# О невозможности реассемблирования PE-бинарников без декомпиляции

*Строгое доказательство, иерархия трудностей и связь с type inference*

---

Z.ai

Ноябрь 2026

*Препринт. Общая теория реверс-инжиниринга PE-формата.*

# Содержание

**1 Аннотация и введение** **2**

**2 Часть I. Строгое доказательство невозможности реассемблирования без декомпиляции** **2**

2.1 Определения и постановка задачи . . . . . 2
2.2 Лемма 1. Неопределимость роли байта (code vs data) . . . . . 3
2.3 Лемма 2. Проблема покрытия при динамически вычисляемых указателях . . . . . 3
2.4 Лемма 3. Проблема размеров и типов данных . . . . . 4
2.5 Главная теорема . . . . . 4
2.6 Роль декомпиляции и природа ИИ-подхода . . . . . 5
2.7 Замечания о расширениях PE (CFG-таблицы и т. п.) . . . . . 5
2.8 Вывод по Части I . . . . . 5

**3 Часть II. Иерархия трудностей** **6**

3.1 Формализация уровней . . . . . 6
3.2 Уровень $C_0$: статические прямые ссылки . . . . . 6
3.3 Уровень $C_1$: таблицы переходов с известными границами . . . . . 6
3.4 Уровень $C_2$: динамически вычисляемые указатели . . . . . 7
3.5 Уровень $C_3$: self-modifying code (SMC) . . . . . 7
3.6 Уровень $C_4$: виртуализированные и упакованные бинари . . . . . 8
3.7 Строгая иерархия . . . . . 8
3.8 Практическое следствие . . . . . 9

**4 Часть III. Связь с type inference** **9**

4.1 Почему типы — необходимый компонент $P$ . . . . . 9
4.2 TIE: constraint-based type recovery . . . . . 9
4.3 retypd: subtyping-based типизация . . . . . 10
4.4 Неразрешимость type inference в общем случае . . . . . 10
4.5 Почему неразрешимость type inference — частный случай основной теоремы . . . . 11
4.6 Практическое ограничение: soundness vs completeness . . . . . 11
4.7 Итоговое замыкание Части III . . . . . 11

**5 Синтез и заключение** **11**

5.1 Сводка результатов . . . . . 11
5.2 Расширенная теорема . . . . . 12
5.3 О природе ИИ-декомпиляции . . . . . 12
5.4 Ограничения и перспективы . . . . . 13
5.5 Финальный вывод . . . . . 13

# 1. Аннотация и введение

В работе рассматривается задача реассемблирования PE-бинарников — восстановления исходного представления бинарного файла, эквивалентного исходному, по его скомпилированному виду. Утверждается и доказывается, что для произвольного PE-бинарника реассемблирование невозможно без предварительной декомпиляции, понимаемой как построение семантической модели бинарника, включающей граф потока управления (CFG), граф потока данных (DFG), типы данных, инварианты и процедурные абстракции.

Специфика PE-формата делает задачу принципиально сложной. Во-первых, формат не кодирует границы инструкций и типы данных: байты секции `.text` могут быть как кодом, так и данными, и различение этих двух ролей по самому формату невозможно (проблема *code vs data*). Во-вторых, формат не обеспечивает покрытия: для построения CFG требуется знать все цели непрямых переходов, но часть этих целей вычисляется в рантайме и не закодирована статически. В-третьих, даже когда ссылка известна, размер данных по указателю, хранящемуся в ссылке, также не закодирован: указатель хранит только адрес, но не размер и не тип объекта. Наконец, на самом сложном уровне указатель может вычисляться в динамике, делая статический анализ адресов принципиально неполным.

Единственное решение, соответствующее природе задачи, — это декомпиляция, понимаемая не как трансляция байт на язык высокого уровня, а как *сжатие* исходной информации до алгоритма, абстракции или неразложимой модели. Декомпиляция строит промежуточную модель $M$, из которой затем генерируется целевое представление. Эта работа формализует данное утверждение и доказывает его строго.

Структура работы: в Части I формулируются определения и доказывается основная теорема невозможности. В Части II строится иерархия трудностей $C_0 \subsetneq C_1 \subsetneq C_2 \subsetneq C_3 \subsetneq C_4$, показывающая, что на каждом уровне сложности статический анализ становится всё менее разрешимым. В Части III доказывается, что восстановление типов данных — необходимый компонент корректной партиции — само по себе неразрешимо в общем случае. В Части IV результаты объединяются в расширенную теорему.

