Санкт–Петербургский государственный университет

ЛЮБАЕВ Даниил Андреевич

Выпускная квалификационная работа

Кодирование Java Bytecode в CIRCUIT-SAT для решения задач проверки эквивалентности программ

Уровень образования: бакалавриат
Направление 02.03.01 «Математика и компьютерные науки»
Основная образовательная программа СВ.5152.2019 «Математика, алгоритмы и анализ данных»

Научный руководитель:
доцент,
Факультет математики
и компьютерных наук СПбГУ,
к.ф. - м.н. Авдюшенко Александр Юрьевич

Рецензент:
научный сотрудник,
Институт динамики систем и теории управления
им. В.М. Матросова Сибирского отделения
Российской академии наук,
к.т.н. Отпущенников Илья Владимирович

Санкт-Петербург
2023 г.

Содержание

Введение

Задача проверки эквивалентности программ — это задача по распознаванию частей одной или нескольких программ, задающих одну и ту же функцию. Такими частями являются кодовые клоны — дублированные фрагменты программы, возникающие вследствие копирования и дальнейшего изменения, рефакторинга, и переиспользования исходного кода.

Обнаружение и удаление таких клонов — важная задача в разработке программного обеспечения, так как они могут привести к увеличению сложности кода, ухудшению его качества, а ошибки в одной части программы также могут присутствовать и в другой, эквивалентной ей, и исправление этой ошибки в одном фрагменте кода может потребовать коррекции эквивалентной части.

Самыми сложными для распознавания являются семантические клоны. Под семантическими клонами обычно понимаются сегменты кода, схожие функционально, но реализованные через разные синтаксические конструкции. Можно считать, что в общем случае семантические клоны — это функции, написанные на каких-либо языках программирования (возможно, различных), имеющие одинаковую сигнатуру, и на общих наборах входных данных выдающие одинаковые выходы.

Одним из способов распознавания клонов является кодирование программ в булевы схемы, построение задачи проверки эквивалентности данных схем, и дальнейшее сведение этой задачи к SAT и применение SAT-решателя. Использование схем и SAT-решателя для обнаружения клонов позволяет находить не только прямые клоны, обычно возникающие при несущественном изменении исходного кода, добавлении бесполезных переменных или переименовании существующих, но и семантические клоны, которые не всегда можно распознать при исследовании только исходного кода.

В настоящей работе описывается разработанный программный комплекс `transbyte` для кодирования части байткода виртуальный машины Java (байткод JVM) в булевы схемы, его архитектура, функциональные возможности, а также приводятся результаты использования различных кодировок для распознавания некоторых семантических клонов с помощью SAT-решателя.

Приведем краткий обзор содержания работы. В главе 1 приводятся теоретические основы техник трансляции программ в булевы формулы, используемых в символьном исполнении и описанных в [15] и [20]. Эти результаты являются базой для последующего материала. В главе 2 приводится концепция трансляции байткода JVM, дается описание архитектуры программного комплекса `transbyte`, а также рассматривается вопрос минимизации генерируемых кодировок. В главе 3 приводятся результаты использования комплекса `transbyte` и создаваемых им кодировок для распознавания некоторых функций сортировок как семантических клонов.

Глава 1. Теоретические основы трансляции программ в схемы и формулы

1.1 Задача выполнимости булевых формул (SAT)

Задача выполнимости булевых формул (SAT) состоит в определении выполнимости для произвольной булевой формулы F, т.е. можно ли присвоить всем переменным из F значения *истина* (true, 1) или *ложь* (false, 0) так, чтобы формула стала истинной. Если такие присвоения существуют, то формула является выполнимой, иначе невыполнимой. В контексте задачи SAT удобно считать, что формула находится в конъюнктивной нормальной форме (КНФ). Важным фактом является то, что используя преобразования Цейтина ([21]) любую булеву формулу можно эффективно (в общем случае за полиномиальное от ее размера время) преобразовать в равновыполнимую КНФ. Поэтому в дальнейшем задача SAT рассматривается только для КНФ.

1.2 Дискретные функции и способы их задания

Дискретными функциями называются произвольные функции вида

$$f\colon \{0,1\}^* \to \{0,1\}^*.$$

Далее рассматриваются такие дискретные функции, описаниями которых являются программы для детерминированной машины Тьюринга (ДМТ) с алфавитом $\{0,1\}$ (см., например, [5]).

Пусть f — произвольная такая функция, и M_f — вычисляющая её ДМТ-программа. Дополнительно также будем предполагать, что $\operatorname{dom} f = \{0,1\}^*$, т.е. M_f останавливается на произвольном двоичном слове, и сложность M_f растет как некоторый полином с ростом длины входа. В таком случае M_f задает счетное семейство функций вида

$$f_n\colon \{0,1\}^n \to \{0,1\}^*, \quad \operatorname{dom} f_n = \{0,1\}^n, \quad n \in \mathbb{N}.$$

Трансляция программ для детерминированной машины Тьюринга основана на следующем факте: процесс работы программы M_f на произвольном

входе можно эффективно (в общем случае за полиномиальное от n время) представить в виде формулы исчисления высказываний. Тем не менее, построение процедуры трансляции для ДМТ-программ имеет чисто теоретический интерес. Гораздо более близкими к современным вычислительным устройствам являются машины с произвольным доступом, впервые описанные в [16].

Далее используется упрощенная форма RAM, которая тем не менее достаточна для построения теории вычислимых (рекурсивных) функций. Рассматривается двоичная RAM в формализме Н. Катленда [18]. Данная модель включает потенциально бесконечную вправо ленту, разбитую на ячейки, пронумерованные натуральными числами. В каждой ячейке может быть записан только один бит. Произвольная двоичная RAM-программа — это нумерованный список команд, каждая из которых может быть командой одного из следующих двух типов:

- команды записи в ячейку с номером k бита 0 или бита 1 — соответственно $B_0(k)$ и $B_1(k)$;
- команды условного перехода $J(k, l, m)$: сравнить содержимое ячеек с номерами l и k, в случае совпадения перейти к команде с номером m, в противном случае перейти к команде, которая следует в списке за командой $J(k, l, m)$.

Вычисление останавливается либо после выполнения последней команды в программе (если это не команда условного перехода), либо если происходит ссылка на несуществующую команду.

Синтаксис такой формы RAM-программ довольно близок к ассемблерным программам, что является крайне важным при построении практичных процедур кодирования.

В соответствии с [5], значение функции сложности $\rho(n)$ программы M_f равно максимуму числа шагов ДМТ, выполняющей M_f, по всевозможным входам из $\{0,1\}^n$. Пусть R_{f_n} — произвольная программа двоичной RAM, вычисляющая функцию f_n. Поставим ей в соответствие значение функции $\nu(n)$, равное максимальному по всевозможным входам из $\{0,1\}^n$ числу обращений к регистрам ленты RAM в процессе выполнения R_{f_n}.

Следующие два утверждения из работы [20] являются теоретической базой для описываемых далее процедур кодирования

Лемма 1.1 (о моделировании)**.** *Пусть M_f — ДМТ-программа, вычисляющая всюду определенную (тотальную) дискретную функцию $f\colon \{0,1\}^* \to \{0,1\}^*$, и функция сложности M_f ограничена некоторым полиномом от n. Существует тотальная алгоритмически вычислимая функция g, которая за полиномиальное от n время по тексту программы M_f и числу n выдает текст программы R_{f_n}, вычисляющей функцию f_n. Функция $\nu(n)$, сопоставляемая получаемому семейству RAM-программ, ограничена сверху полиномом от n.*

Данный факт означает наличие эффективной процедуры перехода от ДМТ-программы, вычисляющей f, к семейству двоичных RAM-программ, каждая из которых вычисляет $f_n, n \in \mathbb{N}$.

Теорема 1.2. *Пусть $f = \{f_n\}_{n\in\mathbb{N}}$ — семейство алгоритмически вычислимых за полиномиальное время дискретных функций и $\{R_{f_n}\}_{n\in\mathbb{N}}$ — семейство двоичных RAM-программ, сопоставляемое f в соответствии с леммой о моделировании. Существует алгоритмически вычислимая тотальная функция h, которая, получая на входе текст программы R_{f_n}, за полиномиальное в общем случае от n время строит такую систему булевых уравнений $S(f_n)$, что для произвольного $y \in \operatorname{range} f_n$ система $S(f_n)|_y$ совместна. Если x^* — произвольное решение системы $S(f_n)|_y$, то из x^* за линейное от $|x^*|$ время можно выделить некоторый $x \in \{0,1\}^n$, такой, что $f_n(x) = y$.*

Через $S(f_n)|_y$ обозначена система булевых уравнений, которая получается из системы $S(f_n)$ в результате подстановки в нее вектора $y \in \operatorname{range} f_n$.

При доказательстве данной теоремы в явном виде строится процедура, которая рассматривает процесс RAM-вычисления как последовательность переходов между конфигурациями RAM-машины $K_0 \to K_1 \to \ldots \to K_e$, где K_0 — начальная конфигурация, K_e — конечная, все остальные конфигурации являются промежуточными. Каждой такой конфигурации K_i, $i \in \{0,1,\ldots,e\}$ сопоставляется множество булевых переменных X^i, а каждому переходу — система булевых уравнений, связывающих соответствующие

соседние множества. Конъюнкция всех таких систем дает систему $S(f_n)$, кодирующую процесс исполнения программы R_{f_n} на произвольном входе. От произвольной системы вида $S(f_n)|_y$ возможен эффективный переход к одному уравнению вида КНФ = 1, который осуществляется при помощи преобразований Цейтина ([21]). Между множеством решений системы $S(f_n)|_y$ и множеством решений получаемого уравнения КНФ = 1 существует биекция ([19]).

Следует отметить, что описанная выше техника преобразования программ в булевы уравнения и формулы на самом деле является развитием фундаментальной идеи С.А. Кука о пропозициональном кодировании программ для машины Тьюринга, высказанной им в основополагающей статье по структурной теории сложности [4]. Позже фактически эта же идея была использована Дж. С. Кингом для формулировки концепции символьного исполнения (Symbolic Execution) программ [13].

Каждый язык программирования имеет семантику исполнения, описывающую правила, по которым операторы языка манипулируют входными данными в соответствии с инструкциями программы. Также мы можем определить альтернативную семантику символьного исполнения для языка программирования, в котором реальные входные данные не используются, но могут описываться некоторыми символами. Символьное исполнение — это естественное расширение обычного исполнения в том смысле, что обычное исполнение является конкретизацией символьного. При символьном исполнении семантика основных операторов языка расширена таким образом, чтобы принимать на входе символы и выдавать на выходе формулы. При этом выполнение инструкций также происходит в символьном виде, без конкретных значений данных, над которыми эти инструкции оперируют.

Идея трансляции программ очень близка к идее символьного исполнения и по сути на ней и основана. Символьными выражениями при трансляции являются наборы булевых переменных, задающих переменные в изначальной программе. Их количество зависит от размера переменной, которую они задают. Выполнение конкретной инструкции над переменными программы задается булевой формулой над соответствующими булевыми переменными. Исполнение всей программы по итогу задается конъюнкцией всех таких фор-

мул.

1.3 Схемы из функциональных элементов как способ задания дискретных функций

Наличие процедуры эффективного перехода от произвольной RAM-программы к булевому уравнению вида КНФ = 1, задающему данную программу, позволяет использовать различные инструменты для работы с булевыми уравнениями с целью анализа функционала RAM-программы. Тем не менее, намного более удобным способом задания RAM-программы является булева схема.

Булевой схемой называется ориентированный ациклический граф $S = (V, E)$, где V — это вершины этого графа, а $E \subseteq V^2$ — ребра. Ребром называется упорядоченная пара вершин. Для каждого ребра $(u, v) \in E$, вершина u называется родительской, а вершина v — дочерней. Вершина называется входом схемы, если у нее нету родительских вершин, и выходом, если у нее нету дочерних вершин. Любая вершина, не являющаяся входом или выходом, называется вентилем. Каждый вентиль связан с некоторой логической связкой из предопределенного множества, называемого базисом (например, $\{\wedge, \neg\}$).

Рассмотрим функцию вида

$$f\colon \{0,1\}^+ \to \{0,1\}^+,$$

где через $\{0,1\}^+$ обозначено множество всех бинарных слов длины $n = 1, 2, \ldots$. Также дополнительно предполагается, что каждая такая функция тотальна на $\{0,1\}^+$, и задается ДМТ-программой M_f. Программа M_f задает счетное семейство функций вида

$$f_n\colon \{0,1\}^n \to \{0,1\}^+, \quad \operatorname{dom} f_n = \{0,1\}^n, \quad n = 1, 2, \ldots \tag{1}$$

В соответствии с леммой о моделировании, каждую функцию $f_n, n \in \mathbb{N}$ можно так же задать RAM-программой.

Зафиксируем n для произвольной функции вида (1). Поскольку время

выполнения программы, задающей соответствующую функцию, ограничено для любого $x \in \{0,1\}^n$, можно рассмотреть эту функцию в форме

$$f_n \colon \{0,1\}^n \to \{0,1\}^m$$

Каждая такая функция может быть естественным образом задана булевой схемой $S(f_n)$ с n входами и m выходами над произвольным полным базисом. В дальнейшем предполагается, что везде используется базис $\{\wedge, \neg\}$.

Предположим, что нам дана схема $S(f_n)$ с n входами и m выходами. Зафиксируем некоторый порядок на множестве входов и выходов. Каждому входу схемы $S(f_n)$ сопоставим булеву переменную, и полученное множество обозначим за $X = \{x_1, \ldots, x_n\}$. То же самое сделаем с выходами и полученное множество обозначим за $Y = \{y_1, \ldots, y_m\}$.

Для схемы $S(f_n)$ за линейное от количества вершин в схеме можно построить КНФ, обозначаемую через $C(f_n)$. Соответствующий алгоритм обходит каждую вершину в схеме ровно один раз. С каждым вентилем $G \in \{\neg, \wedge\}$ сопоставляется вспомогательная переменная $u(G)$ из множества $U \colon U \cap X = \emptyset$. Для произвольной $u(G)$ строится КНФ $C(G)$, которая использует не более 3 булевых переменных. Точное представление $C(G)$ зависит от вентиля G. Результатом данного процесса является КНФ:

$$C(f_n) = \bigwedge_{G \in S(f_n)} C(G).$$

Описанная техника построения КНФ по схеме $S(f_n)$ — это те же преобразования Цейтина, упомянутые в предыдущем разделе ([21]).

1.4 Проверка эквивалентности булевых схем

Для того, чтобы проверить, что две схемы, имеющие одинаковое количество входов и выходов, эквивалентны, т.е. на одинаковых входах генерируют идентичные выходы, нужно построить новую схему, называемую майтером (miter).

Пусть входом первой схемы были булевы переменные $x_1, \ldots, x_n$, а вы-

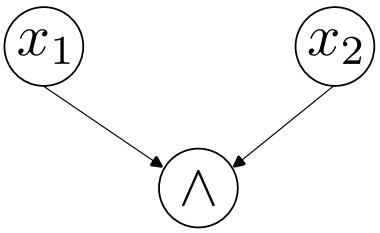

Рис. 1: Схема для конъюнкции.

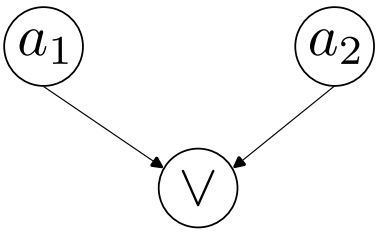

Рис. 2: Схема для дизъюнкции.

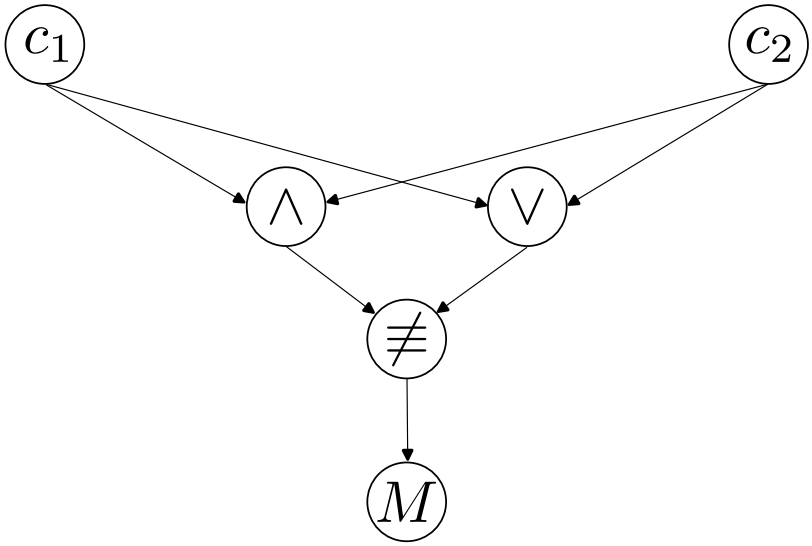

Рис. 3: Майтер для схем конъюнкции 1 и дизъюнкции 2.

ходом $y_1, \ldots, y_k$. Входом второй схемы были переменные $a_1, \ldots, a_n$, а выходом $b_1, \ldots, b_k$. Обозначим за $X_i = \{x_1^i, \ldots, x_{l_i}^i\}$ для $i = 1, \ldots, n$ множество всех вершин, в которые существует ребро из x_i. За $A_i = \{a_1^i, \ldots, a_{m_i}^i\}$ для $i = 1, \ldots, n$ обозначим множество вершин, в которые существует ребро из a_i. Майтер представляет из себя объединение этих двух схем с новыми входами $c_1, \ldots, c_n$, новым выходом M, и некоторыми дополнительными вентилями и ребрами. Сначала в схему добавляются следующие ребра

- $(c_i, x_j^i),\ i = 1, \ldots, n,\ j = 1, \ldots, l_i$;
- $(c_i, a_j^i),\ i = 1, \ldots, n,\ j = 1, \ldots, m_i$.

Эти ребра задают то, что входы двух предыдущих схем должны совпадать, т.е. что у двух предыдущих схем теперь одинаковый вход. Помимо этого, добавляются новые вентили $v_1, \ldots, v_k$, при этом каждый такой вентиль представляет собой логическую связку $\not\equiv$. После чего в схему добавляются новые ребра

- $(y_i, v_i),\ i = 1, \ldots, k$;
- $(b_i, v_i),\ i = 1, \ldots, k$.

Эти ребра уже задают условие, что выходы двух схем не должны совпадать. После этого, все вентили $v_1, \ldots, v_k$ связываются с выходом новой схемы M, который представляет собой дизъюнкцию по k переменным.

В итоге, если выход M равен 0, то это значит, что выходы двух предыдущих схем совпадают. Если выход M равен 1, то это значит, что в каком-то месте в выходах двух предыдущих схем есть отличие.

Данную схему можно эффективно преобразовать в булеву формулу вида КНФ = 1. Если данная КНФ выполнима, то это значит, что существует вход, на котором выходом майтера является 1, следовательно, существует такой вход, на котором две предыдущие схемы отличаются. Если же КНФ невыполнима, то такого входа не существует.

Для проверки эквивалентности булевых схем также нужно каким-то образом хранить их представление в программном комплексе. Наиболее удобным представлением булевых схем над базисом $\{\neg, \wedge\}$ (также называемых И-НЕ графами, или And-Inverter графами) является формат AIGER [2].

И-НЕ граф в формате AIGER представляет из себя файл с расширением `.aag`, являющийся текстовым файлом с представлением схемы, или файл с расширением `.aig`, являющийся сжатым бинарным вариантом файла с расширением `.aag`. Программы, позволяющие оперировать И-НЕ графами, обычно работают с файлами `.aig` в силу того, что такие файлы занимают значительно меньше места на диске и в оперативной памяти. Тем не менее, текстовое представление проще для восприятия, поэтому дальше описывается именно оно.

Первой строчкой в файле с расширением `.aag` является заголовок. В нём содержится некоторая информация, описывающая схему. Нам важно то, что заголовок содержит в себе количество входных битов, количество выходных битов, и количество вентилей. После заголовка идут номера входных битов, номера выходных битов, и записи вида `x y z`, где $x = 2n,\ n \in \mathbb{N},\ y, z \in \mathbb{N} \cup \{0\}$. Каждая такая запись означает следующее

- x представляет булеву переменную с номером n;

- если $y = 0$ или $y = 1$, то y представляет соответствующую булеву константу; если же $y = 2m, m \in \mathbb{N}, m \neq n$, то y представляет булеву переменную с номером m; если же $y = 2m+1$, то y представляет булеву переменную, являющуюся логическим отрицанием булевой переменной с номером m;

- аналогично для z.

В итоге данная запись представляет собой булеву формулу $x \equiv y \wedge z$, где x представляет некоторую булеву переменную, а y и z представляют либо другие булевы переменные, либо константы 0 или 1. Номера входных и выходных битов означают то же самое, за исключением, что номера входных битов всегда должны быть четными натуральными числами, и номерами выходных битов не могут быть 0 и 1.

Схему в AIGER-формате можно перевести в КНФ в наиболее известном формате DIMACS [11]. Это умеет делать, например, ABC [17] — система для синтеза и верификации схем. ABC также умеет создавать майтеры из двух других схем в AIGER-формате. Полученную КНФ для майтера можно использовать в SAT-решателе, чтобы понять, является она выполнимой, или нет.

1.5 Проблема определения кодовых клонов

Копирование различных частей исходного кода комплексных программ и их переиспользование без изменений, либо же с минимальными дополнениями, очень часто встречается в разработке программного обеспечения. Такие дублированные фрагменты называются кодовыми клонами. Клонированием частей программы может привести к тому, что различные ошибки, находящиеся в одной части программы, будут также присутствовать и в скопированных сегментах. Поэтому достаточно важным является обнаружение таких связанных фрагментов программы.

На данный момент не существует единого определения понятия кодового клона. Различные исследователи вкладывают в это понятие разный смысл. Тем не менее, выделяется несколько основных типов клонов ([1]).

Точные клоны (Exact Clones) (1 тип) — это идентичные сегменты кода, за исключением изменений в комментариях, компоновке и оступах.

Переименованные клоны (Renamed Clones) (2 тип) — это сегменты кода, которые синтаксически или структурно похожи, за исключением изменений в комментариях, именовании переменных, типах, литералах, компоновке.

Неточные клоны (Near Miss Clones) (3 тип) — это скопированные сегменты кода с последующей модификацией в виде добавления или удаления различных условий перехода, изменении названий переменных, компоновки, но с одинаковым результатом исполнения.

Семантические клоны (Semantic Clones) (4 тип) — это сегменты коды, схожие функционально, но сильно различающиеся по синтаксической реализации.

Существует достаточно большое количество инструментов и подходов для определения первых трёх типов клонов [1]. Используются подходы, основанные на текстовом, лексическом анализе, подходы, основанные на метриках, показывающих, насколько различным является исходный код. Также используется машинное обучение и NLP. Различные методы также используются одновременно, порождая некоторые гибридные подходы.

Инструментов, умеющих определять четвертый тип клонов, намного меньше. Хорошим решением для обнаружения семантических клонов является абстрагирование от исходного кода и работа с функционалом, задающим часть программы, например анализ графа исполнения программы.

Один из интересных подходов к распознаванию семантичеких клонов, являющихся мотивацией данной работы, это трансляция программ в булевы схемы и дальнейшее сведение этой задачи к задаче SAT, а именно использование SAT-решателя. SAT-решатели работают с булевыми формулами, по сути кодирующими процесс вычисления функции, и абсолютно ничего не знают про исходный код. Как отмечалось в предыдущих разделах, программы можно закодировать в булевы схемы, после чего булевы схемы объединить в майтер. Майтер же, в свою очередь, можно преобразовать в формулу вида КНФ = 1, выполнимость которой можно попытаться установить с помощью SAT-решателя. Если две части программы являются семантическими клонами, то их майтер, закодированный в КНФ, должен быть невыполнимым.

1.6 Проблемы символьного исполнения и трансляции в формулы для высокоуровневых языков программирования

Хотя символьное исполнение программ может быть полезным инструментом в кодировании, анализе и оптимизации программы, оно также сталкивается с некоторыми проблемами, которые могут затруднить его использование.

Одна из главных проблем символьного исполнения — это проблема взрывного роста состояний. Как описано ранее, при символьном исполнении программа рассматривается на основе её символьной модели, что может привести к генерации большого количества символьных выражений, представляющих возможные состояния программы на каждом шаге её исполнения. Это может привести к значительному увеличению времени и объема памяти, необходимых для проведения символьного исполнения.

Кроме того, символьное исполнение может столкнуться с проблемой потери точности в случае наличия сложных зависимостей между переменными программы. Например, символьное исполнение может не учитывать некоторые пути выполнения программы, которые могут быть осуществлены только при определенных значениях входных данных.

Символьное исполнение также может быть ограничено доступными ресурсами, такими как вычислительная мощность и объем памяти. В случае больших и сложных программ, символьное исполнение может потребовать существенных вычислительных и временных затрат, что может стать проблемой при проведении анализа.

Абсолютно идентичные проблемы возникают и при трансляции программ в булевы схемы. Более того, при проверке эквивалентности двух схем также появляется проблема, состоящая в том, что SAT-решатели для установления выполнимости КНФ в общем случае могут задействовать полный перебор, что является практически неэффективным и занимает очень много времени.

Помимо вышеописанных проблем, при анализе высокоуровневых языков программирования, таких как Python, Java, C++, и т.д., возникают и другие ограничения. Одной из главных проблем для высокоуровневых языков

программирования является высокая стоимость для программ с большим объемом кода. Высокоуровневые языки программирования имеют более высокий уровень абстракции, что означает, что для анализа программы может потребоваться более сложная и вычислительно затратная модель. Многие высокоуровневые языки программирования имеют богатую стандартную библиотеку, которая может содержать множество функций и классов, сложных для кодирования. Более того, даже для языков более низкого уровня, например C, разные компиляторы могут генерировать различный ассемблерный код, зависящий от платформы.

Ещё одной проблемой является сложность анализа динамически создаваемых объектов и изменяемых структур данных. Высокоуровневые языки программирования часто используют динамическую память и имеют возможность создавать объекты во время выполнения программы. Это усложняет как символьное исполнение, так и трансляцию, поскольку придётся учитывать все возможные варианты создания объектов и изменения структур данных. Также следует отметить, что с языками, использующими динамические системы типов, которые могут меняться в зависимости от контекста выполнения программы, сложнее работать.

В итоге можно сделать вывод, что работа с RAM-моделью не имеет практического смысла, так как эта модель слишком далека от используемых в реальных вычислениях моделей и инструментов. Тем не менее, кодирование высокоуровневных языков напрямую добавляет большое количество проблем, из-за чего модель транслятора становится объемной, а построение транслятора на практике становится слишком сложным. Именно поэтому в качестве усредненной модели, более низкоуровненой, чем языки программирования, но намного более выразительной, чем RAM-машина, в данной работе рассматривается виртуальная машина.

Глава 2. Трансляция программ для виртуальной машины в And-Inverter графы

2.1 Концепция трансляции программ в And-Inverter графы для виртуальных машин

Наличие виртуальной машины с платформонезависимым байткодом, в который преобразуются инструкции языка программирования, позволяет транслировать программы на данном языке программирования в And-Inverter графы, основываясь на инструкциях байткода, которые получаются после компиляции. Виртуальную машину можно рассматривать как некоторую модель вычисления, достаточно близкую к языкам, компилируемым в ассемблерный код, но без привязке к платформе и ее специфичным расширениям.

Инструкции виртуальной машины обычно намного более низкоуровневые, чем инструкции языка программирования, работающего поверх этой виртуальной машины, поэтому с ними проще работать. Более того, если кодировать именно инструкции виртуальной машины, появляется возможность трансляции всех языков программирования, компилятор или интерпретатор которых генерирует байткод. Например, если кодировать байткод виртуальной машины Java (JVM), то появляется возможность трансляции языков программирования Java, Kotlin, Scala. Для этого нужно будет сначала скомпилировать программу на выбранном языке программирования, после чего промежуточное байткод-представление транслировать в булеву схему. Так можно, например, проверять на эквивалентность схемы даже для программ, написанных на различных языках программирования.

