Программа Ghidra – реверс-инжиниринг

Задача – восстановить большую часть логики игры

Самое важное – восстановить структуры данных. Структура данных – последовательность байт с типами. На физ. уровне типы сложно различимы – только разве их длина. Правда иногда на это намекает тип регистра(xmm - float), но все равно сложно. В любом случае главное – восстановить структуры, придав их полям осмысленные названия.

struct Player

{

int type,

float[3] pos,

float[16] matrix

}

Как восстановить структуру? Алгоритм:

1. Найти базовый адрес структуры, зная, что это действительно структура, а не массив.
2. Определить конец структуры – достаточно узнать размер. Размер можно получить:
   1. Через аллокатор, который выделил в куче память. Как правило он спрятан в конструкторе класса(этой структуры) или может быть снаружи.
   2. Если это стек – то под структуру резервируется память статически. Здесь сложно определить размер, ибо поля сливаются с локальными переменными на физ. уровне. Решение – сгруппировать поля по смыслу и выделить их в структуру. Можно глядеть в какие функции передается структура и с какими её полями(смещениями) она работает.
   3. Если не удается определить, то размер структуры – макс. смещение, которое для нее найдено.
3. Если определен размер и известен адрес, то для нас это пока что массив байт.

Теперь нужно определить поля структуры, пока что без типов. Важно получить список функций, которые работают с данной структурой. Если найдено что-то типа такого: [rcx+0x14], то сразу ясно начало поля структуры. Очевидно, что с 0x10 по 0x18 нет типа размера 8(например указателя) и что тут вполне может быть тип: int, float, bool

Однако обращение к полям происходит с указанием размера: byte, word, dword, qword. Поэтому легко определить и конец поля. Например: dword[rcx+0x14] – получение int или float значения(ну или bool). Необходимо обращать внимание на выравнивание. Очевидно, что указатель после этого поля(с адреса 0x14) начинаться не может.

1. Определив всевозможные поля, можно далее подразделить поля на int и float по регистрам. Также легко определить указатели. Сложнее различить int, byte и bool: нужно смотреть на значения.

Восстановленная структура – это определение всех полей и их типов, а также множество методов(функций), работающих с данными полями. Так как все современные приложения – это ООП, то все строится на структурах. