constraint graph for main.

Constraint Graph Generation. Generating a constraint graph is the first step of the type recovery by Retypd. Specifically, each value in the program is mapped to a node in the constraint graph. Then we create edges according to the constraints posed by the instructions. For assignment instructions, we create subtype relationships without edge labels. For load/store instructions, we create an edge labeled as "recall load/store", pointing from the address value to the value in memory. For pointer addition edges, we create a "recall offset" edge, pointing from the pointer operand of the addition to the result of the addition. For example, in Figure 5, the constraint graph corresponding to the code in Figure 4a is displayed. The `getEnd.in_0` represents the first parameter `L` of `getEnd`. The operation `L=L->next` in the `getEnd` function includes member access (pointer addition, "recall edge @52"), memory loading ("recall load32 edge"), and a subtype edge.

Bottom-up Summary Generation & Top-Down Type Propagation. Similar to other summary-based full-context-sensitive static analyses [28], Retypd consists of a bottom-up summary generation step and a top-down type propagation step. Specifically, the bottom-up summary generation involves extracting the summary for each function, which indicates the impact of the callee on the caller's type. Since the generation of each function's summary depends on the summaries of other functions it calls, we traverse the call graph (with SCCs collapsed) bottom-up to generate the summaries. For

example, the simplified constraint graph of the main function in Figure 4a is shown in Figure 5. When generating the summary for the main function, the summaries of the callees (including `getEnd` and `printBuf`) are first introduced based on the initial constraint graph. Then, the algorithm described in Retypd is used to simplify the constraint graph, which is stored as the summary of the current function.

As for the top-down type propagation, it involves finding all the types passed by its callers and merging these types to derive the most precise parameter type for each function, which is used for the final type computation. For example, in the Figure 4a example of the `printBuf` function, it is reasonable to consider the type of its first parameter as `char*`. However, during the top-down process, it can be inferred that all calls to `printBuf` pass in the type `linkedBuf*`, which is a subtype of `char*` according to width subtyping. Therefore, the parameter of `printBuf` is set to the more precise type `linkedBuf*`.

We underline such an inter-procedural analysis step is optional. Although it enables type information propagation across the whole program, it also dramatically increases the complexity. Its intra-procedural analysis complexity is $O(n^3)$ with respect to the number of instructions, and in the worst case, polymorphic type functions can introduce exponential complexity with the depth of calls. As shown in § 4.1, it may require more than 24 hours for a relative large program. By disabling this step and restricting the analysis scope to the intra-function level, we could achieve a balance between effectiveness and performance. In such cases, structures used inside a function can still be identified, but member hints resulting from operations outside the function are no longer taken into account. As for the full-context-sensitivity without a call-stack depth limitation, it may lead to infinite recursion in the presence of cycles in the call graph, i.e., loops. To address this, Retypd disables context sensitivity within cycles by collapsing strongly connected components (SCCs) in the call graph into single nodes. This transformation produces a directed acyclic graph, only on which the full context-sensitive analysis is then applied.

Sketches Generation and C Type Conversion. Based on the final constraint graph obtained from the previous step, processing through Retypd can yield graph based type representations, referred to as *Sketches*. Then, by collecting pointer offset edges and subsequent memory read/write edges, we can recover custom structure types and generate type declarations, thereby converting the Sketch graph into C types. In the example from Figure 5, without inter-procedual type analysis, a decompiler would not know the specific types of `B1`. After including type summary from callee, we can follow the pointer offset edge "recall @52" and load edge "recall load32", and deduce that `B1` has a 32-bit structure member at offset 52, thus setting `B1` as a structure pointer type.

3.2.3 Stack SROA and Post-optimization. Since the previous type inference steps do not modify the IR, the stack space allocation for functions retains the form after stack pointer operations matching, which means the entire stack space of the function is viewed as a large structure containing all local variables. To make the decompiled result closer to the original code, we employ the method of Scalar Replacement of Aggregate (SROA) from compilation optimization to split the structure into local variables. After splitting, new optimization opportunities may be exposed in the code. Therefore, we apply compiler optimizations again to simplify the code.

3.3 Back End: C Code Generation

In order to convert the SSA and control flow graph-based intermediate representation into more readable C code, we need to address (*i*) the conversion of non-control flow instructions, and (*ii*) the structural analysis of control flow.

Memory SSA for Correct and Readable Load Handling. When generating C code from our SSA-based IR, we convert side effect instructions (e.g., stores) to statements, and we defer insertion of pure compute expressions until they are needed by other instructions, and cache these expressions in a map for later use. A key challenge arises in keeping loads semantically consistent with subsequent updates to the same memory location. For instance, consider:

```
1 %v1 = load i32, i32* %var1 ; %v1 mapped to var1
2 ; Solution 1: int var1.1 = var1;
3 store i32 4, i32* %var1 ; var1 = 4;
4 %v2 = add i32 %v1, 2 ; Incorrect: %v2 mapped to var1 + 2
```

If we naively map `%v2` as `var1+2`, we lose the fact that `%v1` should refer to the pre-store value of var1. A simple fix is to insert a temporary variable at the load, as shown in the second line. Introducing a fresh temporary for every load instruction would preserve semantics but at the cost of cluttering the generated code with too many temporaries, hurting readability.

To balance correctness and readability, we integrate Memory SSA [6] to identify situations where temporary variable is not required. Memory SSA addresses our queries regarding interactions between memory operations by identifying potential pointer aliasing situations through fast intra-procedural analysis. First, when we actually add statements or expressions to the AST, we traverse the entire AST to identify nodes corresponding to original load instructions. Second, For each such node (at current insertion point), we query Memory SSA to ask: "Has the memory location read by this load been modified since it was last loaded?". If not modified, then this usage is correct. If modified, we locate the original load's insertion point, introduce a temporary local there, and rewrite the AST node to refer to that temporary variable.