2.2 Система трансляции байткода виртуальной машины Java

Виртуальная машина Java является, пожалуй, одной из самых популярных виртуальных машин на данный момент. Она также обладает описанными в предыдущем разделе свойствами: байткод JVM является кроссплатформенным, инструкции виртуальной машины намного более низкоуровненые, чем инструкции языка программирования Java, а сама JVM является практической моделью вычислений. Программы для JVM представляют собой классы,

содержащие в себе статические и нестатические поля, статические и нестатические методы, таблицы констант и т.д. JVM является стековой машиной. При исполнении каждой функции создается стек, а также набор пронумерованных локальных переменных. Переменные-члены класса хранятся в куче.

Наличие большого количества информации о виртуальной машине, а также различных инструментов для манипуляции байткодом, позволяет рассматривать виртуальную машину Java как модель, программы которой можно транслировать в булевы схемы. Тем не менее, JVM все еще содержит в себе инструкции, позволяющие манипулировать достаточно высокоуровневыми абстракциями: например, инструкции, позволяющие проверять по классу, является ли он реализацией какого-то конкретного интерфейса, инструкции, дающие возможность вызывать методы у интерфейсов (когда класс-реализация неизвестен), а также инструкции, вызывающие код C/C++, реализованный через Java Native Interface (JNI), и т.д. Именно поэтому описанный далее программный комплекс для трансляции байткода JVM в булевы схемы умеет работать не со всеми инструкциями виртуальной машины, а лишь с некоторым подмножеством, в основном оперирующим примитивными типами, наподобие `int`, `byte`, `boolean`, и т.д.

Помимо вышеописанных проблем, при трансляции возникают и другие проблемы, присущие любому символьному исполнению. Решения некоторых из них будут также рассмотрены в дальнейшем.

2.3 Программный комплекс `transbyte`

Для трансляции подмножества инструкций виртуальной машины Java был разработан программный комплекс `transbyte` [9] (от слов ***trans****lation of* ***byte****code*).

Программный комплекс был написан на языке Kotlin, так как большинство инструментов, позволяющих анализировать и манипулировать байткодом класс-файла JVM, благодаря чему не нужно создавать свой собственный парсер класс-файлов, написаны на Java. Таковыми являются, например, ASM [7], и использованная в `transbyte` библиотека Apache Bytecode Engineering Library [8] (Apache BCEL). Эти библиотеки, как и любые библиотеки для

языка программирования Java, можно использовать и в Kotlin. В качестве библиотеки для CLI (Command Line Interface) была использована библиотека Clikt [6], для логирования использовались SLF4J [12] и kotlin-logging [10].

Данный программный комплекс позволяет получать кодировки в форматах AIGER [2] и DIMACS [11]. Нужный формат можно указать через параметры командной строки. Для этого ему нужно передать пути до всех нужных `.class`-файлов. Программный комплекс также умеет принимать на вход `.java`-файлы и компилировать их, если в системе есть Java-компилятор. Он также обладает несколькими дополнительными параметрами. Некоторые из них являются обязательными, например параметр $--$ `start` $-$ `class`, указывающий название класса, содержащего метод, с которого должна начинаться трансляция, и параметр $--$ `array` $-$ `sizes`, являющийся обязательным в случае, если метод, с которого начинается трансляция, во входных аргументах содержит массив. Параметр $--$ `method`, задающий имя стартового метода, является обязательным в случае, если класс, содержащий данный метод, также содержит другие методы. В ином случае, этот метод и будет использован как стартовый. Дополнительные параметры $--$ `output`, $--$ `debug`, $--$ `format` являются необязательными и предоставляют дополнительные возможности, например сохранение выходной схемы в файл, вывод отладочной информации, формат выходного файла соответственно.

2.4 Описание работы программного комплекса

Программный комплекс умеет работать с примитивными типами `int`, `byte`, `double`, `short`, `boolean`, `long`, `float`, `char`, массивами и статическими методами.

Каждая переменная примитивного типа представляется в виде набора битов (тип `Bit`), каждый из которых имеет свой номер. Количество этих битов зависит от размера примитивного типа (для `int` это 32, для `byte` это 8). Для более компактного хранения помимо номера бита также хранится информация о том, является ли этот бит отрицанием какого-либо другого.

В дальнейшем переменные внутри программы JVM, понимающиеся в традиционном смысле как идентификаторы областей памяти, будут назы-

ваться переменными программы, а наборы битов, задающих эту переменную программы в трансляторе, которые понимаются как пропозициональные переменные в итоговой булевой системе, будут называться переменными трансляции.

При вызове каждого метода создается стек и массив локальных переменных программы. Они нужны для симуляции вычисления виртуальной машины. Помимо этого создается булева система, в которой содержатся кодирующие программу булевы уравнения.

Трансляция программы происходит путем симуляции вычисления виртуальной машины. Каждая инструкция, оперирующая над переменными программы, по сути аналогична некоторому набору действий с регистрами RAM-машины. В программном комплексе каждая такая инструкция кодирует соответствующую операцию в булевых формулах над базисом $\{\wedge, \neg\}$.

Пример 2.1. Для функции суммы двух целочисленных переменных, написанной на языке программирования Java,

```
public static int Sum(int a, int b) {
    return a + b;
}
```

компилятор генерирует следующий байткод:

```
iload_0
iload_1
iadd
ireturn
```

где инструкция `iload` берет целочисленное значение из локальной переменной с соответствующим индексом и кладет на стек; инструкция `iadd` берет два целых числа со стека, складывает их, а результат кладет обратно на стек; инструкция `ireturn` возвращает целочисленное значение, лежащее на стеке.

Рассмотрим процесс трансляции этих инструкций байткода. При вызове функции, как и упоминалось выше, сначала создается стек и множество локальных переменных для переменных трансляции.

При этом в соответствующие индексы кладутся переменные трансляции, сопоставленные переменным программы `int a` и `int b`. После чего есте-

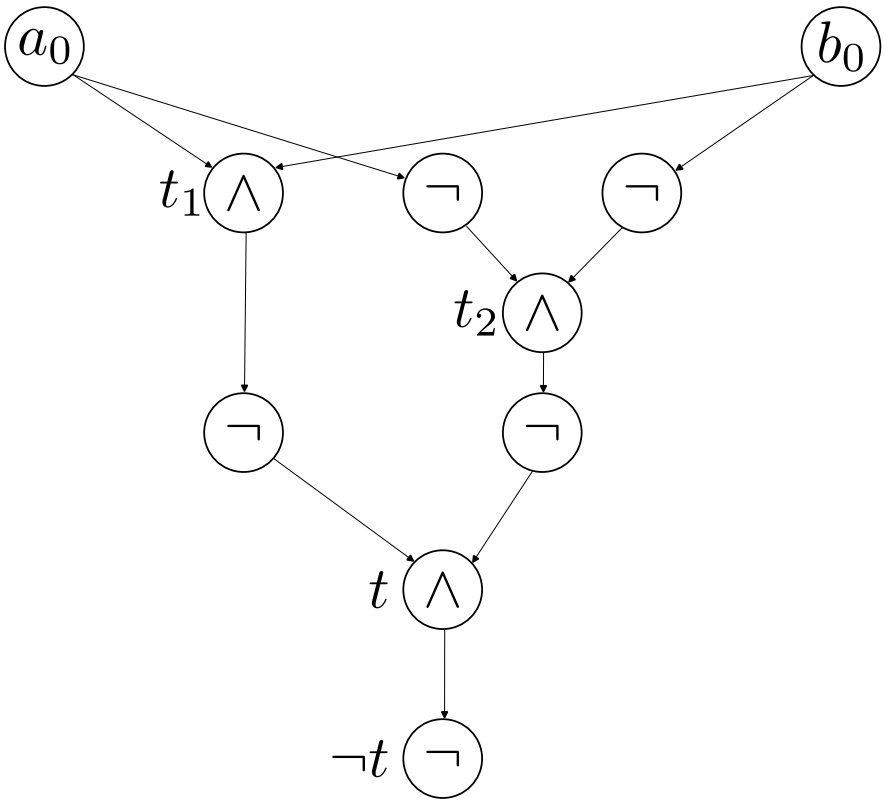

Рис. 4: Схема для «исключающего или» в базисе $\{\neg, \wedge\}$

ственным образом симулируется пара инструкций `iload`: переменные трансляции берутся из множества локальных переменных и кладутся на стек.

Следующим шагом является симуляции инструкции целочисленного сложения `iadd`. Для того, чтобы это сделать, используется алгоритм сложения «в столбик»:

$$\begin{cases} (c_0 \equiv a_0 \oplus b_0) = 1, \\ (p_0 \equiv a_0 \wedge b_0) = 1, \\ (p_j \equiv \mathrm{maj}(a_j, b_j, p_{j-1})) = 1, \quad j = 1, \ldots, n-1, \\ (c_i \equiv a_i \oplus b_i \oplus p_{i-1}) = 1, \quad i = 1, \ldots, n-1. \end{cases}$$

Запись $\mathrm{maj}(x, y, z)$ означает терм $x \wedge y \vee x \wedge z \vee y \wedge z$. Сначала транслятор кодирует все термы, стоящие справа в эквивалентностях (например, $a_i \oplus b_i \oplus p_{i-1}$), в термы над базисом $\{\neg, \wedge\}$, добавляя нужное количество новых переменных трансляции. Например, терм $a_0 \oplus b_0$, схема для которого изображена на рис.

