Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
канд. техн. наук, доцент
_____ С. А. Останин
«26» января 2021 г.

ДИПЛОМНАЯ РАБОТА

Разработка декомпилятора байткода LuaJIT

на основной образовательной программе подготовки специалистов
«Анализ безопасности компьютерных систем»
специальности подготовки 10.05.01 Компьютерная безопасность

Станчин Максим Андреевич

Руководитель ВКР
доцент кафедры компьютерной безопасности ИПМКН
_____ Д.Н. Колегов
«26» января 2021 г.

Автор работы
студент группы № 1155
_____ М.А. Станчин

Томск – 2021

Министерство науки и высшего образования Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Институт прикладной математики и компьютерных наук
Кафедра компьютерной безопасности

УТВЕРЖДАЮ
Руководитель ООП
канд. техн. наук, доцент
_____ С. А. Останин
«9» сентября 2020 г.

ЗАДАНИЕ

по подготовке ВКР специалиста студенту группы 1155 Станчину Максиму Андреевичу.

1. Тема ВКР: «Разработка декомпилятора байткода LuaJIT».
2. Срок сдачи студентом выполненной дипломной работы:
   на кафедре: 19 января 2021
   в ГЭК: 1 февраля 2021
3. Цель работы: разработка декомпилятора байткода LuaJIT, использующего методы устраняющие недостатки существующих декомпиляторов.
4. Основные разделы работы: недостатки подходов существующих декомпиляторов байткода LuaJIT, каскадный анализ, загрузка файла, первичная трансляция инструкций байткода, анализ графа потока исполнения. Объекты исследования: байткод LuaJIT, методы декомпиляции. Методы исследования: теоретическое исследование и практическая реализация.
5. Работа выполняется по заданию кафедры компьютерной безопасности Томского государственного университета.
6. Дата выдачи задания 1 сентября 2020 г.

Руководитель ВКР
доцент кафедры компьютерной
безопасности ИПМКН _____ / Д.Н. Колегов/
Задание принял к исполнению
студент группы № 1155 _____ / М.А. Станчин/

РЕФЕРАТ

Дипломная работа содержит 24 страницы, 12 таблиц, 9 литературных источников.

LUAJIT, REVERSE ENGINEERING, ДЕКОМПИЛЯЦИЯ, БАЙТКОД, ГРАФ ПОТОКА УПРАВЛЕНИЯ.

Объекты исследования: байткод LuaJIT, методы декомпиляции.

Цель работы: разработка декомпилятора байткода LuaJIT, использующего методы устраняющие недостатки существующих декомпиляторов.

Метод исследования: Теоретическое исследование и практическая реализация.

Результат: Описаны недостатки подходов используемых существующими декомпиляторами байткода LuaJIT. Разработан декомпилятор применяющий методы и подходы лишенные описанных недостатков. Реализован прототип декомпилятора байткода LuaJIT. Предложены две мутации графа потока исполнения, для повышения площади покрываемого байткода.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ 5

ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ 7

1 Постановка задачи 8

2 Этапы декомпиляции 9

2.1 Каскадный анализ 9

2.2 Загрузка файла 10

2.2.1 Структура файла с байткодом 10

2.2.2 Анализ структуры файла 10

2.3 Промежуточное представление 12

2.4 Анализ графа потока исполнения 13

2.4.1 Построение CFG 14

2.4.2 Шаблоны управляющих конструкций 14

2.4.3 Схлопывание последовательностей узлов 15

2.4.4 Дублирование терминирующего узла 16

2.5 Работа с переменными 17

2.5.1 Удаление неиспользуемых переменных 17

2.5.2 Подставнока переменной 18

3 Программная реализация 19

ЗАКЛЮЧЕНИЕ 20

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ 20

ПРИЛОЖЕНИЕ А Таблицы трансляции инструкций 22

ВВЕДЕНИЕ

Основная задача специалиста по обратной разработке (англ. reverse engineering) заключается в том, чтобы по данному ему бинарному файлу исследовать логику работы приложения, реализованного в этом файле, а одним из самых главных инструментов статического анализа (анализа без фактического исполнения кода) является декомпилятор. В большинстве случаев подобные файлы содержат машинный код в том или ином формате, полученный в результате компиляции кода, записанном на некотором языке программирования.

Однако существуют языки, для которых конечным результатом компиляции является не машинный код, а байткод, используемый интерпретатором или виртуальной машиной программного окружения (англ. runtime) языка. Одним из таких языков является LuaJIT [1], представляющий собой модифицированную версию популярного языка программирования Lua. Хотя LuaJIT имеет дополнительную возможность компилировать исходный код непосредственно в машинный код посредством JIT-компиляции, в данной работе будет рассматриваться именно работа с байткодом.

Поскольку байткод является более высокоуровневым представлением в отличии от традиционного машинного кода, часто его возможно декомпилировать обратно в исходный код, ввиду чего существующие декомпиляторы LuaJIT-байткода [2] построены на идее полного однозначного восстановления исходного кода. Это может быть выполнимым, если в ходе генерации выходного бинарного файла был использован стандартный компилятор, в таком случае задача декомпилятора сводится к обращению описанных механизмов и алгоритмов используемых при компиляции. Однако в случае, когда выходной бинарный файл изначально был получен иным способом (например скомпилирован не исходный код на языке *LuaJIT*, а набор инструкций транслирован напрямую в байткод), или в процессе компиляции к байткоду были применены нестандартные механизмы оптимизации или защиты, процесс восстановления псевдокода из такого файла уже отличается, а алгоритмы и идеи, заложенные в существующих декомпиляторах, не работают в полной мере, либо не работают совсем, исходя из чего задача разработки и внедрения более универсальных методов декомпиляции становится актуальной.