# 2. Часть I. Строгое доказательство невозможности реассемблирования без декомпиляции

## 2.1. Определения и постановка задачи

Введём необходимые определения. Бинарник $B$ — последовательность байт $\{b_0, \ldots, b_{n-1}\}$ с разметкой PE-формата: заголовки, секции, директории, точки входа. Реассемблирование $R$ — построение $B' = R(\text{repr}(B))$ из промежуточного представления (например, ассемблерного листинга), такого что $B' \equiv B$ побайтово, либо $B' \sim B$ семантически эквивалентно. Декомпиляция $D$ — восстановление семантической модели $M = D(B)$, включающей CFG, DFG, типы данных, процедурные абстракции, инварианты циклов, абстракции памяти.

**Определение 1.** Партиция $P$ бинарника $B$ — разбиение $B$ на семантические единицы: границы инструкций, границы и типы объектов данных, выравнивающий padding, отношения ссылок. Иначе: $P$ есть семантическая типизированная декомпозиция $B$.

Утверждается: $R$ невозможен без предварительного $D$. Доказательство строится через серию лемм, каждая из которых показывает независимый источник неразрешимости.

### 2.2. Лемма 1. Неопределимость роли байта (code vs data)

**Лемма 2.** *По бинарнику $B$ в общем случае неразрешимо определить, является ли $b_i$ кодом (исполняемой инструкцией) или данными.*

*Доказательство.* Сведение к проблеме останова. Пусть $\mathcal{M}$ — машина Тьюринга, $w$ — вход. Построим исходник на C:

```
if (halts(M, w)) {
    ((void(*)())(base + i))();   // b_i executed as instruction
} else {
    printf("%d", base[i]);       // b_i read as data
}
```

Компилируем в PE-бинарник $B_{\mathcal{M},w}$. Если бы существовал разрешитель $A(B, i) \to \{\text{code}, \text{data}\}$, то, применив $A$ к $B_{\mathcal{M},w}$ по смещению $i$, мы решили бы проблему останова $\mathcal{M}$ на $w$. Противоречие. □

**Следствие 3.** *Статическая партиция $P$ (границы инструкций, типы данных) невычислима по $B$ в общем случае. Это снимает надежду на "синтаксическое" реассемблирование.*

### 2.3. Лемма 2. Проблема покрытия при динамически вычисляемых указателях

Для построения CFG нужен полный набор целей непрямых переходов. Часть целей вычисляется в рантайме:

```
mov  eax, [base + (((x * 9E3779B9h) >> 16) AND 0FFh)]
jmp  eax
```

Множество достижимых целей зависит от $x$ во время исполнения. Статическое перечисление этого множества требует абстрактной интерпретации по области значений $x$. Арифметика над адресами Тьюринг-полна: можно закодировать произвольное вычисление в индексном выражении. Поэтому точное перечисление в общем случае эквивалентно решению произвольной задачи вычисления.

**Лемма 4.** *Перечисление достижимых целей непрямых переходов в произвольном PE-бинарнике неразрешимо.*

*Доказательство.* Сведение к задаче Coverage. Пусть $\mathcal{M}, w$ — машина Тьюринга и вход. Закодируем $\mathcal{M}$ так, что переход по адресу $a_k$ соответствует $k$-му шагу $\mathcal{M}$. Тогда $\mathcal{M}$ останавливается на $w$ тогда и только тогда, когда некоторый адрес $a_k$ достижим. Если бы существовал разрешитель "достижим ли $a_k$", мы решили бы проблему останова. □

**Следствие 5.** *Полный CFG по $B$ в общем случае невычислим, и значит, complete coverage (все ссылки на код/данные) недостижим статически.*

### 2.4. Лемма 3. Проблема размеров и типов данных

Указатель $p$ хранит только адрес. Размер и тип объекта по `*p` в указателе не закодированы: байты по `*p` — это просто байты. То, что это `struct { int a; char b[20]; }` или `int[6]`, можно установить только из семантики использования: какие смещения, в каком порядке и какой ширины читаются/пишутся кодом, который разыменовывает `*p`.

**Лемма 6.** *Корректное определение границ и типов данных по указателю требует семантической реконструкции, что равносильно декомпиляции $D$.*

*Доказательство.* По построению. $P$ требует знания размеров и структур данных для: корректного разбиения байт на объекты, различения padding и валидных данных, установления DFG-рёбер (какое поле куда течёт). Размер и тип объекта не закодированы в PE-формате. Единственный источник информации — семантика использования: какие смещения и какой ширины читаются и пишутся кодом. Восстановление этой семантики и есть type recovery — компонента декомпиляции $D$. □

### 2.5. Главная теорема

**Теорема 7.** *Верное реассемблирование $B' \equiv B$ произвольного PE-бинарника невозможно без предварительной декомпиляции.*

*Доказательство.* Доказательство по шагам.