4, сначала преобразуется в

$$\neg\Big(\neg(a_0 \wedge b_0) \wedge \neg(\neg a_0 \wedge \neg b_0)\Big)$$

после чего в систему добавляются новые формулы

$$\begin{cases} (t_1 \equiv a_0 \wedge b_0) = 1, \\ (t_2 \equiv \neg a_0 \wedge \neg b_0) = 1, \\ (t \equiv \neg t_1 \wedge \neg t_2) = 1. \end{cases}$$

Терм $(c_0 \equiv a_0 \oplus b_0) = 1$ в итоге представляется как

$$(c_0 \equiv \neg t) = 1.$$

Это уравнение и добавляется в систему. Аналогичным образом происходит преобразование остальных уравнений.

После того, как все нужные уравнения из алгоритма сложения «в столбик» будут добавлены в систему, результат сложения будет представлен переменной трансляции, задающейся переменными $c_0, \ldots, c_{n-1}$. Новая переменная трансляции кладется на стек, после чего возвращается как результат работы функции через симуляцию инструкции `ireturn`.

Стоит отметить, что в полном алгоритме сложения «в столбик» также должна быть запись $(c_n \equiv p_{n-1}) = 1$. В программном комплексе она убрана для того, чтобы симулировать целочисленное переполнение переменных программы. □

Подобным образом происходит кодирование инструкций над переменными программы, задающих разницу, умножение, логическое И, ИЛИ, исключающее ИЛИ (XOR), и т.д. Различные инструкции, симулирующие добавление/удаление переменных на стек или в массив локальных переменных программы, реализованы естественным образом, только с переменными трансляции.

Помимо примитивных переменных, транслятор также умеет работать с одномерными массивами фиксированного размера. При этом обращение по индексу должно происходить либо с переменной, для которой известно

значение при выполнении программы, либо для этой переменной есть информация о ее версиях, например, чему она могла быть равна при выполнении различных ветвей условных переходов:

```
if (P) {
    x = 1;
} else {
    x = 2;
}
```

Если о переменной, по которой берется индекс, ничего не известно, то трансляция невозможна в силу того, что для этого нужно будет рассматривать все возможные состояния этой переменной (для `int` это будут, например, числа от 0 до $2^{31} - 1$, отрицательные числа не учитываются при взятии индекса), либо рассматривать все переменные в массиве, и то и другое в общем случае приведет к тому, что процесс трансляции будет очень долгим, а полученная кодировка не будет представлять практического интереса для проверки ее эквивалентности с другими кодировками из-за слишком большого размера. Более подробно этот вопрос рассматривается в последующих разделах.

Стоит также отметить, что каждая инструкция, изменяющая какую-либо из локальных переменных программы, при трансляции должна порождать новую переменную трансляции, из-за чего нужно обеспечивать, чтобы номера битов в переменных трансляции не совпадали. Для этого в программном комплексе существует планировщик битов (тип `BitScheduler`), имеющий метод `getAndShift(size : Int) : BitsArray`, который принимает количество битов, которые нужно создать, и возвращает массив этих битов, при этом увеличивая внутреннее количество уже созданных битов. Переменная программы по итогу будет представляться уже новой переменной трансляции.

2.5 Кодирование условных переходов

Одной из самых важных и сложных задач при симуляции вычисления виртуальной машины является интерпретация условного оператора. В `transbyte` эта интерпретация начинается с анализа условного выражения.

Условный оператор можно представлять в следующем виде:

`if` $P(X_1, \ldots, X_l)$ `then Branch 1;`
`else Branch 2;`

где терм условного выражения $P(X_1, \ldots, X_l)$ — терм над переменными программы. Транслятор сначала интепретирует терм условного выражения P, представляя его термом над переменными трансляции (т.е. по сути некоторой булевой формулой) $p(x_1, \ldots, x_l)$, после чего интерпретирует инструкции из ветвей условного оператора. При этом одна и та же переменная может появляться в обеих ветвях оператора.

Рассмотрим теперь условный оператор с двумя ветвями условного перехода. Пусть в обеих ветвях выполняется операция, которая присваивает некоторой переменной программы Z новое значение (для каждой ветви разное). Пусть эти значения в переменных программы были Δ_1 и Δ_2 для первой и второй ветви соответственно, а δ_1 и δ_2 — соответствующие им переменные трансляции. В таком случае транслятор связывает переменную Z с новой переменной трансляции z и добавляет в систему уравнений следующее уравнение:

$$z \equiv \Big(p(x_1, \ldots, x_l) \wedge \delta_1 \vee \neg p(x_1, \ldots, x_l) \wedge \delta_2\Big) = 1.$$

При этом, если для переменной Δ_1 или Δ_2 при исполнении было известно значение (т.е. это была некоторая константа), то во внутреннее представление переменной z также добавляется информация о том, что при выполнении соответствующего условия (p или $\neg p$) эта переменная имела какое-то конкретное значение. Если, например, переменная Δ_1 была равна c_1, то информация добавляется в виде пары (p, c_1). Если переменная Δ_2 была равна c_2, то также добавляется пара $(\neg p, c_2)$. Это нужно для некоторых оптимизаций и работы с другими условными операторами или циклами.

Вложенные условные операторы интерпретируются похожим образом, но с учетом глубины вложенности. Например, по следующему условному

оператору:

```
if P1(X1, ..., Xl) then
  if P2(Y1, ..., Yk) then Z = A;
  else Z = B;
else Z = C;
```

с соответствующими переменным программы Z, A, B, C переменным трансляции z, a, b, c и соответствующим термам над переменными программы $P_1(X_1, \ldots, X_l)$, $P_2(Y_1, \ldots, Y_k)$ термам над переменными трансляции $p_1(x_1, \ldots, x_l)$, $p_2(y_1, \ldots, y_k)$, в систему будет добавлена следующая формула:

$$z \equiv \Big(p_1(\ldots) \wedge p_2(\ldots) \wedge a \vee p_1(\ldots) \wedge \neg p_2(\ldots) \wedge b \vee \neg p_1(\ldots) \wedge c\Big) = 1.$$

То есть, при выполнении условий p_1 и p_2, переменная z равна a. Если p_1 выполняется, а p_2 нет (т.е. выполняется $\neg p_2$), то z равна b. Если же p_1 не выполняется, то z равна c. Таким же образом происходит интерпретация условных операторов с большей вложенностью, как в `if`-ветви, так и в `else`-ветви.

2.6 Условные переходы и циклы в байткоде JVM

В виртуальной машине Java, как и в ассемблерных языках, интепретация условных переходов происходит через инструкции условного перехода (например, `IF_ICMPEQ idx`, которая берет два числа со стека, сравнивает их, и в случае равенства начинает исполнение с инструкции на позиции `idx`) и через инструкцию безусловного перехода `GOTO idx` (или `GOTO_W idx`).

В такой ситуации возникает следующая проблема: если при написании программы на высокоуровневом языке программирования мы точно знаем границы ветвей в условных переходах, то на уровне байткода JVM эта граница становится размыта. Более того, оператор безусловного перехода может находиться на любой позиции, создавая нелинейную, запутанную управляющую логику (т.н. «спагетти-код»).

Тем не менее, при компиляции условных переходов на языке Java в

байткод JVM все же прослеживается определенный шаблон в положении инструкций условного и безусловного перехода. Например, для условного оператора

```
if (a < b) {
    // if-logic
} else {
    // else-logic
}
// after-logic
```

инструкции байткода будут примерно следующими:

```
1 // Put a and b on stack from local variables
2 IF_ICMPGE 5    // compare a and b
3 // if-logic
4 GOTO 6
5 // else-logic
6 // after-logic
```

Индексы в инструкциях условно обозначены номером строки, на которую надо переходить. В реальном байткоде эти индексы обозначают не номер строки, а номер байта, на котором находится нужная инструкция (потому что разные инструкции занимают разное количество байт). Здесь для простоты и наглядности взяты именно номера строк. В данном листинге виден следующий шаблон: индекс в инструкции условного перехода (в данном случае инструкции `IF_ICMPGE`) всегда указывает на номер инструкции, с которой начинается ветвь `else`. После самой инструкции условного перехода идут инструкции из ветви `if`, вплоть до инструкции безусловного перехода `GOTO`. Сама же `GOTO` перемещает на инструкцию после `else`-ветви, то есть по индексу в инструкции безусловного перехода можно понять, где кончаются инструкции из этой ветви.

Если же `else`-ветвь отсутствует, т.е. условный переход в программе выглядит следующим образом:

```
if (a < b) {
    // logic
```

```
}
// after-logic
```

то инструкции байткода будут примерно следующими:

```
// Put a and b on stack from local variables
IF_ICMPGE 4    // compare a and b
// logic
// after-logic
```

То есть индекс в инструкции условного перехода сразу указывает на инструкции после `if`-ветви, а оператор `GOTO` отсутствует.

С шаблоном для цикла все еще проще. Весь цикл по сути можно представлять как последовательность условных переходов для каждой итерации, а сами по себе циклы как раз задаются операторами условного и безусловного перехода. Например, для кода следующего цикла:

```
for (int i = 0; i < 10; i++) {
    // logic
}
// after-logic
```

генерируются примерно такие JVM-инструкции:

```
// load i on stack, push 10 on stack
IF_ICMPGE 6    // compare i and 10
// logic
// increment i
GOTO 1
// after-logic
```

То есть индекс в инструкции условного перехода все так же указывает на инструкцию после цикла (или же после условного перехода, если представлять цикл последовательностью условных переходов, как сказано выше), но при этом индекс в инструкции `GOTO` перемещает исполнение назад, а не вперед. Именно этот факт позволяет транслятору различать обычные условные переходы и циклы.