Целью данной работы является разработка и реализация декомпилятора байткода *LuaJIT*, использующего принципы и подходы позволяющие обеспечить более гибкий анализ, а также устраняющие недостатки существующих аналогов.

В первой главе работы описываются недостатки подходов существующих декомпиляторов байткода LuaJIT и возможные пути их исправления.

Во второй главе описывается архитектура декомпилятора, с краткими описаниями предлагаемых подходов и модулей, выполняющих этапы анализа и декомпиляции. Этап загрузки и дизассемблирования: формирования дерева прототипов, применяемые технологии для описания структуры файла, пример вывода дизассемблированного ко-

да. Этап трансляции дизассемблированного кода в промежуточное представление в виде псевдокода LuaJIT. Работа с графом потока исполнения функций: формирования графа, восстановление конструкций передачи управления, применяемые к графу мутации.

В заключении подводятся итоги проделанной работы.

ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ

Приведем основные термины, используемые в контексте описания процесса работы декомпилятора.

Reverse engineering (обратная разработка) - процесс исследования готового устройства или программы, а также предоставленной документации с целью понять принцип и логику его работы.

LuaJIT (Lua Just-In-Time-compiler) - модифицированная версия ЯП Lua, с возможностью just-in-time компиляции.

Control-flow graph (CFG, граф потока управления) - множество всех возможных путей исполнения программы, представленное в виде направленного графа.

Basic block (базовый блок, ББ) - понятие, обозначающее последовательность инструкций или кода, имеющую одну точку входа, одну точку выхода и не содержащую инструкций передачи управления ранее точки выхода.

Bytecode (байткод) - форма представления инструкций и данных используемая для эффективной работы интерпретаторов ЯП.

IR (intermediate representation) - промежуточное представление кода или данных в рамках определенного приложения (компилятор, виртуальная машина и тп.), как правило проектируемое и используемое для упрощения работы с оптимизацией или трансляцией.

Static Single Assignment (SSA) - форма записи, используемая компиляторами, в которой каждой переменной значение присваивается лишь единожды.

1 Постановка задачи

В настоящее время декомпилятор байткода LuaJIT [2], а также его расширение [3] разработаны таким образом, что в процессе анализа пытаются восстановить код так, как будто он был скомпилирован стандартными средствами языка, ввиду чего имеют следующие недостатки:

1. В ходе восстановления исходного кода приложения, пользователю либо отдается финальный псевдокод, либо декомпилятор завершает свою работу без результата, из-за чего в тех случаях, когда файл не может быть в полной мере обработан и проанализирован декомпилятором, а также при возникновении непридвиденных ошибок в процессе декомпиляции, пользователь не получает никакой информации о логике работы приложения

2. Вместо анализа графа потока исполнения, работа происходит над конкретными инструкциями (проивзодится поиск конкретных инструкций с аргументами конкретного вида), из-за чего крайне сильно снижается покрытие анализом возможных конструкций внутри байткода

Данные подходы накладывают существенные ограничения на возможности анализа при декомпиляции и не соответствуют современным тенденциям разработки декомпиляторов. В данной работе разработан и реализован декомпилятор байткода LuaJIT лишенный описанных недостатков и использующий следующие подходы:

1. Процесс всего анализа декомпилируемого файла делится на этапы (каскады). Во время работы каждого из этапов происходит формирование промежуточного результата, который отдается пользователю в случае если анализ на следующим этапе не может быть завершен.

2. Высокоуровневый код на языке LuaJIT восстанавливается в несколько этапов, что позволяет обеспечить больше гибкости анализа в процессе формирования псевдокода: сначала генерируется код, получаемый путём трансляции инструкций в эквивалетные выражения языка, затем восстанавливаются структуры управления.

3. Для восстановления таких конструкций управления исполнением как циклы (while..do, repeat..until, for) и операторы ветвления (if..then..else) строится граф потока исполнения (CFG) и вместо поиска конкретных инструкций восстановление происходит основываясь на закономерностях, выявляемых в графе.

2 Этапы декомпиляции

Процесс декомпиляции является комплексной задачей и состоит из нескольких этапов анализа, в связи с чем декомпилятор разбит на отдельные модули отвечающие за соответствующие этапы.

2.1 Каскадный анализ

Для получения максимально возможного, с точки зрения автора, количества информации о логике работы приложения предлагается использовать каскадный подход при анализе кода приложения. Смысл данного подхода заключается в формировании промежуточных результатов по мере продвижения анализа приложения, и в случае возникновения ошибки во время работы очередного этапа, пользователю предоставляется результат работы предыдущего этапа. Примером декомпилятора использующего подобный подход является [4].

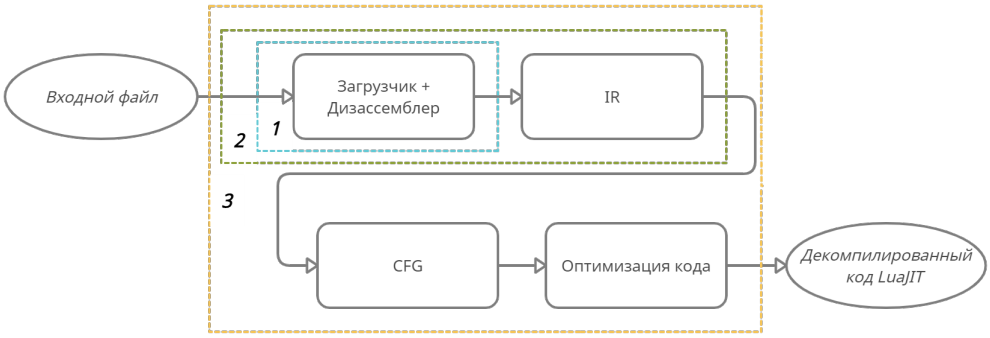


Рисунок 1 — Схематичное изображение основных этапов работы декомпилятора

На Рисунке 1 изображены этапы генерации кода декомпилятором, данный подход позволяет в процессе декомпиляции файла подготавливать промежуточные представления, чтобы в случае возникновения непредвиденных ошибок или невозможности восстановления кода приложения в полной мере, предоставить реверс инженеру какую-то информацию.