Structural Analysis of Control Flow. Usually IR adopts a representation based on control flow graphs. To convert it into a representation in C language based on control flow statements, we need to employ decompilation structural analysis algorithms. Existing research on structure analysis is already quite mature [11, 13, 18, 40]. We implemented a semantic-preserving structural analysis algorithm [11, 13] to ensure that the generated code is semantically consistent with the control flow of the IR.

4 Implementation & Evaluation

Implementation. We implemented three main modules as separate reusable projects. Specifically, the front end is implemented with approximately 3K lines of C++ code, in which the WebAssembly code parsing relies on the WebAssembly Binary Toolkit (WABT) project. The middle end uses LLVM IR as the intermediate language and implements the low-level pattern matching and type recovery algorithms with about 9K lines of C++ code. The back end generates

C code based on the Clang AST, using approximately 6K lines of C++ code.

Experiment Setup. All experiments were conducted on a computer with 64GB of memory, two NVIDIA GeForce RTX 2080 Ti graphics cards, and an Intel i9 9900 CPU, running Ubuntu 22.04.5.

Benchmark. Our dataset consists of the Juliet dataset [2] and Howard dataset [32]. the Juliet dataset contains 5,241 samples from the Juliet Test Suite 1.3 C code vulnerability dataset, which were selected by a prior study on porting code to WebAssembly [36]. Each of these samples contains 3–6 simple functions, making them relatively small in code size. The Howard dataset comprises five substantial programs, *i.e.,* `fortune`, `grep`, `gzip`, `lighttpd`, and `wget`, with source code sizes ranging from 2K to 46K lines of C code. We used the latest release versions of these projects as of October 2025.

We compiled all samples into WebAssembly using wasi-sdk-20 [10], adding relevant compilation options (like `-no-standard-libraries` and `-fno-builtin`) to avoid introducing unnecessary library functions. For the Howard dataset, some relied on features not supported by the compilation toolchain. We addressed this by introducing additional header files and avoid linking to the actual libraries to allow compilation to proceed. The resulting WebAssembly binaries, while suitable for decompilation and static analysis, cannot be actually executed. We obtained two different binaries for each project with `-O3` and `-O0` compilation options.

Baselines. To the best of our knowledge, there are only three available WebAssembly decompilation tools as follows. (*i*) **ghidra-wasm-plugin** [4]: Ghidra does not natively support WebAssembly, but the community-developed ghidra-wasm-plugin enables WebAssembly decompilation in Ghidra by converting WebAssembly into the P-code intermediate representation used by Ghidra's decompiler. (*ii*) **WaDec** [31]: WaDec is a WebAssembly bytecode-to-C code solution based on fine-tuned large language models. (*iii*) **WasmDec** [1]: WasmDec is the most starred WebAssembly decompiler on GitHub. It simply converts WebAssembly into the "low-level" C code without further stack pointer handling or type recovery.

Evaluation. In this work, we aim to answer the following research questions (RQs):

- **RQ1:** How efficient is NotDec compared with baselines?
- **RQ2:** How effective is NotDec compared with baselines?
- **RQ3:** How important is the optimization and type recovery phase mentioned in § 3.2 for decompilation?

To answer RQ1, we measure success rates, runtime, and memory consumption of NotDec and three baselines (WasmDec, Ghidra, WaDec). For RQ2, we evaluate effectiveness from three aspects: recompilation success, line of code (LoC), and type recovery accuracy, measured by two metrics: the number of correctly reconstructed structs, and the number of struct member accesses recovered from low-level pointer operations. For RQ3, we perform an ablation study with four NotDec variants on Juliet samples, to show the importance of the optimization and type recovery phase.

4.1 RQ1: Efficiency

To evaluate the efficiency, we decompile samples in both datasets using NotDec and three baselines. We count the number of successful decompilations for each tool and also record the time and memory consumption. We set a 5-minute and 24-hour timeout for samples in Juliet and Howard dataset, respectively.

Table 3: Statistics on 5,241 samples from Juliet C dataset.

Tool	# Success	Avg. Time (s)	Avg. Memory (MB)
NotDec	5,241	1.73	119.13
Ghidra	5,241	27.62	572.14
WaDec	1,940	108.11	9,890.45
Wasmdec	5,241	0.02	6.46

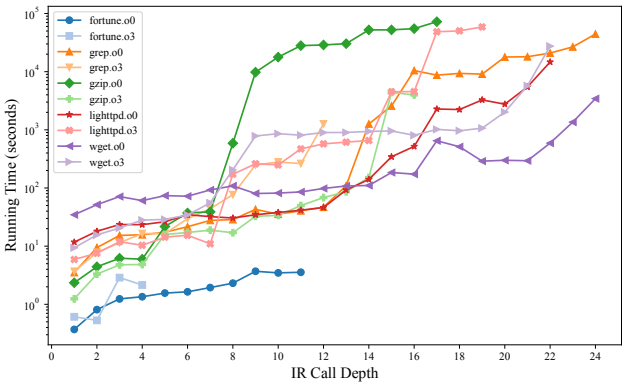


Figure 6: The time spent on inter-procedural type recovery analysis with the increase of IR call depth on the Howard dataset.

Results on Juliet. As shown in Table 3, we can observe that WaDec only successfully decompiles 1,940 samples. It gets stuck and does not output the final result for all the remaining 63.0% (3,301) samples, reaching the timeout. After an investigation, we figure out that it is related to the phenomenon where large language models fall into endless repeated sequence output [3].