Также всегда подразумевается, что цикл конечен, то есть итерации происходят фиксированное количество раз. Если игнорировать данное условие, то в общем случае нельзя будет сказать, когда цикл должен остановиться, и для этого придется рассматривать все возможные состояния переменной, по которой происходит итерация, а это, как уже отмечалось в предыдущих разделах, практически неэффективно.

Благодаря наличию таких шаблонов транслятор может определять границы условного оператора и границы цикла, а также различать их между собой. Это является очень важным моментом, так как именно по тому, какие переменные изменяются внутри ветвей условного перехода или цикла, определяется, какие версии, возникающие при выполнении или невыполнении условия, добавляются в данные переменные.

Стоит также отметить, что транслятор всегда подразумевает наличие какого-либо из данных шаблонов, то есть подразумевается, что инструкции в программе для JVM действительно были получены именно путем компиляции программы высокоуровневого языка программирования, например Java. В языках программирования вроде Java на уровне синтаксиса практически полностью запрещается использование оператора безусловного перехода, его использование ведет к ошибке компиляции. Это важно, потому что работа с абсолютно произвольным потоком исполнения (который обычно и создает оператор безусловного перехода) приводит к разрастанию размеров итоговой кодировки из-за огромного числа состояний и возможных путей исполнения программы, что является неэффективным.

2.7 Минимизация And-Inverter графов, генерируемых системой трансляции

При трансляции программ важно создавать такую модель транслятора, чтобы она генерировала как можно меньшие кодировки (но при этом эквивалентные) и содержала в себе минимум избыточной информации. Для этого в программном комплексе присутствуют некоторые оптимизации, которые минимизируют итоговые кодировки.

Одна из таких оптимизаций — переиспользование кодировок перемен-

ных, имеющих известное значение во время исполнения (т.е. констант). Например, если в программе несколько раз создавались переменные, имеющие одно и то же значение (такое может быть, например, в циклах), то можно закодировать эту константу с помощью какой-нибудь переменной трансляции (естественным образом через бинарное представление константы). В дальнейшем можно не создавать новую переменную трансляции для данной константы, а переиспользовать уже имеющуюся. Для этого в программном комплексе существует словарь, отображающий константы в соотвествующие им переменные трансляции. Такая несложная оптимизация тем не менее достаточно хорошо уменьшает объем кодировки, так как, например, операции вроде целочисленного инкремента больше не требуют каждый раз создавать новую переменную трансляции для константы равной 1.

Еще одной достаточно очевидной, тем не менее очень эффективной, оптимизацией, связанной с переменными с известным во время исполнения значением, является учет этого значения при анализе условного оператора и при исполнении цикла. Например, если известно, что переменная $\texttt{x} = \texttt{10}$, и дальше в программе встречается условное выражение вида $\texttt{if x} < \texttt{C}$ (конечно, не в таком виде, а в байткоде — но транслятор понимает, что это именно такое условное выражение), где C — некоторая константа, то программный комплекс автоматически исполняет либо только `if`-ветвь, либо только `else`-ветвь (в зависимости от того, выполняется или нет условное выражение). Это позволяет избежать добавления в систему достаточно объемного уравнения и, как следствие, размер и сложность итоговой кодировки сильно уменьшается. С циклами такая же логика: если в программе встречается выражение вроде $\texttt{for}\,(\texttt{int i} = \texttt{0};\ \texttt{i} < \texttt{x};\ \texttt{i}++)$, то транслятор понимает, что нужно просто 10 раз оттранслировать инструкции в цикле.

В программном комплексе также присутствует оптимизация, связанная с версиями переменных трансляции. Если, например, у переменной трансляции z в множестве версий есть две пары (p_1, C) и (p_2, C), то транслятор превращает их в одну пару (p_3, C), добавляя в систему уравнение $p_3 \equiv (p_1 \vee p_2) = 1$. Так происходит с абсолютно всеми парами, в которых константа совпадает.

Оптимизации, описанные выше, хоть и достаточно заметно уменьшают объем финальной кодировки, тем не менее намного лучше с этим справляется

программа ABC [17], работающая уже с итоговой кодировкой в формате .aig. Одной из самых эффективных команд в данной программе является команда fraig, преобразующая And-Inverter граф в эквивалентный функционально-редуцированный. Эту минимизацию можно применять и к создаваемым майтерам. В дальнейших экспериментах также будут рассматриваться графы, минизированные через эту команду.

Глава 3. Эксперименты и результаты

В данном разделе рассматриваются кодировки операций суммы и умножения. Их корректность устанавливается через подстановку конкретных значений в схему и проверку выходных значений. Помимо этого, рассматриваются кодировки некоторых криптографических функций, корректность которых устанавливается через создание майтера с референтными кодировками, после чего сравниваются размеры итоговых кодировок до и после минимизации с помощью `ABC` [17]. Также рассматриваются кодировки функций следующих сортировок: Bubble, Insertion, Selection, Pancake. Они тоже сравниваются с референтными кодировками, а также рассматривается время решения майтеров для пар различных сортировок, кодировки которого генерирует программный комплекс `transbyte`.

Референтными кодировками являются кодировки, генерируемые программным комплексом `transalg` [14]. Это доменно-специфичный программный комплекс, принимающий на вход программу, написанную на Си-подобном языке TA, и выдающий кодировку в формате `.aag`. Тем не менее, с помощью этого программного комплекса можно закодировать различные криптографические функции и алгоритмы сортировок.

Все кодировки в дальнейшем рассматриваются в бинарном AIGER-формате `.aig`, если не оговорено иное. Из текстового формата `.aag` в бинарный `.aig` кодировку можно перевести с помощью инструмента `aigtoaig` [2], как отмечалось в предыдущих разделах.

Для минимизации использовалась программа ABC [17], команда `fraig`.

В качестве SAT-решателя используется `kissat` [3]. Командой для запуска является `./kissat -- --unsat encoding.cnf`, если не оговорено иное.

Все замеры производились на ноутбуке с операционной системой Ubuntu 22.04, ядром linux-5.19.0-41-generic, процессором Intel Core i5-1135G7 с частотой 4200 МГц и 24GB оперативной памяти.

3.1 Сумма и умножение

Функции суммы и умножения из листинга Java программы 1 проверялись на корректность следующим образом: брались 2000 случайных чисел,

после чего первая 1000 использовалась как вход для первой переменной, а вторая 1000 как вход для второй. Т.е. всего перебирались $1000 \cdot 1000$ случайных входов на каждую функцию. После чего в итоговую кодировку подставлялись эти входы в бинарном виде, а схема решалась SAT-решателем, и проверялось, что выходы SAT-решателя, собранные в бинарные последовательности и преобразованные в целочисленные 32-битные переменные, соответствуют нужному результату (для суммы или умножения, соответственно). Дополнительно проверялись входы, которые приводят к переполнению 32-битной целочисленной переменной. На нескольких таких запусках было установлено, что кодировки суммы и умножения действительно задают соответствующую операцию.

3.2 Криптографические функции: LFSR, A5/1 Generator, Wolfram Generator

	LFSR	A5/1	Wolfram
transbyte	1327 / 1308	38216 / 38152	73856 / 73728
transalg	1000 / 981	29926 / 29862	49280 / 49152

Таблица 1: Размеры схем криптографических функций до минимизации

	LFSR	A5/1	Wolfram
transbyte	1000 / 981	28146 / 28082	49280 / 49152
transalg	1000 / 981	28146 / 28082	49280 / 49152

Таблица 2: Размеры схем криптографических функций после минимизации

	LFSR	A5/1	Wolfram
Кол-во записей	2	2	2
Макс. индекс переменных	66	21	130

Таблица 3: Размеры майтеров (в формате DIMACS) после минимизации

Корректность функций регистра сдвига с линейной обратной связью (Linear Feedback Shift Register, LFSR) (листинг на Java 2), генератора A5/1 (листинг на Java 3) и генератора Wolfram (листинг на Java 4) проверялась

через создание майтера с референтными кодировками. Созданные майтеры передавались в SAT-решатель. Было установлено, что все майтеры являются невыполнимыми, т.е. схемы, генерируемые разработанным программным комплексом `transbyte`, являются эквивалентными референтным схемам.

Данные о размерах схем в формате `.aig` приведены в таблице 1. Данные записаны в виде «количество гейтов в схеме / количество переменных в схеме». В таблице 2 приведены размеры схем после минимизации. В таблице 3 приведено количество записей в КНФ майтеров, в формате DIMACS [11], которые были предварительно минимизированы. Примечательно то, что количество записей во всех майтерах равно двум, т.е. кодировки настолько похожи между собой, что минимизация сразу же сводит итоговую КНФ к виду $x \wedge \neg x$. Время решения каждого такого майтера меньше, чем $0,01$ секунда, поэтому оно здесь не приводится.

3.3 Функции сортировки: Bubble, Selection, Insertion, Pancake

5x8	Bubble	Selection	Insertion	Pancake
transbyte	1180 / 1140	1190 / 1150	2694 / 2654	18602 / 18562
transalg	1790 / 1750	4867 / 4827	-	29926 / 29862

Таблица 4: Размеры схем функций сортировок до минимизации, размер 5x8

5x8	Bubble	Selection	Insertion	Pancake
transbyte	890 / 850	3890 / 3730	1165 / 1125	1585 / 1545
transalg	1090 / 1050	8290 / 8130	-	1853 / 1813

Таблица 5: Размеры схем функций сортировок после минимизации, размер 5x8

5x8	Bubble	Selection	Pancake
Кол-во записей	2	4513	2
Макс. индекс переменной	42	1074	42
Время решения майтера	0.00c	0.91c	0.00c

Таблица 6: Размеры майтеров с референтными кодировками (в формате DIMACS) после минимизации, размер 5x8

Функции сортировки, Bubble, Selection, Insertion и Pancake, представленные в листингах Java-программ 5, 6, 7 и 8, а также их вариации, но не с 32-битными переменными типа `int`, а с 8-битными переменными типа `byte`, рассматривались следующим образом. Создавались несколько кодировок с различными размерами входных данных: 5 переменных типа `byte` (размер 5x8), 5 переменных типа `int` (размер 5x32), 8 переменных типа `byte` (размер 8x8), и 7 переменных типа `int` (размер 7x32). Количество записей в этих кодировках в формате `.aig`, а также в референтных кодировках, до и после минимизации, для размера 5x8 представлено в таблицах 4, 5. Помимо этого, в таблице 6 представлены размеры минимизированных майтеров с референтными кодировками, а также времена решения этих майтеров. Информация о количестве записей для кодировок других размеров представлена в таблицах 9, 10, 14, 15, 19, 20. В таблицах 11, 16, 21 представлена информация о

майтерах с референтными кодировками для других размеров. К сожалению, в таблицах представлена не вся информация, связанная с референтными кодировками — программный комплекс `transalg` [14] генерирует некорректную схему для сортировки Insertion. Тем не менее, корректность кодировки для сортировки Insertion, генерируемой программным комплексом `transbyte`, устанавливалась с помощью создания майтеров с кодировками других сортировок, генерируемых этим же программным комплексом. Эти эксперименты будут описаны далее. Довольно примечательными являются две вещи: во-первых то, насколько минимизация в некоторых случаях уменьшает размеры кодировок; во-вторых то, что некоторые из майтеров на маленьких размерах превращаются в КНФ вида $x \wedge \neg x$, что, опять же, говорит о том, что некоторые кодировки функций сортировок достаточно сильно схожи с референтными кодировками.

Намного более интересным является рассмотрение майтеров между парами кодировок функций сортировок, генерируемых программным комплексом `transbyte`. SAT-решатель на каждый майтер между двумя парами сортировок говорит, что этот майтер неразрешим. По сути, таким образом проверяется, что две различные функции сортировки являются семантическими клонами. Количество записей для различных майтеров для размера 5x8 представлено в таблице 8. в формате «количество вентилей / максимальный индекс переменной». Стоит также обратить внимание, что «максимальный индекс переменной» в данном случае не является реальным количеством переменных в КНФ — обычно их количество меньше. Помимо количества записей, в таблице 7 представлено время решения майтеров SAT-решателем, т.е. время, спустя которое SAT-решатель говорит, что данный майтер является неразрешимым. Информация о майтерах для остальных размеров представлена в таблицах 12, 17, 22, 13, 18, 23. Довольно примечательным является тот факт, что увеличение количества переменных в массиве, передающимся в функцию сортировки, так же сильно влияет на время решения майтера, как и увеличение разрядности переменной (т.е. смена `byte` на `int`), при чем эта разница и в одном, и в другом случае колоссальна. Размерности больше, например 10x8 или 10x32 не приводятся в данной работе в силу того, что для размера 10x32 SAT-решатель производит вычисления более 12 часов, а

5x8	Selection	Insertion	Pancake
Bubble	0.31c	0.02c	0.86c
Selection	-	0.32c	1.14c
Insertion	-	-	0.60c

Таблица 7: Время решения майтеров для попарных сортировок (схемы из `transbyte`), размер 5x8

5x8	Selection	Insertion	Pancake
Bubble	2130 / 490	1904 / 457	3059 / 674
Selection	-	2580 / 603	3097 / 683
Insertion	-	-	3497 / 786

Таблица 8: Размеры майтеров для попарных сортировок (в формате DIMACS, схемы из `transbyte`) после минимизации, размер 5x8

10x8 более 5 часов. Точное время не устанавливалось, так как SAT-решатель был остановлен в силу ограниченности вычислительных мощностей и того, что полученные результаты времени работы для больших размеров уже будут представлять больше теоретический интерес, нежели практический. Тем не менее, на практике обычно достаточно небольших размеров для того, чтобы установить эквивалентность между двумя схемами, либо же обнаружить какие-либо ошибки в работе самого транслятора.

Данные эксперименты показывают, что подход с использованием трансляции и SAT-решателя действительно позволяет распознавать семантические клоны и может иметь практическую пользу при условии ограниченности модели транслятора, а также с ограничением на размеры входных данных.

Заключение

В данной работе исследовалась задача проверки эквивалентности программ. Для решения данной задачи применялся подход, использующий трансляцию программ в And-Inverter графы и SAT-решатель. Для трансляции программ, компилирующихся в байткод виртуальной машины Java, был разработан программный комплекс `transbyte`, доступный на GitHub по адресу https://github.com/eqimd/transbyte. Описана его работа и использование сторонних инструментов для минимизации схем и создания майтеров.

Была рассмотрена трансляция некоторых операций и криптографических функций, а также установлена корректность генерируемых кодировок с помощью референтных кодировок. Также была рассмотрена трансляция некоторых функций сортировок и приведены результаты экспериментов для задачи распознавания данных функций сортировок как семантических клонов.

Результаты работы демонстрируют, что описанный метод может применяться для верификации программ, допускающих относительно компактное представление в виде схем. Такими программами могут быть, например, арифметические функции, при реализации которых на Java высоки шансы ошибки программиста. Для таких функций могут быть созданы специальные библиотеки эталонных реализаций (возможно, на специальном предметно-ориентированном языке). Впоследствии при некоторой очередной промышленной реализации такой функции можно верифицировать корректность этой реализации за счет проверки ее эквивалентности соответствующему эталону. Именно на этом этапе может использоваться программный комплекс, представленный в настоящей работе: для обеих функций строится их AIG-представление и решается задача проверки эквивалентности соответствующих булевых схем.

Список литературы

[1] Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE Access*, 7:86121–86144, 2019.

[2] Armin Biere. AIGER Format and Toolbox.

[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

[4] S. A. Cook. The complexity of theorem proving procedures. In *Proceedings of the Third Annual ACM Symposium*, pages 151–158, New York, 1971. ACM.

[5] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, first edition, 1979.

[6] Clikt.

[7] ASM — all purpose Java bytecode manipulation and analysis framework.

[8] Apache Bytecode Engineering Library.

[9] transbyte — Translator of Java Bytecode to CIRCUIT-SAT.

[10] kotlin-logging.

[11] DIMACS Format.

[12] Simple Logging Facade for Java.

[13] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, jul 1976.

[14] Ilya Otpuschennikov, Alexander Semenov, and Stepan Kochemazov. Transalg: a tool for translating procedural descriptions of discrete functions to sat, 2015.

[15] Alexander Semenov, Ilya Otpuschennikov, Irina Gribanova, Oleg Zaikin, and Stepan Kochemazov. Translation of Algorithmic Descriptions of Discrete Functions to SAT with Applications to Cryptanalysis Problems. *Logical Methods in Computer Science*, Volume 16, Issue 1, March 2020.

[16] J. C. Shepherdson and H. E. Sturgis. Computability of recursive functions. *J. ACM*, 10(2):217–255, apr 1963.

[17] Berkeley Logic Synthesis and Verification Group. ABC — A System for Sequential Synthesis and Verification.

[18] Катленд Н. Вычислимость. Введение в теорию рекурсивных функций. *М.: Мир*, 1983.

[19] А.А. Семёнов. О преобразованиях Цейтина в логических уравнениях. *Прикладная дискретная математика*, 2009.

[20] Отпущенников И.В. Семёнов А.А. Технология трансляции комбинаторных проблем в булевы уравнения. *Прикладная дискретная математика*, № 1, 96 – 115, 2011.

[21] Г.С. Цейтин. О сложности вывода в исчислении высказываний. *Записки научных семинаров ЛОМИ АН СССР*, 8:234–259, 1968.

Приложения

Листинги

В данном разделе приведены различные листинги Java-программ, байт-код которых использовался для трансляции и экспериментов.