Так как первым этапом является загрузка файла и дизассемблирование байткода, первым, возможным к предоставлению пользователю, результатом является листинг инструкций с дополнительной информацией относительно прототипов функций (количество аргументов, наличие замыкаемых переменных и тп.), используемых каждой функцией литералов (целочисленных значений, строк, таблиц), а также иерархии прототипов.

Далее для каждой функции строится граф потока исполнения и производится первичная трансляция в псевдокод. Возможным результатом, представляемым пользователю на данном этапе, является псевдокод LuaJIT, однако на данном этапе чтение псевдокода

может быть затруднено из-за отсутствия каких-либо оптимизаций или таких конструкций передачи управления как циклы *while/repeat/for*, а все переходы производятся с помощью операторов *goto*.

Финальным этапом является восстановление структур передачи управления, так как на этом этапе уже есть рабочий псевдокод LuaJIT. В случае когда у декомпилятора не получается восстановить управляющие конструкции, псевдокод остается без изменений.

2.2 Загрузка файла

Основная задача данного этапа - загрузка файла в память декомпилятора, разбор заголовков и прототипов функций, подготовка прототипов для передачи следующим этапам анализа, а также формирование первого представления кода для пользователя ввиде дизасемблерного листинга.

2.2.1 Структура файла с байткодом

Файл с байткодом LuaJIT представляет из себя несколько служебных заголовков после которых идет массив прототипов функций внутри каждого из которых находятся все используемые соответствующим прототипом константы и байткод. С описанием структуры файла можно ознакомиться в [5].

Однако не смотря на то, что внутри файла структура является плоским массивом, набор прототипов, загруженных в память, представляет из себя древовидную структуру.

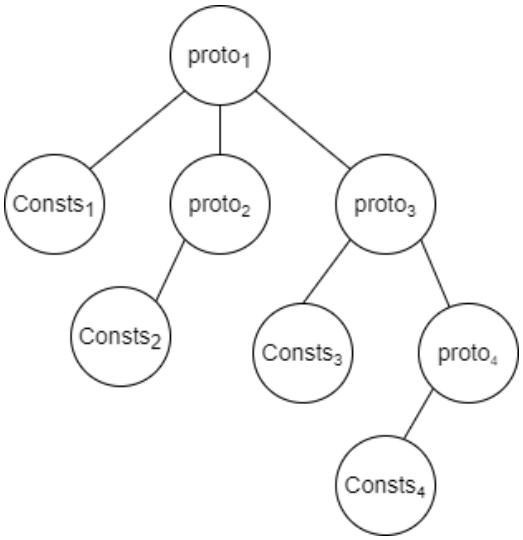


Рисунок 2 — Структура кода в памяти

2.2.2 Анализ структуры файла

Обработка файла на данном этапе предусматривает следующую логику работы:

1. входной файл с байткодом полностью анализируется модулем разбора: разбираются заголовки файла, прототипы функций, таблицы констант каждого прототипа, ссылки на замыкаемые переменные, количество аргументов функций, в результате работы которого, получается дерево объектов

2. полученное дерево объектов передается модулю обработки опкодов и непосрдественно участков кода.

В случае необходимости вывода результата работы данного этапа пользователю, производится обход дерева прототипов в глубину, в результате которого выводится листинг инструкций байткода с дополнительной информацией об иерархии прототипов, листинг 1 демонстрирует результат такой процедуры.

Листинг 1 — Пример вывода модуля дизассемблирования

```
——— BYTECODE ———
Function: foo
ChildOf: function_0
Children: []
Upvalues: [0]

0001     UGET     0    0        ; b
0002     KSHORT   1  1000
0003     TSETB    1    0    1
0004     RET0     0    1


——— BYTECODE ———
Function: function_0
ChildOf: None
Children: [str("test"), str("print"), str("foo"), foo, ([nil,1,2,3,4], {})]
Upvalues: None

0001     TDUP     0    0
0002     FNEW     1    1        ; foo prototype
0003     GSET     1    2        ; "foo"
0004     TGETB    1    0    1
0005     ISNEN    1    0        ; 1000
0006     JMP      10
0007     GGET     1    3        ; "print"
0008     KSTR     2    4        ; "test"
0009     CALL     1    1    2
0010 => GGET      1    2        ; "foo"
0011     CALL     1    1    1
0012     UCLO
0013 => RET0      0    1
```

2.3 Промежуточное представление

Как было сказано ранее, байткод LuaJIT является весьма высокоуровневым сам по себе, что позволяет однозначно транслировать большинство из его инструкций в псевдокод. Данный этап является вторым в процессе работы декомпилятора, что позволяет формировать первичный декомпилированный код практически сразу после загрузки файла в память декмопилятора. Отсутствие необходимости восстановления сложных конструкций языка на данном этапе позволяет обеспечивать практически полное покрытие обрабатываемого байткода. Исключениями являются только инструкции, не несущие никакой информации для пользователя, а имеющие значения исключительно для интерпретатора (например *LOOP*, *UCLO*) или некорректиные с точки зрения самого интерпретатора инструкции.

Данный этап не предусматривает контекстного анализа кода функций, так как главная задача здесь это - трансляция ввиде последовательности объектов для дальнейшей работы, а также формирование, возможно, трудночитаемого, однако совместимого с LuaJIT, псевдокода, который может быть исполнен интерпретатором. Фактически каждая инструкция на данном этапе просто транслируется в набор выражений LuaJIT выполняющие аналогичные действия.