For memory and time consumption, Wasmdec consumes the least memory and time (6.46 MB, 0.02s) because it simply performs a conversion of WebAssembly instructions, lacking in-depth analysis. NotDec consumes more memory and time (119.13 MB, 1.73 s) because of code optimization and type inference algorithms. As for Ghidra, it consumes more memory and time (572.14 MB, 27.62s). This is because Ghidra exhibits a longer cold-start time due to its full-fledged disassembly and analysis framework written in Java that supports multiple architectures. In contrast, NotDec is written in C++ and only contains decompiler and WebAssembly analysis logic, leading to its high efficiency. For WaDec, as it is based on a fine-tuned LLMs, the average peak memory usage is approximately 9.9 GB, while it also requires about 8.1 GB of GPU memory.

Results on Howard. The Howard dataset results are shown in Table 4. As we can observe, with `-O3` enabled, the number of instructions in binaries decreases to 29%–41% of the original, leading to reduced time and memory consumption across all tools. NotDec requires over 24 hours for six cases under unoptimized (`-O0`) conditions due to the high complexity of its adopted inter-procedural type recovery. To address this, we introduced NotDec$_F$, which restricts

Table 4: Average time and memory consumption on samples (`-O0` and `-O3` enabled) from Howard dataset.

Benchmark	#Inst in binary	Avg. Time (s)					Avg. Memroy (MB)				
		NotDec	NotDec$_F$	Ghidra	WaDec	WasmDec	NotDec	NotDec$_F$	Ghidra	WaDec	WasmDec
fortune (O0)	13,653	4.08	3.25	23.25	1,431.54	0.01	193.70	99.84	1,017.85	10,523.44	7.91
fortune (O3)	4,797	2.07	2.38	12.44	145.83	0.02	122.29	101.00	730.70	10,517.71	7.56
grep (O0)	245,806	51,092.00	68.05	522.53	8,508.00	0.10	13,397.80	452.22	2,492.05	10,521.06	27.82
grep (O3)	93,227	>24h	46.81	45.72	15,897.00	0.04	4,815.32	510.36	907.94	10,523.42	15.62
gzip (O0)	124,327	>24h	39.65	1,665.50	3,490.40	0.05	6,325.64	247.40	14,244.22	10,527.30	17.08
gzip (O3)	51,279	3,657.00	29.35	55.43	4,230.00	0.03	2,010.41	297.44	1,735.79	10,523.79	11.12
lighttpd (O0)	484,209	>24h	141.06	630.11	31,747.00	0.19	20,617.43	898.55	7,344.68	10,527.17	48.91
lighttpd (O3)	141,083	>24h	69.89	55.43	>24h	0.06	3023.89	795.36	2,194.95	10,515.05	20.04
wget (O0)	433,082	>24h	129.01	1,550.37	25,069.00	0.17	8152.79	730.00	14,742.57	10,520.45	44.56
wget (O3)	139,729	>24h	78.32	70.22	42,774.00	0.06	8,645.49	730.28	2,472.37	10,514.49	19.82

type recovery to the intra-procedural level. For `-O3` cases, NotDec$_F$ and Ghidra exhibit comparable decompilation times with NotDec$_F$ consuming 43.77% to 90.19% less memory. With optimizations disabled (`-O0`), NotDec$_F$ requires only a small fraction of Ghidra's time (ranging from 2.3% on gzip to 22.7% on lighttpd). We attribute this advantage to the compilation optimizations applied by NotDec prior to decompilation (see § 3.2.1), which reduce the input size for subsequent code analysis. The results for WasmDec and WaDec remain consistent with previous observations. WasmDec, performing only simple conversions of WebAssembly instructions, continues to scale effectively to larger binaries. WaDec's decompilation time correlates with the scale of input and output and depends on GPU performance. However, we still observe occasional instances of endless repeated sequence output on certain functions, leading to unstable final decompilation times.

Complexity of the Type Recovery Algorithm. To further depict and analyze the time complexity of the inter-procedural type inference algorithm employed by NotDec, *i.e.,* Retypd mentioned in § 3.2.2, we examine the relationship between the time cost of type inference and call depth, as shown in Figure 6. Since the y-axis is a logarithmic axis, we can easily observe the exponential relationship between them. Moreover, when reaching a particular depth, the analysis time even increases again dramatically (like `gzip-O0` from depth 7 to 9). This is because type summaries from deeper functions continuously accumulate summaries from callees. When a function calls multiple complex callees simultaneously, the constraint graph size expands several-fold, causing the original $O(n^3)$ algorithm to exhibit exponential time growth, eventually reaching the timeout we set in the experiment.

RQ1 Answer: *NotDec demonstrates high efficiency on the Juliet dataset (1.73 s & 119.13 MB) compared with Ghidra (27.62 s & 572.14 MB). On the Howard dataset, while full NotDec faces scalability challenges, its intra-procedural variant NotDec$_F$ achieves a balance between time and decompilation capability, particularly showing advantages over Ghidra in processing unoptimized binaries, using less than one-quarter of Ghidra's execution time and less than half its memory consumption.*

4.2 RQ2: Effectiveness

To comprehensively evaluate decompilation effectiveness, we assess 3 key aspects: recompilation success (syntactic correctness), Lines of Code (LoC, for conciseness), and type recovery accuracy. Specifically, we first measure recompilation success by compiling the decompiled C code with GCC. Next, we assess code conciseness through LoC measurement, following established practices in prior work [11, 40]. Finally, we assess type recovery accuracy by counting the number of struct variable and struct member accesses recovered. Note that all baselines are not robust enough to be adopted directly. We discuss our fixes on them in § 6.