**Шаг 1.** $R$ требует знания партиции $P$: границ инструкций, границ и типов данных, выравнивания, отношений ссылок. Иначе либо $B' \neq B$ побайтово, либо $B'$ семантически расходится с $B$ (отсутствующий padding, слияние/разделение объектов, неверные размеры косвенных обращений к памяти).

**Шаг 2.** $P$ не закодирована в PE-формате. Формат знает только секции и ограниченный набор директорий (import, export, resources, CFG-таблицы и т. д.). Никакой информации об объектном уровне — границы отдельных инструкций после декодирования, размеры и типы данных, рёбра ссылок — формат не несёт.

**Шаг 3.** По леммам 1–3, $P$ не вычислима по $B$ в общем случае:

- Лемма 1: неразрешимо разделение code/data.
- Лемма 2: неразрешим полный CFG при динамических указателях.
- Лемма 3: неразрешимы размеры/типы данных без семантики.

**Шаг 4.** Единственный источник $P$ — семантическая реконструкция: построение модели $M \supseteq P$ (CFG, DFG, типы, инварианты) методами абстрактной интерпретации, динамического исполнения, symbolic execution или, в случае ИИ-подхода, вероятностного вывода. Это и есть декомпиляция $D$.

**Заключение.** Следовательно, $R \Rightarrow D$: реассемблирование без декомпиляции невозможно. □

## 2.6. Роль декомпиляции и природа ИИ-подхода

Декомпиляция — это не перевод байт на язык высокого уровня. Это *сжатие* $B$ в абстрактную модель $M$:

$$B \xrightarrow{D} M \xrightarrow{G} B',$$

где $M$ содержит алгоритм, типы, потоки управления и данных, инварианты — то есть семантику, без которой $P$ не восстанавливается. $G$ — генерация (сборка) в целевой формат/язык/механику.

Прямой путь $B \to B'$ (побайтовое реассемблирование) требует информации, которой в $B$ нет: типов, исходной партиции, исходной структуры. Эта информация *физически отсутствует* в бинарнике — она была потеряна при компиляции. Единственный способ её восстановить — инференс по семантике использования, то есть $D$.

ИИ-декомпиляция здесь работает именно как компрессор: $B \to M$, где $M$ — неразложимая на байты, но эквивалентная по семантике модель. Это согласуется с тем, что ИИ сжимает исходную информацию до алгоритма, а не переводит байты.

## 2.7. Замечания о расширениях PE (CFG-таблицы и т. п.)

Современные расширения PE-формата, такие как CFG-таблицы (Control Flow Guard), маркируют *только* валидные цели непрямых вызовов — это защита от ROP-атак. Они не покрывают: прямой код (включая fall-through), данные, размеры и типы объектов, динамически вычисляемые указатели. CFG-таблицы — это “совет”, а не исчерпывающее описание $P$.

Более общий аргумент: любое расширение PE, претендующее на полноту описания $P$, должно закодировать партицию байт на инструкции/данные, типы и размеры данных, все рёбра CFG/DFG, отношения эквивалентности и инварианты. Но это в точности и есть приложение к бинарнику его декомпиляции $M$. То есть “бинарь + полное расширение” эквивалентно “бинарь + декомпиляция”. В таком случае реассемблирование тривиально, но формат перестаёт быть бинарником в исходном смысле — он становится двусторонним представлением, эквивалентным наличию исходников.

## 2.8. Вывод по Части I

Реассемблирование произвольного PE-файла без декомпиляции *невозможно* в строгом смысле: требуемая партиция $P$ не закодирована в формате и невычислима из $B$ (сводится к проблеме останова и к динамически вычисляемым ссылкам). Декомпиляция — единственный механизм восстановления $P$, и в этом смысле она является необходимым условием реассемблирования.

Утверждение сильнее: путь “декомпиляция $\to$ генерация” — это не один из возможных подходов, а *единственно возможный* для нетривиальных бинарников. Любой метод, претендующий на реассемблирование без декомпиляции, либо (а) вводит $D$ под другим именем, либо (б) работает только на ограниченном классе бинарников (например, со статическими ссылками и известной разметкой), что не противоречит теореме, а является её частным случаем.