Каждый слот функции представляется в виде переменной с индексом слота - var_i. В случае если переменная замыкается хотя бы одной из дочерних функций к её имени добавляется уникальный префикс (например название текущей функции) для избежания коллизий имен. Каждая инструкция, работающия со слотами, функций транслируется в выражение псевдокода, использующее соответствующие переменные. Листинги 2 и 3 демонстрируют пример такой трансляции.

Листинг 2 — Листинг дизассемблированной функции

```
0001      KSHORT   0   10
0002 =>   ISEQN    0    0        ; 0
0003      JMP      0012
0004      LOOP     0012
0005      GGET     1    0        ; "print"
0006      KSTR     2    1        ; "Value: "
0007      MOV      3    0
0008      CAT      2    2    3
0009      CALL     1    1    2
0010      SUBVN    0    0    1   ; 1
0011      JMP      0002
0012 =>   RET0     0    1
```

Листинг 3 — Результат первичной трансляции в псевдокод LuaJIT

```
1  local var_0, var_1, var_2, var_3
2  var_0 = 10
3  ::label_0::
4  if var_0 == 0 then goto label_1 end
5  var_1 = print
6  var_2 = "Value: "
7  var_3 = var_0
8  var_2 = var_2 .. var_3
9  var_1(var_2)
10 var_0 = var_0 - 1
11 goto label_0
12 ::label_1::
13 return
```

Код, в который транслируются инструкции на данном этапе, эквивалентен байткоду и совместим с интерпретатором LuaJIT, что крайне сильно облегчает задачу интегрирования собственного кода внутрь приложения для динамического анализа (например: добавление отладочных выводов на экран), а также изменения логики работы самого приложения.

С полным списком предлагаемых соотношений инструкций и псевдокода, а также форматом инструкций, можно ознакомиться в Приложении 3 к данной работе.

2.4 Анализ графа потока исполнения

На текущий момент, существующие решения декомпиляции байткода LuaJIT используют шаблонный метод восстановления конструкций языка, суть этого метода заключается в том, что так как стандартный компилятор LuaJIT в ходе своей работы такие управляющие конструкции языка как циклы *for*, *while*, *repeat/until*, операторы ветвления *if/then/else*, транслирует в набор инструкций байткода с одними и теми же закономерностями, данный метод полностью полагается на эти закономерности. В том случае, если не получается сопоставить шаблон набору инструкций, декомпилятор завершает работу.

Однако, существует подход восстановления конструкций языка, управляющих выполнением кода, рассматривающий каждую функцию не как последовательный набор инструкций, а как *CFG*, позволяющий обобщить работу с конструкциями передачи управления с помощью графов. Основные принципы данного подхода рассматриваются в [6, 7].

Так как метод шаблонного вычленения конструкций языка можно представить в виде работы с некоторым графом G , в данной главе рассматривается предлагаемый комбинированный вариант восстановления структуры кода, работающий не с конкретными инструкциями байткода, а непосредственно с *CFG* функций, но при этом использующий шаблонные способы восстановления и поиска конструкций передачи управления поверх графа для ускорения и упрощения анализа.

2.4.1 Построение CFG

Граф потока исполнения строится путём обхода в глубину, то есть при совершении обхода байткода функции, встречая инструкцию передачи управления, декомпилятор добавляет в граф очередной базовый блок, а обход рекурсивно продолжается с адреса, на который передается управление. В случае, если происходит передача управления внутрь уже существующего базового блока, он разбивается. Данный способ позволяет избавляться от недостижимых участков кода сразу на этапе формирования графа потока исполнения.

Далее условимся обозначать граф функции - G, стартовый узел в графе функции (базовый блок с которого начинается исполнение функции) - n_s, а все терминирующие узлы (узлы, содержащие инструкции возврата из функции, *RET0 RET1*, *RETM*, *CALLT*, на графе такие узлы не имеют исходящих ребер) - n_{e_i}.

2.4.2 Шаблоны управляющих конструкций

Типы управляющих конструкций разделяются на циклические и нециклические. Нециклическими являются операторы *if-then* и *if-then-else*. В процессе обработки *CFG* декомпилятор пытается найти в подграф удовлетворяющий одному из шаблонов указаных на рисунке 4. В случае если находится для n_s, n_f и n_t, подграф из узлов n_s, n_f и n_t заменяется на виртуальный узел восстановленной конструкции.

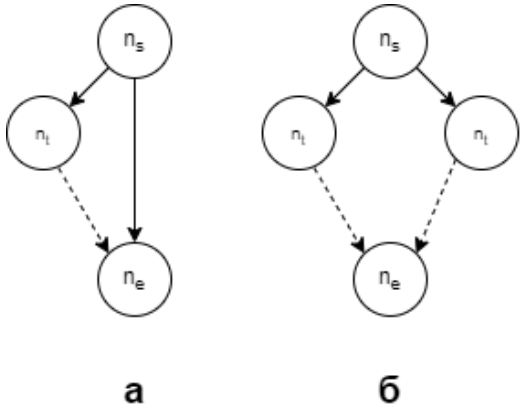


Рисунок 3 — Нецикличные подграфы, а) if-then-else, б) if-then

К циклическим конструкциям относятся *while-do*, *repeat-until*. Конструкция *for* задается отдельными специальными инструкциями LuaJIT, однако фактически имеет такой шаблон как и *while-do*. Если в *CFG* функции находится циклический подграф, то узел n_h, являющийся доминирующим для любого узла в подграфе, считается точкой входа в цикл, а все остальные узлы - телом цикла.