Table 5: Recompilation success rate and average line of code on 5,241 samples from the Juliet dataset.

Tool	# Recompile Success Rate	Avg. LoC
NotDec	5,241/5,241 (100.00%)	87.60
Ghidra	2,408/5,241 (45.95%)	102.78
WaDec	216/1,940 (11.13%)	66.77
Wasmdec	1,546/5,241 (29.50%)	347.23

Results on Juliet. As shown in Table 5, NotDec achieves a 100% recompilation success rate on the Juliet dataset, demonstrating syntactic correctness in its decompiled results. WaDec achieves only 11.13% recompilation success, primarily due to syntactic issues such as undeclared variables and missing global variables. WasmDec exhibits a 29.50% recompilation success rate, with common syntax errors including incorrect insertion of break statements outside loop structures during WebAssembly-to-C conversion. Ghidra's recompilation success rate of 45.95% is limited by its primary design focus on interactive reverse engineering rather than full-binary decompilation and recompilation workflows, exhibiting many syntax errors, such as conflicting types for global variables like "`DAT_ram_0000093c`".

For LoC metrics, WaDec produces the lowest average line count (66.77). However, further investigation reveals frequent statement omissions and the presence of operations not present in the original code, which may be attributed to the common hallucination issues associated with large language model. NotDec achieves a much lower average LoC (87.60) compared to Ghidra (102.78). This reduction demonstrates the effectiveness of the in-depth type analysis described in § 3.2, which minimizes stack variables and transforms complex pointer operations into struct member accesses, thereby enhancing both code conciseness and readability. WasmDec, on

Table 6: Recompilation success and line of code counts for the Howard dataset. NotDec failed to decompile 6 programs within 24 hours, resulting in missing data.

Benchmark	#Inst in binary	Recompile Success					Line of Code				
		NotDec	NotDec$_F$	Ghidra	WaDec	WasmDec	NotDec	NotDec$_F$	Ghidra	WaDec	WasmDec
fortune (O0)	13,653	Y	Y	N	N	N	2,404	2,306	1,982	890	8,914
fortune (O3)	4,797	Y	Y	N	N	N	2,135	2,084	2,017	87	1,822
grep (O0)	245,806	Y	Y	N	N	N	50,914	38,663	27,145	6,062	157,039
grep (O3)	93,227	–	Y	N	N	N	–	55,758	36,257	7,788	35,331
gzip (O0)	124,327	–	Y	N	N	N	–	17,575	11,040	2,295	79,624
gzip (O3)	51,279	Y	Y	N	N	N	28,574	25,521	19,157	4,762	16,630
lighttpd (O0)	484,209	–	Y	N	N	N	–	77,165	56,883	14,995	309,847
lighttpd (O3)	141,083	–	Y	N	N	N	–	80,115	54,995	40,119	48,677
wget (O0)	433,082	–	Y	N	N	N	–	63,911	51,866	10,252	276,468
wget (O3)	139,729	–	Y	N	N	N	–	77,854	56,629	21,284	54,189

the other hand, yields the highest average LoC (347.23) as it fails to convert low-level operations (such as stack pointer subtraction) into equivalent C semantics, resulting in significantly less readable output.

Results on Howard. As shown in Table 6, NotDec and NotDec$_F$ are the only tools that consistently achieved successful recompilation across all five programs in the Howard dataset (for successfully decompiled cases). In contrast, all other baselines consistently fail to recompile the decompiled code, exhibited persistent syntactic errors, similar to those observed in the Juliet dataset. Regarding code conciseness, NotDec$_F$ produces higher LoC than Ghidra, which attributes to the declaration of recovered custom structures, which can be up to tens of thousands LoCs, like 19K for `lighttpd-O3`. WasmDec's output LoC directly correlates with the binary instruction count, yielding significantly more lines for unoptimized (`-O0`) binaries compared to optimized (`-O3`) versions. WaDec, while producing the lowest LoC count, frequently failed on larger functions due to "input too long" errors in LLMs, likely because it exceeded the model's maximum context window capacity.

Type Recovery Accuracy. To further evaluate the type recovery accuracy (detailed in § 4), we manually inspected 319 functions (269 functions from randomly selected 50 Juliet cases, and 5 randomly selected functions from each of the 5 Howard programs at both `-O0` and `-O3` optimization levels). For a complete comparison, we use results of NotDec$_F$ for the Howard dataset. We obtained the original source code for each function and compared it with the decompiled results.

On the Juliet dataset, functions are relatively simple. Among the 269 sampled functions, we identified only 6 struct-typed variables. Our approach successfully reconstructed all of them as proper struct types, whereas Ghidra type their pointers as plain integers. Furthermore, we correctly converted all 14 struct member accesses in Juliet into struct field accesses. Ghidra, in contrast, retained them as raw pointer arithmetic and dereference operations.

For the Howard dataset, there are 29 struct definitions and declarations that are accessed. NotDec$_F$ successfully reconstructed 24 (82.76%) of them. For the remaining five, we found that four of them lack any member accesses within the current function, and one global struct is flattened during decompilation, with its members treated as independent global variables. In contrast, Ghidra identified only 4 (13.79%) of the 29 structs, primarily recognizing standard types like `timespec` and `stat` through applying known library function signatures. Regarding struct member accesses, the source code contains 368 such accesses. NotDec$_F$ correctly recovered 314 (85.33%) of them. The remaining 54 accesses correspond to members of global structs that were flattened during decompilation, which, consequently, were directly mapped to the deconstructed global variables. Ghidra, on the other hand, recovered only 34 (9.24%) member accesses. Beyond the same 54 flattened global accesses, an additional 241 were left as raw pointer arithmetic and dereferences, and 39 were represented as array-indexed accesses instead of struct field references. Take the `kwsalloc` function from `grep` as an example, shown in Figure 7. NotDec$_F$ correctly reconstructed custom structure allocation at line 20, and converted those structure member accesses at line 23-26, while Ghidra could only represent these operations as low-level pointer arithmetics and array assignments at line 11-27.

Ghidra

```
1 int kwsalloc(undefined4 param1) {
2   char *pcVar1;
3   int param1_00;
4   undefined8 *puVar2;
5   int iVar3;
6   undefined8 *puVar4;
7   int iVar5;
8
9   param1_00 = xmalloc(s___directories_ram_00000553 + 1);
10  ...
11  *(undefined4 *)(puVar4 + 3) = 0;
12  puVar4[2] = 0;
13  puVar4[1] = 0;
14  *puVar4 = 0;
15  pcVar1 = s_binary_files_ram_00000546 + param1_00 + 10;
16  pcVar1[0] = '\x0e';
17  pcVar1[1] = '\0';
18  pcVar1[2] = '\0';
19  pcVar1[3] = '\0';
20  *(undefined4 *)(s_fixed_strings_ram_00000534 +
21      param1_00 + 0xc) = param1;
22  pcVar1 = s_fixed_strings_ram_00000534 + param1_00 + 4;
23  pcVar1[0] = '\0';
24  pcVar1[1] = '\0';
25  pcVar1[2] = '\0';
26  pcVar1[3] = '\0';
27  *(undefined4 *)(param1_00 + 0x34) = 0x7fffffff;
28  return param1_00;
29 }
```

NotDec-F

```
1 struct struct_35694 {
2     void *field_35695 /* at offset: 4 */ ;
3     ...
4     undef32 field_35706 /* at offset: 52 */ ;
5     char padding_35707[1280] /* at offset: 56 */ ;
6     undef32 field_35708 /* at offset: 1336 */ ;
7     char padding_35709[4] /* at offset: 1340 */ ;
8     void *field_35710 /* at offset: 1344 */ ;
9     char padding_35711[12] /* at offset: 1348 */ ;
10    undef32 field_35712 /* at offset: 1360 */ ;
11 };
12
13 struct struct_35694 *kwsalloc(void *_arg_0) {
14     void *_local_2_0;
15     struct struct_35694 *new_17129;
16     char *new_17235;
17     char *new_17308;
18     void *spec_store_select;
19     void *new_17250;
20     new_17129 = xmalloc(1364);
21     ...
22     memset(new_17250, 0, 28ULL);
23     new_17129->field_35712 = 14;
24     new_17129->field_35710 = _arg_0;
25     new_17129->field_35708 = 0;
26     new_17129->field_35706 = 2147483647;
27     return new_17129;
28 }
```

Figure 7: Decompilation results of `kwsalloc` from `grep`, where NotDec can recover fields of the custom structure used.

Table 7: Ablation study results in terms of efficiency, syntactic correctness, and code conciseness.

Tool	# Success	Avg. Time (s)	Avg. Memory (MB)	# Recompile Success	Avg. LoC
NotDec$_{d0}$	5,241	0.14	85.16	5,241	455.36
NotDec$_{d1}$	5,241	0.94	87.83	5,241	131.62
NotDec$_{d2}$	5,241	1.71	110.45	5,241	131.22
NotDec	5,241	1.73	119.13	5,241	87.60

RQ2 Answer: *NotDec achieves a 100% recompilation success rate, far exceeding Ghidra's 45.95% on the Juliet dataset. In terms of type recovery accuracy, the variant NotDec$_F$ can recover 82.76% structure variables and 85.33% struct member accesses while Ghidra can only recover 13.79% structure and 9.24% member accesses on samples from the Howard dataset.*

4.3 RQ3: Ablation Study

Regarding NotDec, we underline that only the optimization and type recovery module (see § 3.2) can be removed without affecting the functional integrity. Removing the code lifting module (see § 3.1) prevents the tool from parsing WebAssembly code, while removing the C code generation module (see § 3.3) results in the tool generating only an intermediate representation, making it unable to produce C language code.

To study the impact of different levels of code optimization and type recovery on decompilation results, we compare NotDec with the following three variants:

- NotDec$_{d0}$: without any code optimization and type recovery.
- NotDec$_{d1}$: only running the code optimization and low-level pattern matting (§ 3.2.1).
- NotDec$_{d2}$: performing type recovery while treating the stack as a whole structure.

We conducted the same experiment under the same settings of RQ1 and RQ2. Due to scalability of our type recovery algorithm, we perform ablation study only on the Juliet dataset.

Efficiency. In Table 7, all variants can successfully decompile all samples. For efficiency, the average decompilation time gradually increases with the introduction of optimizations and type recovery. The overhead of NotDec$_{d0}$ is the lowest, at only 0.14s, followed by NotDec$_{d1}$ (0.94s), while NotDec$_{d2}$ and NotDec have similar times of 1.71s and 1.73s, respectively. Memory consumption shows a similar trend: NotDec$_{d0}$ has the least memory usage (85.16 MB), while the complete type recovery NotDec uses the most (119.13 MB). This shows that type recovery, especially advanced analyses such as structure recognition, increases resource overhead.

Effectiveness. As in RQ2, we assess the variants from three aspects: syntactic correctness (via recompilability), code conciseness (by LoC), and type recovery accuracy. For syntactic correctness, all variants produce recompilable code for all 5,241 samples. In terms of code conciseness, the code generated by NotDec$_{d0}$ is the most verbose, averaging 455.36 LoC. The introduction of compilation optimizations in NotDec$_{d1}$ effectively eliminates redundant codes, significantly reducing the average LoC to 131.62. NotDec$_{d2}$ further simplifies the code through type analysis. However, by treating the entire function stack as a structure, it inserts many structure definitions, resulting in an average LoC similar to NotDec$_{d1}$ (131.22). The fully configured NotDec produces the most concise code, with an average of only 87.6 LoC, by splitting the stack. Overall, as optimization and type recovery strategies are gradually introduced, the conciseness of the decompiled code continue to improve.

Table 8: Ablation study results in terms of type recovery accuracy on sampled 269 functions of Juliet dataset.

Tool	# Recovered Structure Variable	# Recovered Member Access
NotDec$_{d0}$	0	0
NotDec$_{d1}$	0	0
NotDec$_{d2}$	275	1,905
NotDec	6	14

For type recovery accuracy, similar to RQ2, we count the number of recovered structure variables and member accesses on the sampled 269 functions, as shown in Table 8. Since the underlying WebAssembly instructions contain no structural information, NotDec$_{d0}$ recovers no structure variables or member accesses. NotDec$_{d1}$, which only applies additional code optimizations, exhibits similar behaviors. By interpreting the entire stack as a structure, NotDec$_{d2}$ creates a substantial number of structure variables (275) and member accesses (1,905). In contrast, NotDec further refines this approach by splitting the stack structure, converting most member accesses into direct variable accesses. Manual inspection of the source code for these 269 functions confirms that all recovered structure variables and member accesses are accurate and complete, with no omissions.

RQ3 Answer: *Through an ablation study, we found that code optimization combined with low-level pattern matching effectively reduces redundancy in binary code, achieving about 70% reduction in lines of code (LoC). The type recovery module contributes to the recovery of 100% of struct types and their associated accesses. Furthermore, deconstructing the stack into concrete variables yields an additional 30% reduction in LoC.*

5 Related work

WebAssembly Analysis Techniques. WebAssembly analysis resembles traditional binary analysis. Wassail[37] applies a compositional analysis technique to find vulnerabilities in a binary. Wasmati [12] employs Code Property Graphs (CPG) with pattern matching to find vulnerabilities. MINOS [26] detects cryptocurrency-related operations by extracting runtime features like instruction-type distributions. Eunomia [19] applies symbolic execution to WebAssembly code. However, Due to the complexity of analyzing linear memory, these approaches often face high complexity (e.g., symbolic execution's path explosion) or reduced accuracy.

Native Decompilation. The conversion of control flow graphs into control flow statements has been thoroughly researched [11, 13, 18, 40]. Recently, there has been an increasing number of works on decompiling binary code based on large language models [38, 43]. However, methods based on large models face issues of hallucination, and the results lack interpretability.

Binary Type Recovery For type recovery, previous type inference schemes [16, 22] often relied on the results of pointer analysis. Retypd [27] proposed an inter-procedural polymorphic type recovery scheme that does not require pointer analysis. There is also an attempt to simplify the Retypd algorithm [33]. In recent research, Osprey attempted to recover types in binary code based on probabilistic sampling [42]. BinPointer proposed an accurate pointer analysis at the binary level [21]. Manta [41] introduced hybrid-sensitive type analysis and implemented binary bug detection.

6 Discussion

Effectiveness Threats. Four factors may affect effectiveness. (*i*) Our type recovery approach does not support indirect calls, which compromises the accuracy of type inference. This limitation can be addressed by incorporating function pointer analysis to resolve the targets of indirect calls. (*ii*) Our approach relies on assumptions for recognizing low-level features, which, if unmet, threaten the effectiveness. These include the heuristic for stack allocation failing if the global stack pointer is not found, and pattern matching for libc functions (e.g., memset, memcpy) failing when they are inlined by LTO. Similarly, the heuristic for detecting variadic functions by matching argument arrays may be incomplete, requiring manual pattern updates. (*iii*) If PNDiff analysis does not have enough information to recognize all pointer arithmetic, it will result in the absence of hints for some structure members. This can be solved by allowing users to provide more information about which variable is pointer or number. (*iv*) The Memory SSA used in the backend code generation process may face over-approximation issues. However, we use Memory SSA solely to determine whether to insert temporary variables to hold the results of memory loads. In cases of uncertainty, our approach conservatively assumes potential aliasing and inserts the temporary variable, thereby avoiding any incorrect omission that could alter program semantics.

Efficiency of Type Recovery. Type recovery for WebAssembly faces challenges similar to those encountered in native binaries. Specifically, the Retypd type recovery algorithm we employed is not sufficiently fast. As shown in §4.1, inter-procedural analysis must be disabled to scale to large binaries. We underline that this inefficiency arises partly because Retypd implicitly performs computations analogous to those in pointer analysis. Explicitly decoupling and exposing pointer analysis results could therefore enable a broader range of downstream analyses that rely on points-to information. Furthermore, binary type recovery is fundamentally a form of type inference, a problem extensively studied in programming language (PL) theory. Recent advances in PL-type inference, particularly efficient algorithms supporting polymorphism and subtyping [15, 29], offer promising pathways toward more accurate and scalable type recovery techniques for binary code.