Листинг 1: Функции суммы и умножения двух целочисленных переменных

```
public static void Sum(int a, int b) {
    return a + b;
}

public static void Multiply(int a, int b) {
    return a * b;
}
```

Листинг 2: Linear Feedback Shift Register (LFSR)

```
class LFSR {
    public static boolean[] get_lfsr(boolean[] reg) {
        int len = 128;

        boolean[] output = new boolean[len];
        for (int i = 0; i < len; i++) {
            output[i] = shiftReg(reg);
        }

        return output;
    }

    static boolean shiftReg(boolean[] reg) {
        boolean x = reg[18];
        boolean y = reg[18]^reg[17]^reg[16]^reg[13];
        for (int j = 18; j > 0; --j) {
            reg[j] = reg[j - 1];
```

```
        }

        reg[0] = y;

        return x;
    }
}
```

Листинг 3: A5/1 Generator

```
class A5_1 {
    public static boolean[] generate(
        boolean[] regA,
        boolean[] regB,
        boolean[] regC
    ) {
        int len = 128;

        boolean[] result = new boolean[len];

        int midA = 8;
        int midB = 10;
        int midC = 10;
        boolean maj;

        for (int i = 0; i < len; i++) {
            maj = majority(
                regA[midA],
                regB[midB],
                regC[midC]
            );

            if (!(maj^regA[midA])) {
                shift_rslosA(regA);
```

```
            }

            if (!(maj^(regB[midB]))) {
                shift_rslosB(regB);
            }

            if(!(maj^(regC[midC]))) {
                shift_rslosC(regC);
            }

            result[i] = regA[18]^regB[21]^regC[22];
        }

        return result;
    }

    public static boolean majority(
        boolean A,
        boolean B,
        boolean C
    ) {
        return A&B | A&C | B&C;
    }

    public static boolean shift_rslosA(
        boolean[] reg
    ) {
        boolean x = reg[18];
        boolean y = reg[18]^reg[17]^reg[16]^reg[13];

        for (int j = 18; j > 0; j--) {
            reg[j] = reg[j-1];
        }
```

```
        reg[0] = y;

        return x;
    }

    public static boolean shift_rslosB(
        boolean[] reg
    ) {
        boolean x = reg[21];
        boolean y = reg[21]^reg[20];

        for (int j = 21; j > 0; j--) {
            reg[j] = reg[j-1];
        }

        reg[0] = y;

        return x;
    }

    public static boolean shift_rslosC(
        boolean[] reg
    ) {
        boolean x = reg[22];
        boolean y = reg[22]^reg[21]^reg[20]^reg[7];

        for (int j = 22; j > 0; j--) {
            reg[j] = reg[j-1];
        }

        reg[0] = y;
```

```
        return x;
    }
}
```

Листинг 4: Wolfram Generator

```
class WolframGenerator {
    public static boolean[] generate(
        boolean[] reg
    ) {
        int reg_len = 128;

        boolean[] buff = new boolean[reg_len];
        boolean[] result = new boolean[reg_len];


        for (int i = 0; i < reg_len; i++) {
            update(reg, buff);
            result[i] = reg[0];
        }

        return result;
    }

    public static void update(
        boolean[] reg,
        boolean[] buff
    ) {
        int reg_len = 128;

        boolean left;
        boolean right;

        for (int i = 0; i < reg_len; i++) {
```

```
            if (i == 0) {
                left = reg[reg_len - 1];
            } else {
                left = reg[i - 1];
            }

            if (i + 1 == reg_len) {
                right = reg[0];
            } else {
                right = reg[i + 1];
            }

            buff[i] = left ^ (reg[i] | right);
        }

        for (int i = 0; i < reg_len; i++) {
            reg[i] = buff[i];
        }
    }
}
```

Листинг 5: Bubble Sorting

```
public static int[] bubbleSort(int[] sortArr) {
    for (int i = 0; i < sortArr.length - 1; i++) {
        for(
            int j = 0;
            j < sortArr.length - i - 1;
            j++
        ) {
            if(sortArr[j + 1] < sortArr[j]) {
                int swap = sortArr[j];
                sortArr[j] = sortArr[j + 1];
                sortArr[j + 1] = swap;
```

```
            }
        }
    }

    return sortArr;
}
```

Листинг 6: Selection Sorting

```
public static int[] selectionSort(int[] sortArr) {
    for (int i = 0; i < sortArr.length; i++) {
        int pos = i;
        int min = sortArr[i];
        for (
            int j = i + 1;
            j < sortArr.length;
            j++
        ) {
            if (sortArr[j] < min) {
                pos = j;
                min = sortArr[j];
            }
        }
        sortArr[pos] = sortArr[i];
        sortArr[i] = min;
    }

    return sortArr;
}
```

Листинг 7: Insertion Sorting

```
public static int[] insertionSort(int[] sortArr) {
    int j;
    for (int i = 1; i < sortArr.length; i++) {
```

```
        int swap = sortArr[i];
        for (
            j = i;
            j > 0 && swap < sortArr[j - 1];
            j--
        ) {
            sortArr[j] = sortArr[j - 1];
        }
        sortArr[j] = swap;
    }

    return sortArr;
}
```

Листинг 8: Pancake Sorting

```
public static void pancakeSortFlip(int p, int[] data)
    {
    for (int i = 0; i < p; i++) {
        p--;
        int tmp = data[i];
        data[i] = data[p];
        data[p] = tmp;
    }
}

public static int[] pancakeSort(int[] data) {
    int k = data.length;

    for (int i = k - 1; i >= 0; i--) {
        int max = 0;

        for (int j = 1;  j <= i; j++) {
            for (int d = 0; d <= i; d++) {
```

```
                if (
                    (max == d) && (data[d] < data[j])
                ) {
                    max = j;
                }
            }
        }

        for (int t = 1; t < i; t++) {
            if (max == t) {
                pancakeSortFlip(t + 1, data);
            }
        }

        if (max != i) {
            pancakeSortFlip(i + 1, data);
        }
    }

    return data;
}
```

Таблицы

В данном разделе приведены таблицы с различными данными о количестве записей в разных кодировках и майтерах, а также времена решения майтеров.

8x8	Bubble	Selection	Insertion	Pancake
transbyte	3256 / 3192	3312 / 3248	10410 / 10346	78035 / 77971
transalg	4964 / 4900	20706 / 20642	-	19692 / 19628

Таблица 9: Размеры схем функций сортировок до минимизации, размер 8x8

8x8	Bubble	Selection	Insertion	Pancake
transbyte	2444 / 2380	2581 / 2517	3637 / 3573	6522 / 6458
transalg	3004 / 2940	6497 / 6433	-	7552 / 7488

Таблица 10: Размеры схем функций сортировок после минимизации, размер 8x8

8x8	Bubble	Selection	Pancake
Кол-во записей	2	14567	2
Макс. индекс переменной	66	3403	66
Время решения майтера	0.00с	5с	0.00с

Таблица 11: Размеры майтеров с референтными кодировками (в формате DIMACS) после минимизации, размер 8x8

8x8	Selection	Insertion	Pancake
Bubble	6194 / 1384	7195 / 1675	10869 / 2332
Selection	-	8160 / 1879	11006 / 2361
Insertion	-	-	12829 / 2828

Таблица 12: Размеры майтеров для попарных сортировок (в формате DIMACS, схемы из `transbyte`) после минимизации, размер 8x8

8x8	Selection	Insertion	Pancake
Bubble	3м 47с	0.52с	13м 9с
Selection	-	4м 51с	36м 31с
Insertion	-	-	11м 50с

Таблица 13: Время решения майтеров для попарных сортировок (схемы из `transbyte`), размер 8x8

5x32	Bubble	Selection	Insertion	Pancake
transbyte	4900 / 4740	4910 / 4750	10830 / 10670	27938 / 27778
transalg	22550 / 22390	68155 / 67995	-	68695 / 68535

Таблица 14: Размеры схем функций сортировок до минимизации, размер 5x32

5x32	Bubble	Selection	Insertion	Pancake
transbyte	3890 / 3730	3927 / 2767	4962 / 4802	6465 / 6305
transalg	8290 / 8130	14550 / 14390	-	15125 / 14965

Таблица 15: Размеры схем функций сортировок после минимизации, размер 5x32

5x32	Bubble	Selection	Pancake
Кол-во записей	13226	31670	25453
Макс. индекс переменной	3091	7046	5760
Время решения майтера	0.01c	12.41c	0.02c

Таблица 16: Размеры майтеров с референтными кодировками (в формате DIMACS) после минимизации, размер 5x32

5x32	Selection	Insertion	Pancake
Bubble	9225 / 2090	9707 / 2243	12355 / 2695
Selection	-	11337 / 2551	12397 / 2705
Insertion	-	-	14407 / 3141

Таблица 17: Размеры майтеров попарных сортировок (в формате DIMACS, схемы из `transbyte`) после минимизации, размер 5x32

5x32	Selection	Insertion	Pancake
Bubble	3.83c	0.28c	8.12c
Selection	-	3.21c	9.67c
Insertion	-	-	5.84c

Таблица 18: Время решения майтеров для попарных сортировок (схемы из `transbyte`), размер 5x32

7x32	Bubble	Selection	Insertion	Pancake
transbyte	10178 / 9954	10213 / 9989	28361 / 28137	77601 / 77377
transalg	47243 / 47019	193820 / 193596	-	182114 / 181890

Таблица 19: Размеры схем функций сортировок до минимизации, размер 7x32

7x32	Bubble	Selection	Insertion	Pancake
transbyte	8057 / 7833	8152 / 7928	11719 / 11495	17504 / 17280
transalg	17297 / 17073	39220 / 38996	-	41712 / 41488

Таблица 20: Размеры схем функций сортировок после минимизации, размер 7x32

7x32	Bubble	Selection	Pancake
Кол-во записей	2	766554	46668
Макс. индекс переменной	226	16917	10316
Время решения майтера	0.00с	36с	0.34с

Таблица 21: Размеры майтеров с референтными кодировками (в формате DIMACS) после минимизации, размер 7x32

7x32	Selection	Insertion	Pancake
Bubble	19331 / 4316	24340 / 5525	30013 / 6417
Selection	-	26390 / 5893	30099 / 6436
Insertion	-	-	36964 / 7971

Таблица 22: Размеры майтеров для попарных сортировок (в формате DIMACS, схемы из `transbyte`) после минимизации, размер 7x32

7x32	Selection	Insertion	Pancake
Bubble	11м 5с	3с	23м 37с
Selection	-	11м 16с	3ч 10м 19с
Insertion	-	-	15м 45с

Таблица 23: Время решения майтеров для попарных сортировок (схемы из `transbyte`), размер 7x32