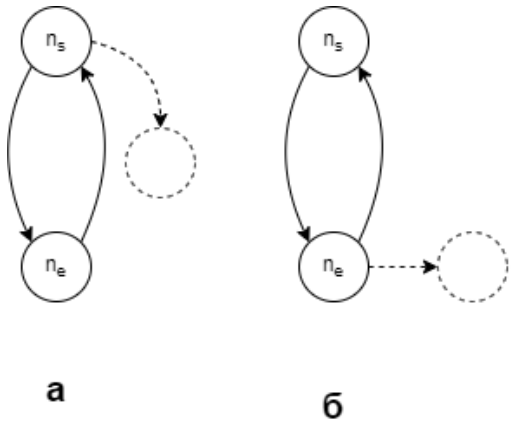


Рисунок 4 — Цикличные подграфы, а) while-do, б) repeat-until

Также нужно учитывать возможность наличия в графе функции циклических подграфов, не удовлетворяющих шаблонам. Условие выхода из таких циклических подграфов находится непосредственно в теле самого цикла, а не в крайних его узлах, как в случае *while-do* и *repeat-until*. Подобные подграфы в псевдокоде, генеруемом декомпилятором, представляются как бесконечные while-do циклы, в теле которых могут быть операторы *goto* или операторы *break* для выхода из цикла.

2.4.3 Схлопывание последовательностей узлов

Пусть по некоторой данной функции F имеется граф потока управления $G(N, E, n_s)$, где каждый узел $n \in N$ – базовый блок функции. Узел n_s – точка входа F (стартовый узел G). Ребро $e = (n_f, n_t) \in E$ - означает возможную передачу управления от блока n_f блоку n_t .

Мутация 1. Любой $S \subseteq G$, все узлы которого в G имеют только одно исходящее (за исключением конечного узла подграфа S) и одно входящее ребра (за исключением начального узла подграфа S), можно представить в виде одного неделимого базового блока (узла), при этом корректность работы данной последовательности не нарушится.

Если к цепочке узлов применяется описанная мутация, все инструкции передачи управления удаляются из всех мутируемых узлов кроме последнего.

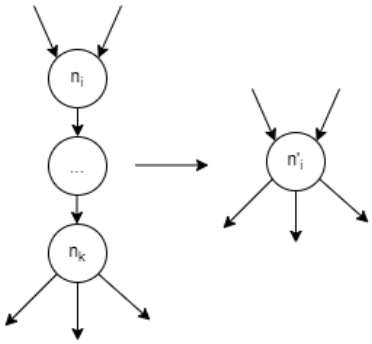


Рисунок 5 — Схлопывание последовательности узлов CFG

2.4.4 Дублирование терминирующего узла

В случаях, если существует такое исходящее ребро из тела рассматриваемого подграфа, что оно идёт к терминирующему узлу графа всей функции и данный подграф не сопоставляется ни с одним из шаблонов, то используется дублирование данного терминирующего узла.

Мутация 2. Если существует ребро от одного из узлов n_s, n_f, n_t нециклического подграфа к терминирующему узлу, и не существует ребра между узлом n_e нециклического подграфа и терминирующим узлом, то терминирующий узел копируется и полученная копия либо объединяется с одним из узлов n_f, n_t. В случае если какого-то из узлов нет, то становится одним из этих узлов.

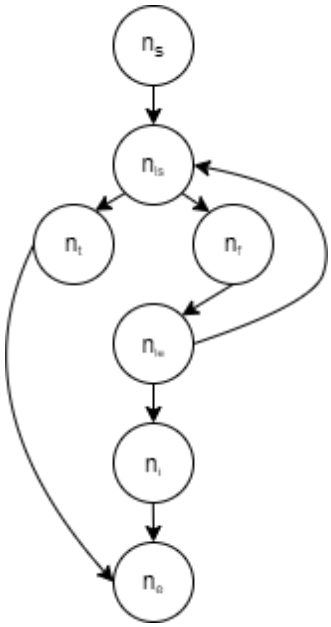


Рисунок 6 — Пример нетипичного выхода из цикла repeat...until

Рисунок 6 демонстрирует нестандартный выход из тела цикла repeat...until, без мутации *CFG* или применения оператора *goto*, получить высокоуровневый код LuaJIT не получится т.к. все циклы в случае выхода передают управление узлу находящемуся сразу за последним узлом самого цикла, в данном случае это узел n_i. Однако узел внутри оператора ветвления if-else передает управление узлу, несмежному с циклом, но являющимся терминирующим узлом *CFG* функции, что позволяет мутировать граф следующим образом: создается копия терминирующего узла n_e, назовем ее n'_e, исходящее ребро (n_t, n_e) заменяется на (n_t, n'_e). В результате получаются последовательности узлов $[n_t, n'_e]$ и $[n_i, n_e]$ полностью удовлетворяющие условиям мутации, описанной в 2.4.3. Таким образом дан-

ные преобразование позволяют транслировать *CFG* в стандартные конструкции языка, так как полученный граф удовлетворяет шаблонам.