# 3. Часть II. Иерархия трудностей

## 3.1. Формализация уровней

Введём меру сложности — *глубину косвенности* $d$ адресной ссылки. Это минимальное число разыменований памяти между исходной инструкцией и фактической целью:

$$d(\text{ref}) = \min\{k : \text{ref} = \sigma_0 \to \sigma_1 \to \cdots \to \sigma_k = \text{target}\},$$

где $\sigma_i$ — последовательные состояния адреса в цепочке разрешения. По $d$ введём классы $C_0, C_1, C_2, C_3, C_4$. Для каждого формулируются: модель адресации, разрешимость $P$, редукция к предыдущему уровню.

Утверждается, что $\{C_i\}$ образует строгое включение: $C_0 \subsetneq C_1 \subsetneq C_2 \subsetneq C_3 \subsetneq C_4$, и при этом трудность вычисления $P$ строго возрастает на каждом шаге.

## 3.2. Уровень $C_0$: статические прямые ссылки

**Модель.** `call 0x401000`, `jmp 0x401000`, `mov eax, [0x402000]` с immediate-адресом. Цель закодирована как константа внутри инструкции.

**Разрешимость $P$.** Алгоритм: линейный проход (linear sweep) + декодирование. $O(n)$, разрешимо.

**Ограничение.** Не работает, как только: код имеет перекрытие инструкций (overlapping instructions, x86 CISC); в теле функции есть jump-таблица; встречается непрямой вызов. То есть $C_0$ — это класс бинарей, где адресация полностью статична и выражается immediate-операндами. В реальном компилированном коде этого никогда не бывает, кроме тривиальных вырожденных случаев.

## 3.3. Уровень $C_1$: таблицы переходов с известными границами

**Модель.** `jmp [table + i*4]`, где `table` — массив адресов, лежащий в `.rdata`. Это классическая jump-таблица для `switch`.

**Трудность.** Цель не закодирована в инструкции — она лежит в данных. Необходимо определить, что `table` — действительно массив указателей, и определить *границы* массива (сколько в нём элементов).

**Лемма 8.** *Границы jump-таблицы по $B$ в общем случае невычислимы без анализа DFG.*

*Доказательство.* Компилятор генерирует `table` без указания размера; размер закодирован в guard-инструкциях: `cmp i, N; ja default`. Если $N$ — константа, восстановление возможно тривиально. Но $N$ может быть вычислено (`mov ecx, [size_var]; cmp i, ecx`), что превращает задачу в $C_2$. Пусть $N$ константно. Тогда мы знаем “потолок”, но не знаем, все ли элементы таблицы фактически достижимы и валидны. Компиляторы вставляют дубликаты `default` для padding; различить валидный указатель и паддинг по байтам нельзя (это и есть лемма 1). □

**Разрешимость.** Разрешимо при условии (а) константного размера и (б) возможности отличить код от данных (что само по себе неразрешимо — лемма 1). То есть $C_1$ *не разрешим* без предположений, в общем случае сводится к лемме 1.

### 3.4. Уровень $C_2$: динамически вычисляемые указатели

**Модель.** Цель перехода или доступа вычисляется арифметикой:

```
mov  eax, [base + hash(x) % N * 4]
jmp  eax
```

где `hash` — произвольная функция, `x` — значение, доступное только в рантайме.

**Лемма 9.** *Перечисление достижимых целей на $C_2$ неразрешимо.*

*Доказательство.* Сведение к задаче Coverage. Пусть $\mathcal{M}, w$ — машина Тьюринга и вход. Закодируем $\mathcal{M}$ так, что переход по адресу $a_k$ соответствует $k$-му шагу $\mathcal{M}$. Тогда $\mathcal{M}$ останавливается на $w$ тогда и только тогда, когда некоторый адрес $a_k$ достижим. Если бы существовал разрешитель "достижим ли $a_k$", мы решили бы проблему останова. □

**Методы.** Абстрактная интерпретация, symbolic execution, concolic testing. Все дают *супермножество* целей (over-approximation), никогда — точное множество. Недостижение полноты структурно.

**Почему это не "улучшение $C_1$".** На $C_1$ таблица — статический объект в памяти. На $C_2$ "таблица" может быть *виртуальной*: не существовать в памяти как массив, а вычисляться "на лету". Например, `hash(x)` возвращает адрес напрямую, минуя разыменование памяти. Тогда $d = 1$ для $C_1$, $d = 0$ для $C_2$ в смысле разыменований, но $C_2$ *сложнее*, потому что цель — чистое вычисление, и нет статического объекта для анализа.

Это опровергает наивную интуицию "меньше разыменований = проще". Сложность определяется не $d$, а *необходимостью символьного исполнения*.

### 3.5. Уровень $C_3$: self-modifying code (SMC)

**Модель.** Байты по адресу $a$ изменяются во время исполнения. Инструкция, декодированная при первом достижении $a$, может отличаться от инструкции при втором достижении.