Choice of Decompiler IR. We opted to use LLVM IR as the intermediate representation for decompilation. This decision is supported by the fact that existing decompilers already adopt SSA-based design principles and optimization algorithms, such as Ghidra's P-Code and Binary Ninja's IL. Since semi-SSA forms are capable of representing both SSA and non-SSA code, they exhibit no fundamental divergence from LLVM IR in practice. Furthermore, we aim to faithfully capture the complete semantics of the binary in the IR, potentially through custom intrinsic functions or externally maintained mappings.

Other Source Programming Languages. We underline that decompiling the WebAssembly code compiled by other languages can largely follow the method proposed in this paper. Emerging languages such as Go and Rust also adopt a stack and heap-based memory allocation model. To support these languages, it is necessary to write targeted patterns for stack and heap memory allocation matching (§3.2.1), as well as to develop a dedicated backend to generate the corresponding syntax tree for them (§3.3).

Obfuscated WebAssembly Code. We assume that the input of NotDec is generated by a common compiler. Though NotDec can handle simple arithmetic obfuscation that can be recognized and reverted by the compiler's optimization, other complex code obfuscations can significantly negatively impact the results. For example, obfuscation of stack pointer operations can affect the recognition of stack memory allocation, thereby impacting the identification of local variables. We leave this as a future direction.

Fixes on Baselines. We applied lightweight patches to each decompiler to eliminate obvious syntax errors: for Ghidra (v10.3.2), we merged the pending upstream commit[2] to enable global-variable exports, then used simple pattern matching to correct string globals from the unsupported `string` type to valid C types (`char*`); for `wasmdec`, we initialized an uninitialized class flag to zero (preventing garbage-dependent missing semicolons) and resolved type-mismatch errors in load/store instructions; and for WaDec, `wasmdec`, and Ghidra outputs, we declared all called functions upfront to satisfy inter-function dependencies, additionally fixing WaDec's malformed `main` signature via a straightforward regex replacement.

7 Conclusion

This paper presents a complete WebAssembly-to-C decompilation pipeline NotDec, organized into three phases: (*i*) Code Lifting: parses Wasm bytecode and converts it into an SSA-based IR, exposing use-def chains. (*ii*) Optimization & Type Recovery: applies compiler optimizations and Retypd enhanced with pointer and numeric type differentiation analysis (PNDiff) to recover high-level C types, and splits stack with modified scalar replacement of aggregate (SROA) optimization. (*iii*) Control-Flow Structuring & C Code Generation: uses Memory SSA for correct and readable transformation of non-control-flow instructions, restructures high-level control constructs with semantic-preserving algorithm, and emits readable C code. Evaluation on 5,241 Juliet samples and five real-world programs demonstrates that NotDec is the only tool achieving 100% recompilation success across all binaries, while recovering 85.33% of struct member accesses compared to Ghidra's 9.24%, with 15.9× faster execution and 4.8× lower memory consumption.

Acknowledgments

This work was supported in part by the National Natural Science Foundation of China (grant No.62502414, and No.62572209) and the Hubei Provincial Key Research and Development Program (grant No. 2025BAB057).

[2] https://github.com/NationalSecurityAgency/ghidra/commit/7559acf5

References

[1] 2018. wwwg/wasmdec: WebAssembly to C decompiler. https://github.com/wwwg/wasmdec. Accessed: June 2025.

[2] 2021. SAC 2022 Dataset. https://figshare.com/articles/dataset/SAC_2022_Dataset/17297477?file=31953221. Accessed: July 2025.

[3] 2023. Small LLMs may stuck in a loop. https://news.ycombinator.com/item?id=37554095. Accessed: July 2025.

[4] 2024. nneonneo/ghidra-wasm-plugin: Ghidra Wasm plugin with disassembly and decompilation support. https://github.com/nneonneo/ghidra-wasm-plugin. Accessed: July 2025.

[5] 2025. IDA Decompiler. https://hex-rays.com/decompiler. Accessed: July 2025.

[6] 2025. MemorySSA - LLVM Documentation. https://llvm.org/docs/MemorySSA.html. Accessed: June 2025.

[7] 2025. NationalSecurityAgency/ghidra: Ghidra is a software reverse engineering (SRE) framework. https://github.com/NationalSecurityAgency/ghidra. Accessed: July 2025.

[8] 2025. Validation Algorithm - WebAssembly 2.0. https://webassembly.github.io/spec/core/appendix/algorithm.html. Accessed: July 2025.

[9] 2025. WebAssembly Design Rationale. https://github.com/WebAssembly/design/blob/af8e951/Rationale.md#why-a-stack-machine. Accessed: March 2025.

[10] 2025. WebAssembly/wasi-sdk: WASI-enabled WebAssembly C/C++ toolchain. https://github.com/WebAssembly/wasi-sdk. Accessed: July 2025.

[11] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. 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)*. USENIX Association, Philadelphia, PA, 361–378. https://www.usenix.org/conference/usenixsecurity24/presentation/basque

[12] Tiago Brito, Pedro Lopes, Nuno Santos, and José Fragoso Santos. 2022. Wasmati: An efficient static vulnerability scanner for WebAssembly. *Computers & Security* 118 (2022), 102745.

[13] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *22nd USENIX Security Symposium (USENIX Security 13)*. USENIX Association, Washington, D.C., 353–368. https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/schwartz

[14] Shangtong Cao, Ningyu He, Yao Guo, and Haoyu Wang. 2023. BREWasm: A General Static Binary Rewriting Framework for WebAssembly. In *Static Analysis: 30th International Symposium, SAS 2023, Cascais, Portugal, October 22–24, 2023, Proceedings* (Lisbon, Portugal). Springer-Verlag, Berlin, Heidelberg, 139–163. doi:10.1007/978-3-031-44245-2_8

[15] Stephen Dolan. 2017. *Algebraic subtyping*. BCS, The Chartered Institute for IT.