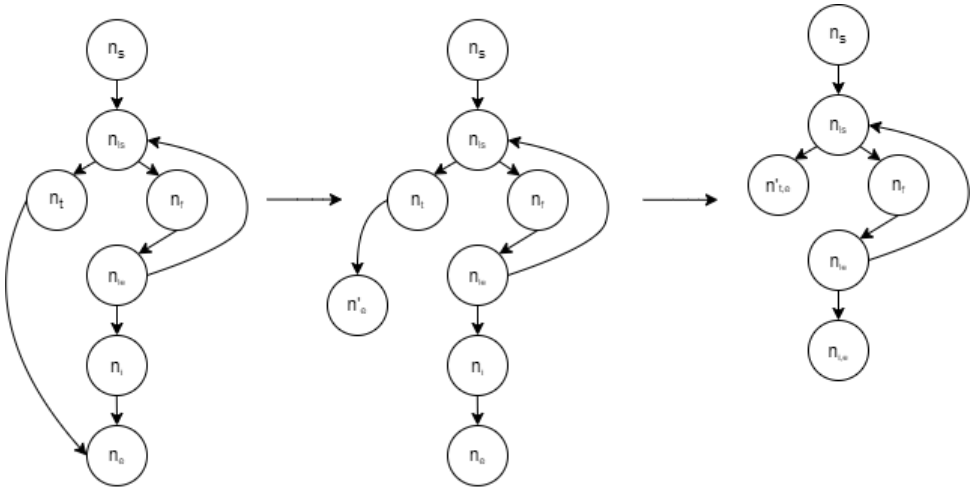


Рисунок 7 — Мутация CFG посредством дублирования терминирующего узла

В случае, если декомпилятору не удается сопоставить подграф какому-то из шаблонов управляющих конструкций с помощью мутаций, то в получаемом псевдокоде LuaJIT передача управления внутри данного подграфа остается в том же виде, в котором была на этапе первичной трансляции, то есть посредством использования операторов *goto*.

2.5 Работа с переменными

Для повышение удобочитаемости получаемого псевдокода используются оптимизации описываемые далее, однако они касаются исключительно локальных переменных функций, глобальные переменные остаются без изменений.

Учет переменных производится во время первичной трансляции инструкций байткода в выражения языка LuaJIT посредством заполнения таблицы использования переменных функции, назовем ее *useMap*. Так как каждая инструкция может иметь возможность как записи так и чтения какой-то переменной, в *useMap* учитывается и то и другое.

2.5.1 Удаление неиспользуемых переменных

Переменная V считается неиспользуемой в том случае, если она инициализируется инструкцией байткода в процессе работы функции, но дальше ни одна другая инструкция не производит операций чтения из этой переменной, а также переменная не замыкается ни одним из дочерних прототипов функции - такая переменная может быть удалена.

Переменная V - неиспользуемая и может быть безопасно удалена тогда и только тогда, когда выполняется условие - $useMap[V][read] == \varnothing$.

2.5.2 Подставнока переменной

В том случае? если переменная не замыкается ни одной из дочерних функций рассматриваемой функции и ее значение читается лишь единожды, значение переменной напрямую подставляется вместо самой переменной, а переменная удаляется.

Значение переменной V может быть подставленно, а переменная удалена тогда и только тогда, когда выполняется условие - $|useMap[V].read| == 1$.

3 Программная реализация

В качестве языка программирования для реализации прототипа был выбран язык *python*, так как он удовлетворяет условиям простоты и скорости разработки.

LuaJIT является довольно активно развивающимся проектом, в следствии чего формат его компилируемых файлов, а также инструкции претерпевают постоянные изменения. Существующие решения для анализа и декомпиляции байткода LuaJIT используют самописные модули, поэтому поддержка актуальности самого инструмента сторонним разработчиком осложняется необходимостью изучения исходного кода проекта и его архитектуры. В данной работе было решено использовать для этапа анализа сторонний модуль, поддерживающий декларативное описание для более наглядного и понятного описания формата файла, а также для облегчения поддержки актуальности, благодаря унификации описания структуры.

Ввиду распространенности задачи анализа различных форматов файлов и данных, на текущий момент существует довольно много инструментов для описания и создания синтаксических анализаторов файлов, одним из которых является *Kaitai Struct* [8] - инструмент, использующий декларативный способ описания структуры файлов, с поддержкой условных операторов, динамических секций и рекурсивных типов данных, благодаря чему было решено, в качестве стороннего модуля синтаксического анализа, использовать именно его.

Листинг 4 — Пример описания структуры файла в *KaiTai Struct*

```
1  local_file_header:
2      seq:
3        - id: version
4          type: u2
5        - id: flags
6          type: u2
7        - id: compression_method
8          type: u2
9          enum: compression
10       - id: file_mod_time
11         type: u2
```

ЗАКЛЮЧЕНИЕ

В работе предложены архитектура декомпилятора байткода LuaJIT и способы, предотвращающие недостатки существующих декомпиляторов:

1. Архитектура декомпилятора построена в виде нескольких каскадов анализа файла, что позволяет добавить гибкости в предоставлении информации пользователю.

2. Код функции представляется ввиде графа, что позволяет увеличить покрытие возможных обрабатываемых вариантов конструкций передачи управления.

3. Предложены две мутации, используемые при работе с графом потока управления функции, для повышения возможного покрытия кода функции: метод схлопывания последовательностей узлов и метод дублирования терминирующего узла.

Были составлены и предложены пары трансляции инструкций в псевдокод на языке LuaJIT. На языке *python* реализован прототип декомпилятора с описанной архитектурой, реализация представлена в [9].

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ И ЛИТЕРАТУРЫ

1. The LuaJIT project. — URL: https://luajit.org/luajit.html/ — Дата доступа: 14 января 2021.

2. LuaJIT Raw-Bytecode Decompiler (LJD). — URL: https://github.com/DrNewbie/luajit-decompiler — Дата доступа: 14 января 2021.

3. LuaJIT Raw-Bytecode Decompiler (LJD). — URL: https://gitlab.com/znixian/luajit-decompiler — Дата доступа: 14 января 2021.

4. Binary Ninja. — URL: https://binary.ninja/ — Дата доступа: 14 января 2021.

5. LuaJIT bytecode dump format. — URL: http://wiki.luajit.org/Bytecode-2.0#luajit-2-0-bytecode-dump-format — Дата доступа: 14 января 2021.

6. *Cifuentes C.* Reverse Compilation Techniques / C. Cifuentes. — 1994. — С. 126–163.