**Структурное отличие.** На уровнях $C_0$–$C_2$ партиция $P$ хотя бы *стационарна*: существует истинная $P$ (пусть и невычислимая), не зависящая от момента исполнения. На $C_3$ это ломается:

$$P = P(t) \quad \text{— партиция зависит от момента декодирования.}$$

**Лемма 10.** *На $C_3$ не существует единой статической $P$, корректной для всех исполнений.*

*Доказательство.* Пусть $a$ — адрес. В начале $b_a =$ NOP, но после записи из другой инструкции $b_a =$ JMP evil. Любая статическая $P$, выбирающая `NOP` для $a$, некорректна для пути, проходящего через запись. Любая $P$, выбирающая `JMP`, некорректна для пути без записи. Статическая $P$ не существует; существует лишь $P(t)$ или $P(\text{path})$. □

**Следствие.** Любой статический декомпилятор *по построению* некорректен на $C_3$. Корректность требует моделирования времени/состояний. Это выводит задачу за пределы статического анализа — нужна символьная модель исполнения, отслеживающая порядок записи/чтения.

**Замечание о protection.** SMC часто используется в packer'ах и протекторах (UPX, Themida), в JIT-компиляторах (V8, .NET CLR), в гипервизорах. На уровне ОС страница помечена RWX; факт записи и исполнения ничего сам по себе не говорит о том, что код менялся.

### 3.6. Уровень $C_4$: виртуализированные и упакованные бинари

**Модель.** Исходный x86-код транслирован в байт-код виртуальной машины VM. x86-секция бинаря содержит: (1) интерпретатор VM (dispatcher + handlers); (2) байт-код VM в данных; (3) опционально контекст VM — регистры VM, стек VM.

**Структурное отличие.** На $C_0$–$C_3$ целевой ISA бинаря совпадает с ISA исполнения. На $C_4$ это не так: ISA бинаря — это x86, но семантика закодирована в другой ISA (VM bytecode). Стандартный декомпилятор x86, применённый к $C_4$, декомпилирует интерпретатор, а не исходный код.

**Лемма 11.** *Восстановление исходной семантики на $C_4$ требует: (i) выделения интерпретатора VM; (ii) восстановления таблицы опкодов; (iii) реконструкции семантики каждого опкода (декомпиляция handler'а); (iv) применения полученной модели к байт-коду — то есть фактического исполнения интерпретатора в абстрактной форме.*

*Доказательство.* По построению. Семантика байт-кода = композиция семантик handler'ов. Чтобы восстановить её, нужно знать семантику каждого handler'а. Эта семантика — произвольный фрагмент x86-кода, который сам может относиться к $C_2$ или $C_3$ (в протекторах handler'ы обфусцированы). □

**Иерархический парадокс.** Бинарь $C_4$ содержит внутри себя байт-код VM, который *сам* может использовать конструкции $C_0$–$C_3$. То есть:

$$\text{сложность}(C_4) = \text{сложность}(C_i) + \text{метауровень VM},$$

где $i$ — максимальный уровень, достигаемый внутри handler'ов. В протекторах (Themida, VMProtect) handler'ы намеренно пишутся с $C_2$–$C_3$, что делает итоговую сложность $C_4 \circ C_3$.

### 3.7. Строгая иерархия

**Теорема 12** (иерархии)**.** *Классы $C_0 \subsetneq C_1 \subsetneq C_2 \subsetneq C_3 \subsetneq C_4$ образуют строгое вложение, и соответствующая трудность вычисления $P$ строго возрастает.*

*Доказательство.* $C_0 \subsetneq C_1$**:** $C_0$ разрешает $P$ за $O(n)$; $C_1$ уже требует анализа guard'ов и проверки data/code (лемма 4). Бинари $C_1$, не лежащие в $C_0$: любые функции со `switch` $> 4$ case (компиляторы их всегда превращают в таблицы).

$C_1 \subsetneq C_2$**:** $C_1$ имеет хотя бы статический объект-таблицу; $C_2$ не предполагает существования статического объекта. Бинари $C_2 \setminus C_1$: любой код с вычислением адреса по хэшу, любые vtable в C++ с devirtualization, большинство Rust-диспетчеров.

$C_2 \subsetneq C_3$**:** $C_2$ стационарен ($P$ не зависит от времени), $C_3$ — нет (лемма 6). Строгое различие — по существованию статической $P$.