[16] Khaled ElWazeer, Kapil Anand, Aparna Kotha, Matthew Smithson, and Rajeev Barua. 2013. Scalable variable and data type detection in a binary rewriter. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 51–60. doi:10.1145/2491956.2462165

[17] WebAssembly Community Group. 2025. WebAssembly Specifications. https://webassembly.github.io/spec/. Accessed: March 2025.

[18] Andrea Gussoni, Alessandro Di Federico, Pietro Fezzardi, and Giovanni Agosta. 2020. A Comb for Decompiled C Code. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 637–651. doi:10.1145/3320269.3384766

[19] Ningyu He, Zhehao Zhao, Jikai Wang, Yubin Hu, Shengjian Guo, Haoyu Wang, Guangtai Liang, Ding Li, Xiangqun Chen, and Yao Guo. 2023. Eunomia: Enabling User-Specified Fine-Grained Search in Symbolically Executing WebAssembly Binaries. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (Seattle, WA, USA) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 385–397. doi:10.1145/3597926.3598064

[20] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An empirical study of real-world webassembly binaries: Security, languages, use cases. In *Proceedings of the web conference 2021*. 2696–2708.

[21] Sun Hyoung Kim, Dongrui Zeng, Cong Sun, and Gang Tan. 2022. BinPointer: towards precise, sound, and scalable binary-level pointer analysis. In *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction* (Seoul, South Korea) *(CC 2022)*. Association for Computing Machinery, New York, NY, USA, 169–180. doi:10.1145/3497776.3517776

[22] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[23] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 217–234. https://www.usenix.org/conference/usenixsecurity20/presentation/lehmann

[24] Daniel Lehmann and Michael Pradel. 2022. Finding the Dwarf: Recovering Precise Types from WebAssembly Binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 410–425. doi:10.1145/3519939.3523449

[25] Marius Musch, Christian Wressnegger, Martin Johns, and Konrad Rieck. 2019. New Kid on the Web: A Study on the Prevalence of WebAssembly in the Wild. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, Roberto Perdisci, Clémentine Maurice, Giorgio Giacinto, and Magnus Almgren (Eds.). Springer International Publishing, Cham, 23–42.

[26] Faraz Naseem Naseem, Ahmet Aris, Leonardo Babun, Ege Tekiner, and A Selcuk Uluagac. 2021. MINOS: A Lightweight Real-Time Cryptojacking Detection System.. In *NDSS*.

[27] Matthew Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. arXiv:1603.05495 [cs.PL] https://arxiv.org/abs/1603.05495

[28] Erik Matthew Nystrom. 2005. *FULCRA pointer analysis framework*. University of Illinois at Urbana-Champaign.

[29] Lionel Parreaux. 2020. The simple essence of algebraic subtyping: principal type inference with subtyping made easy (functional pearl). *Proc. ACM Program. Lang.* 4, ICFP, Article 124 (Aug. 2020), 28 pages. doi:10.1145/3409006

[30] Andreas Rossberg, Ben L Titzer, Andreas Haas, Derek L Schuff, Dan Gohman, Luke Wagner, Alon Zakai, JF Bastien, and Michael Holman. 2018. Bringing the web up to speed with WebAssembly. *Commun. ACM* 61, 12 (2018), 107–115.

[31] Xinyu She, Yanjie Zhao, and Haoyu Wang. 2024. WaDec: Decompiling WebAssembly Using Large Language Model. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*. 481–492.

[32] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures.. In *NDSS*.

[33] Ian Smith. 2024. BinSub: The Simple Essence of Polymorphic Type Inference for Machine Code. arXiv:2409.01841 [cs.PL] https://arxiv.org/abs/2409.01841

[34] Benedikt Spies and Markus Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. 1–10. doi:10.1109/CLEI53233.2021.9640153

[35] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. 2022. Security risks of porting C programs to webassembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22)*. Association for Computing Machinery, New York, NY, USA, 1713–1722. doi:10.1145/3477314.3507308

[36] Quentin Stiévenart, Coen De Roover, and Mohammad Ghafari. 2022. Security risks of porting C programs to webassembly. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (Virtual Event) *(SAC '22)*. Association for Computing Machinery, New York, NY, USA, 1713–1722. doi:10.1145/3477314.3507308

[37] Quentin Stiévenart and Coen De Roover. 2020. Compositional Information Flow Analysis for WebAssembly Programs. In *2020 IEEE 20th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 13–24. doi:10.1109/SCAM51674.2020.00007

[38] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286* (2024).

[39] Michael Van Emmerik. 2007. *Static Single Assignment for Decompilation*. PhD Thesis. The University of Queensland. doi:10.14264/158682

[40] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS*. Citeseer.

[41] Chengfeng Ye, Yuandao Cai, Anshunkang Zhou, Heqing Huang, Hao Ling, and Charles Zhang. 2025. Manta: Hybrid-Sensitive Type Inference Toward Type-Assisted Bug Detection for Stripped Binaries. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4* (Hilton La Jolla Torrey Pines, La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 170–187. doi:10.1145/3622781.3674177

[42] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. 2021. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 813–832.

[43] Muqi Zou, Hongyu Cai, Hongwei Wu, Zion Leonahenahe Basque, Arslan Khan, Berkay Celik, Antonio Bianchi, Dongyan Xu, et al. 2025. D-LiFT: Improving LLM-based Decompiler Backend via Code Quality-driven Fine-tuning. *arXiv preprint arXiv:2506.10125* (2025).