7. *Khaled Yakdan Sebastian Eschweiler E. G.-P. M. S.* No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantics-Preserving Transformations. — Электрон. дан. — 2018. — URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/11_4_2.pdf — Дата доступа: 14 января 2021.

8. KaiTai Struct framework. — URL: https://kaitai.io/ — Дата доступа: 14 января 2021.

9. LuaJIT decompiler prototype. — URL: https://github.com/mostobriv/luajit_decomp — Дата доступа: 14 января 2021.

ПРИЛОЖЕНИЕ А

Таблицы трансляции инструкций

Размер инструкций - 32 бита

Размер операнда - 8/16 бит (A/B/C - 8 бит, D - 16 бит)

Размер опкода - 8 бит

Таблица 1 – Форматы инструкций байткода

C	B	A	OP
D		A	OP

Таблица 2 – Типы операндов

Тип операнда	**Описание**
none	Операнд не используется инструкцией
var	Индекс в массиве слотов переменных
uvvar	Индекс в массиве слотов замыкаемых переменных
lit	8/16-битное целочисленное беззнаковое значение
slit	8/16-битное целочисленное знаковое значение
pri	Примитивный тип (0 - *nil*, 1 - *false*, 2 - *true*)
str	Индекс в таблице констант с конца (значение по индексу - строка)
tab	Индекс в таблице констант с конца (значение по индексу - таблица)
num	Индекс в таблице констант (значение по индексу - число)
func	Индекс в таблице констант с конца (значение по индексу - функция)
cdata	Индекс в таблице констант с конца (значение по индексу - тип данных язка Си)
jump	Смещение до инструкции

Таблица 3 – Операции возврата

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
RET	*var*	*lit*	*return* var_A, ..., var_{A+D-2}
RET0	*var*	*lit*	*return*
RET1	*var*	*lit*	*return var*
CALLT	*var*	*lit*	*return* $var_A(var_{A+1}, ..., var_{A+D-1})$

Таблица 4 – Операции копирования и тестирования

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
ISTC	*var*	*var*	$var_A = var_D$; *if* var_D *then goto label end*
ISFC	*var*	*var*	$var_A = var_D$; *if not* var_D *then goto label end*
IST	-	*var*	*if* var_D *then goto label end*
ISF	-	*var*	*if not* var_D *then goto label end*

Таблица 5 – Операции сравнения

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
ISLT	*var*	*var*	*if* $var_A < var_D$ *then goto label end*
ISGE	*var*	*var*	*if* $var_A >= var_D$ *then goto label end*
ISLE	*var*	*var*	*if* $var_A <= var_D$ *then goto label end*
ISGT	*var*	*var*	*if* $var_A > var_D$ *then goto label end*
ISEQV	*var*	*var*	*if* $var_A == var_D$ *then goto label end*
ISNEV	*var*	*var*	*if* $var_A \sim= var_D$ *then goto label end*
ISEQS	*var*	*str*	*if* $var_A == str_D$ *then goto label end*
ISNES	*var*	*str*	*if* $var_A \sim= str_D$ *then goto label end*
ISEQN	*var*	*num*	*if* $var_A == num_D$ *then goto label end*
ISNEN	*var*	*num*	*if* $var_A \sim= num_D$ *then goto label end*
ISEQP	*var*	*pri*	*if* $var_A == pri_D$ *then goto label end*
ISNEP	*var*	*pri*	*if* $var_A \sim= pri_D$ *then goto label end*

Таблица 6 – Унарные операции

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
MOV	*var*	*var*	$var_A = var_D$
NOT	*var*	*var*	$var_A =$ *not* var_D
UNM	*var*	*var*	$var_A =$ -var_D
LEN	*var*	*var*	$var_A = \#var_D$

Таблица 7 – Бинарные операции

Инструкция	**A**	**B**	**C**	**Псевдокод LuaJIT**
ADDVN	*var*	*var*	*num*	$var_A = var_B + num_C$
SUBVN	*var*	*var*	*num*	$var_A = var_B - num_C$
MULVN	*var*	*var*	*num*	$var_A = var_B * num_C$
DIVVN	*var*	*var*	*num*	$var_A = var_B / num_C$
MODVN	*var*	*var*	*num*	$var_A = var_B \% num_C$
ADDNV	*var*	*var*	*num*	$var_A = num_C + var_B$
SUBNV	*var*	*var*	*num*	$var_A = num_C - var_B$
MULNV	*var*	*var*	*num*	$var_A = num_C * var_B$
DIVNV	*var*	*var*	*num*	$var_A = num_C / var_B$
MODNV	*var*	*var*	*num*	$var_A = num_C \% var_B$
ADDVV	*var*	*var*	*var*	$var_A = var_B + var_C$
SUBVV	*var*	*var*	*var*	$var_A = var_B - var_C$
MULVV	*var*	*var*	*var*	$var_A = var_B * var_C$
DIVVV	*var*	*var*	*var*	$var_A = var_B / var_C$
MODVV	*var*	*var*	*var*	$var_A = var_B \% var_C$
POW	*var*	*var*	*var*	$var_A = var_B \wedge var_C$
CAT	*var*	*var*	*var*	$var_A = var_B .. \sim .. var_C$

Таблица 8 – Табличные операции

Инструкция	**A**	**B**	**C/D**	**Псевдокод LuaJIT**
TNEW	*var*		*lit*	$var_A = \{\}$
TDUP	*var*		*tab*	$var_A = tab_D$
GGET	*var*		*str*	$var_A = glob_D$
GSET	*var*		*str*	$glob_D = var_A$
TGETV	*var*	*var*	*var*	$var_A = var_B[var_C]$
TGETS	*var*	*var*	*lit*	$var_A = var_B[str_C]$
TGETB	*var*	*var*	*var*	$var_A = var_B[C]$
TSETV	*var*	*var*	*str*	$var_B[var_C] = var_A$
TSETS	*var*	*var*	*str*	$var_B[str_C] = var_A$
TSETB	*var*	*var*	*lit*	$var_B[C] = var_A$