$C_3 \subsetneq C_4$**:** $C_3$ сохраняет целевую ISA; $C_4$ вводит уровень мета-ISA. Это *не сводимо*: для работы с $C_4$ необходимо сначала декомпилировать интерпретатор VM, а затем применить результат. Каждый шаг сам по себе задача уровня $C_i$.

Каждый уровень строго труднее предыдущего, потому что каждый вводит *новый источник неопределённости*, отсутствующий на предыдущих:

- $C_0 \to C_1$: размер и валидность таблицы;

- $C_1 \to C_2$: отсутствие статического объекта; чистое вычисление;
- $C_2 \to C_3$: зависимость $P$ от времени;
- $C_3 \to C_4$: мета-уровень ISA.

□

### 3.8. Практическое следствие

На каждом уровне $C_i$ для $i \geq 1$ статический реассемблер либо: даёт неверный результат (теряет ссылки, путает code/data); либо прибегает к динамическому анализу, symbolic execution, декомпиляции, что и есть $D$. Чистых методов между $C_0$ и $D$ не существует — они либо опускаются в $C_0$ (соответственно, работают только на тривиальном подмножестве), либо поднимаются до $D$.

В пределе $C_4 \circ C_3$ единственный осмысленный подход — это полная семантическая реконструкция: декомпиляция интерпретатора, реконструкция VM, исполнение байт-кода в восстановленной модели, и только затем — генерация. Это в точности описанная в Части I схема $B \to M \to B'$.

## 4. Часть III. Связь с type inference

### 4.1. Почему типы — необходимый компонент $P$

Партиция $P$ из основной теоремы включает: границы инструкций (для этого нужен CFG) и границы и типы данных. Рассмотрим данные. PE-формат знает только, что секция `.data` содержит байты. Что именно там лежит — `int`, `struct`, `float`, массив указателей, комбинация thereof — формат не несёт.

Пусть по адресу $a$ лежат 24 байта. Возможные интерпретации:

- `int[6]`
- `struct { int x; char name[20]; }`
- `struct { void* p; long y; long z; }` с padding
- `double[3]`
- массив из 3 указателей на функции, чередующихся с padding

Все они совместимы с одними и теми же байтами. Различение возможно *только* по семантике использования: какие инструкции читают/пишут смещения, какой ширины, в каком порядке, как адреса вычисляются.

Восстановление типов из паттернов использования — это и есть *type inference на бинарниках*.

### 4.2. TIE: constraint-based type recovery

**Архитектура.** TIE (Type Inference on Executables, Lee et al., 2011) моделирует память в виде *slow memory*: каждая ячейка адреса рассматривается как отдельная переменная (без алиасинга на уровне модели). Из инструкций извлекаются ограничения:

- `mov dword [eax], 0` генерирует: $\tau(\text{eax}) = \text{ptr}(\tau_1)$, $\tau_1 \supseteq \text{int32}$;
- `mov eax, [ebx+4]` генерирует: $\tau(\text{ebx}) = \text{ptr}(\sigma)$, $\sigma \supseteq \text{struct}\{\ldots, f_4 : \tau(\text{eax}), \ldots\}$.

Ограничения собираются в систему, решаемая по типам Хиндли-Милнера с subtyping.

**Слабости.** Slow memory игнорирует алиасинг: если `eax` и `ebx` ссылаются на одну область, TIE теряет информацию. Не различает поля, разделённые разными смещениями, если они перекрываются. Только конечные типы (без рекурсивных типов структур).

**Сильные стороны.** Дисциплинированная формализация; разрешимая система ограничений; аудиторность.

### 4.3. retypd: subtyping-based типизация

**Архитектура.** retypd (Noonan et al., 2016) использует теорию подтипирования (subtyping) с явной решёткой типов. Ключевые идеи: введение *атомов типов* — например, `ptr.`$\leq$`4` (указатель с полем "что-то на смещении 4"), `int32.`$\leq$`width`; использование *графа подтипирования* вместо деревьев типов TIE; поддержка *рекурсивных типов* (`struct { int x; struct *next; }` для списков); поддержка *полиморфизма* через параметрические типы.

**Сильные стороны.** Выразительность значительно выше TIE; корректно работает с циклическими/рекурсивными структурами; полиморфизм позволяет разделять типы функций, использующих разные инстанциации.

**Слабости.** Система ограничений сложнее; вычисление может не завершиться (см. ниже).

### 4.4. Неразрешимость type inference в общем случае

**Лемма 13.** *Задача восстановления типов в произвольном бинарнике неразрешима.*

*Доказательство.* Сведение к неразрешимости type inference с подтипами. Известно (Henglein, 1994): type inference для простого типизированного $\lambda$-исчисления с подтипами неразрешим. retypd именно такую систему и строит: типы формируют решётку, в которой подтипирование встречается повсеместно (int16 $\leq$ int32, struct { a: int } $\leq$ struct { a: int16 } и т. д.).

Для TIE ситуация ещё жёстче: хотя ограниченная система HM с подтипированием при определённых ограничениях остаётся разрешимой, в реальных бинарниках эта разрешимость теряется из-за: (1) алиасинга (нескольких указателей на одну память); (2) арифметики указателей (`ptr + n`), которая в типизированных языках запрещена; (3) self-references (рекурсивных типов структур); (4) union-типов, возникающих из `union` в C и из перекрывающихся доступов.

Каждое из этих условий достаточно, чтобы вывести систему за пределы разрешимого фрагмента HM.

**Сведение.** Пусть $\mathcal{M}, w$ — машина Тьюринга и вход. Закодируем $\mathcal{M}$ как C-программу, использующую `void*`-объединение: при шаге $i$ машина смотрит на тип данных, который определяется её собственным состоянием. Если бы существовал разрешитель типов в бинарнике, мы получили бы решение задачи об остановке $\mathcal{M}$ — по типу восстановленной структуры данных определяем, остановилась ли $\mathcal{M}$. $\square$

### 4.5. Почему неразрешимость type inference — частный случай основной теоремы

**Лемма 14.** *Восстановление типов данных по $B$ — необходимый компонент $P$.*

*Доказательство.* $P$ требует знания размеров и структур данных для: корректного разбиения байт на объекты; различения padding и валидных данных; установления DFG-рёбер (какое поле куда течёт). Без типов $P$ содержит только границы инструкций (если они даже восстановимы — лемма 1), но не объектную структуру. □

**Следствие 15.** *Если бы $R$ (реассемблирование) было возможно без $D$, это означало бы, что мы обошли type inference — противоречие с леммой 8.*

Это *усиливает* основную теорему: даже в предположении, что code/data разрешимо (лемма 1 снята), даже в предположении, что CFG полон (лемма 2 снята), задача всё ещё неразрешима из-за одной лишь type inference (лемма 8).

### 4.6. Практическое ограничение: soundness vs completeness

На практике системы type inference (TIE, retypd) выбирают одну из двух стратегий:

**Sound**: вычисляют надмножество типов. Каждое восстановленное отношение $\tau_1 \subseteq \tau_2$ гарантированно верно, но многие типы "размазываются" до `int` или `void*`. Практически бесполезно для $R$ (теряем конкретику).

**Complete**: вычисляют подмножество типов. Каждая найденная типовая аннотация точна, но многие поля остаются без типа. Практически бесполезно для $R$ (остаются пробелы в $P$).

Это классическая дилемма статического анализа, следующая из неразрешимости: нельзя одновременно быть sound и complete для неразрешимой задачи. TIE выбирает sound; retypd частично компромиссный. Ни один не даёт $P$ в строгом виде, требуемом для $R \equiv B$.

### 4.7. Итоговое замыкание Части III

Объединяя основную теорему (Часть I) с иерархией трудностей (Часть II) и неразрешимостью type inference (настоящая Часть), получаем расширенное утверждение: реассемблирование произвольного PE-бинарника без декомпиляции невозможно. Неразрешимость проявляется независимо через (i) лемму 1 — code vs data; (ii) лемму 2 — coverage при динамических ссылках; (iii) лемму 3 — размеры объектов данных; (iv) лемму 8 — типы данных. Каждое из условий (i)–(iv) *независимо* делает $R$ невозможным: даже снятие любого одного не снимает других.

## 5. Синтез и заключение

### 5.1. Сводка результатов

Настоящая работа устанавливает строгий результат: реассемблирование произвольного PE-бинарника без декомпиляции невозможно. Этот результат получен через три независимых линии рассуждения, которые в совокупности дают исчерпывающее доказательство.

**Линия 1 (Часть I): основная теорема.** Реассемблирование $R$ требует знания партиции $P$ — границ инструкций, границ и типов данных, выравнивания, отношений ссылок. Партиция $P$ не

закодирована в PE-формате и не вычислима по $B$ в общем случае. Невычислимость следует из трёх независимых лемм: (1) code vs data неразрешимо через сведение к проблеме останова; (2) coverage при динамически вычисляемых указателях неразрешимо через сведение к задаче Coverage; (3) размеры данных по указателю требуют семантики использования, что равносильно декомпиляции. Следовательно, $R \Rightarrow D$.