Таблица 9 – Операции с константами

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
KSTR	*var*	*str*	$var_A = str_D$
KCDATA	*var*	*cdata*	$var_A = const_D$
KSHORT	*var*	*slit*	$var_A = D$
KNUM	*var*	*num*	$var_A = num_D$
KPRI	*var*	*pri*	$var_A = pri_D$
KNIL	*var*	*var*	$var_A, ..., var_D = nil, ..., nil$

Таблица 10 – Операции для работы с циклами

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
ITERL	*var*	*jump*	$var_{A-1} = var_A;$ $if\ var_A\ \sim=\ nil\ then\ goto\ label_D\ end$
JMP	-	*jump*	$goto\ label_D$

Таблица 11 – Операции вызова функций

Инструкция	**A**	**B**	**C/D**	**Псевдокод LuaJIT**
CALL	*var*	*lit*	*lit*	$var_A, ..., var_{A+B-2} = var_A(var_{A+1}, ..., var_{A+C-1})$
CALLT	*var*		*lit*	$return\ var_A(var_{A+1},\ ...,\ var_{A+D-1})$
ITERC	*var*	*lit*	-	$var_A, var_{A+1}, var_{A+2} = var_{A-3}, var_{A-2}, var_{A-1}$ $var_A, ..., var_{A+B-2} = var_A(var_{A+1}, var_{A+2})$

Таблица 12 – Операции с замыканиями

Инструкция	**A**	**D**	**Псевдокод LuaJIT**
UGET	*var*	*uvvar*	$var_A = uvvar_D$
USETV	*uvvar*	*var*	$uvvar_A = var_D$
USETS	*uvvar*	*str*	$uvvar_A = str_D$
USETN	*uvvar*	*num*	$uvvar_A = num_D$
USETP	*uvvar*	*pri*	$uvvar_A = pri_D$
FNEW	*var*	*func*	$var_A = func_D$

АНТИПЛАГИАТ
ТВОРИТЕ СОБСТВЕННЫМ УМОМ

Отчет о проверке на заимствования №1

Автор: Bozetcki Andrew sobozzyy@gmail.com / ID: 8660606
Проверяющий: Bozetcki Andrew (sobozzyy@gmail.com / ID: 8660606)

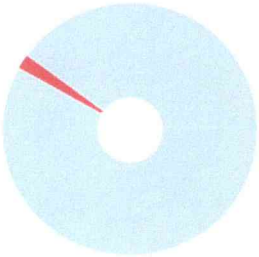
Отчет предоставлен сервисом «Антиплагиат»- http://users.antiplagiat.ru

ИНФОРМАЦИЯ О ДОКУМЕНТЕ

№ документа: 6
Начало загрузки: 26.01.2021 08:03:54
Длительность загрузки: 00:00:01
Имя исходного файла: DiplomaText.pdf
Название документа: StanchinDiplomaText
Размер текста: 32 кБ
Символов в тексте: 33185
Слов в тексте: 4237
Число предложений: 169

ИНФОРМАЦИЯ ОБ ОТЧЕТЕ

Последний готовый отчет (ред.)
Начало проверки: 26.01.2021 08:03:56
Длительность проверки: 00:00:02
Комментарии: не указано
Модули поиска: Модуль поиска Интернет



ЗАИМСТВОВАНИЯ	САМОЦИТИРОВАНИЯ	ЦИТИРОВАНИЯ	ОРИГИНАЛЬНОСТЬ
1,96%	0%	0%	98,04%

Заимствования — доля всех найденных текстовых пересечений, за исключением тех, которые система отнесла к цитированиям, по отношению к общему объему документа.
Самоцитирования — доля фрагментов текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника, автором или соавтором которого является автор проверяемого документа, по отношению к общему объему документа.
Цитирования — доля текстовых пересечений, которые не являются авторскими, но система посчитала их использование корректным, по отношению к общему объему документа. Сюда относятся оформленные по ГОСТу цитаты; общеупотребительные выражения; фрагменты текста, найденные в источниках из коллекций нормативно-правовой документации.
Текстовое пересечение — фрагмент текста проверяемого документа, совпадающий или почти совпадающий с фрагментом текста источника.
Источник — документ, проиндексированный в системе и содержащийся в модуле поиска, по которому проводится проверка.
Оригинальность — доля фрагментов текста проверяемого документа, не обнаруженных ни в одном источнике, по которым шла проверка, по отношению к общему объему документа.
Заимствования, самоцитирования, цитирования и оригинальность являются отдельными показателями и в сумме дают 100%, что соответствует всему тексту проверяемого документа.
Обращаем Ваше внимание, что система находит текстовые пересечения проверяемого документа с проиндексированными в системе текстовыми источниками. При этом система является вспомогательным инструментом, определение корректности и правомерности заимствований или цитирований, а также авторства текстовых фрагментов проверяемого документа остается в компетенции проверяющего.

№	Доля в отчете	Источник	Ссылка	Актуален на	Модуль поиска
[01]	0,32%	http://vital.lib.tsu.ru/vital/access/services/Download/vital:10648/SOURCE01	http://vital.lib.tsu.ru	24 Янв 2020	Модуль поиска Интернет
[02]	0,54%	http://vital.lib.tsu.ru/vital/access/services/Download/vital:10545/SOURCE01	http://vital.lib.tsu.ru	24 Янв 2020	Модуль поиска Интернет
[03]	0,48%	Шаблоны титульных листов ВКР	http://tsu.ru	10 Дек 2018	Модуль поиска Интернет

Еще источников: 11
Еще заимствований: 0,62%