**Линия 2 (Часть II): иерархия трудностей.** Сложность PE-бинарников образует строгую иерархию $C_0 \subsetneq C_1 \subsetneq C_2 \subsetneq C_3 \subsetneq C_4$, где каждый уровень вводит новый источник неопределённости. На $C_0$ (статические прямые ссылки) $P$ вычислима за $O(n)$. На $C_1$ (jump-таблицы) требуется анализ DFG и различение code/data. На $C_2$ (динамические указатели) требуется символьное исполнение. На $C_3$ (self-modifying code) $P$ становится зависимой от времени, и статический анализ принципиально некорректен. На $C_4$ (виртуализация) целевая ISA отличается от ISA бинаря, и задача сводится к декомпиляции интерпретатора VM плюс исполнению байт-кода в восстановленной модели. В пределе $C_4 \circ C_3$ единственный подход — полная семантическая реконструкция, что в точности есть $D$.

**Линия 3 (Часть III): type inference.** Восстановление типов данных — необходимый компонент $P$, поскольку без типов невозможно корректно разбить байты на объекты, различить padding и валидные данные, установить DFG-рёбра. Type inference на бинарниках (в формализациях TIE и retypd) неразрешим в общем случае через сведение к неразрешимости type inference для типизированного $\lambda$-исчисления с подтипами (Henglein, 1994). Это *независимый* источник неразрешимости: даже если снять леммы 1 и 2, задача остаётся неразрешимой из-за одной лишь type inference.

## 5.2. Расширенная теорема

**Теорема 16** (расширенная)**.** *Реассемблирование произвольного PE-бинарника без декомпиляции невозможно. Неразрешимость проявляется независимо через четыре канала:*

1. *лемма 1 — code vs data;*
2. *лемма 2 — coverage при динамических ссылках;*
3. *лемма 3 — размеры объектов данных;*
4. *лемма 8 — типы данных.*

*Каждое из этих условий независимо делает $R$ невозможным: даже снятие любого одного не снимает других. Декомпиляция $D$, понимаемая как построение семантической модели $M \supseteq P$, является необходимым условием $R$.*

## 5.3. О природе ИИ-декомпиляции

Результат проясняет природу ИИ-подхода к реверс-инжинирингу. ИИ-декомпиляция здесь не является “улучшенным транслятором байт”. Её функция — *сжатие* бинарника $B$ в семантическую модель $M$, содержащую алгоритм, типы, потоки управления и данных, инварианты. Формально:

$$B \xrightarrow{D_{\mathrm{AI}}} M \xrightarrow{G} B'.$$

В модели $M$ побайтовое тождество теряется, но сохраняется семантика. Это согласуется с тем, что источник информации (типы, CFG, потоки, инварианты) физически отсутствует в $B$ и может быть только *выведен* из семантики исполнения. Вывод требует модели, абстрагирующей от байт к алгоритму.

ИИ-декомпиляция функционирует именно как компрессор: $B \to M$, где $M$ — неразложимая на байты, но эквивалентная по семантике модель. Это не “один из возможных подходов”, а единственный, соответствующий природе задачи.

### 5.4. Ограничения и перспективы

Настоящий результат устанавливает *общую* невозможность. Это не исключает существования разрешимых подклассов: бинарников с полностью статической адресацией ($C_0$), без self-modifying code, без виртуализации, с известными типами. Для таких подклассов частичные реассемблеры могут работать. Однако это не противоречит теореме — это лишь её частные случаи.

Практические следствия для инструментов реверс-инжиниринга (IDA Pro, Ghidra, Binary Ninja, Radare2) состоят в том, что все они, явно или неявно, реализуют ту или иную форму декомпиляции $D$. Чисто “ассемблерные” подходы без семантической реконструкции ограничены классом $C_0$ и неприменимы к нетривиальным бинарникам.

Перспективы дальнейших исследований связаны с (а) формализацией классов эквивалентности $B' \sim B$, отличных от побайтового тождества; (б) построением семантических моделей $M$, включающих инварианты и абстрактные домены; (в) информационно-теоретической формализацией ИИ-декомпиляции через Kolmogorov complexity.

### 5.5. Финальный вывод

Реассемблирование произвольного PE-бинарника без декомпиляции *невозможно* в строгом математическом смысле. Это не технологическое ограничение, преодолимое совершенствованием инструментов, а структурное следствие природы бинарного формата: информация, необходимая для реассемблирования, физически отсутствует в бинарнике и не может быть извлечена из него без построения семантической модели. Декомпиляция — единственный механизм построения такой модели, и в этом смысле она является не одним из возможных подходов, а *необходимым условием* реассемблирования.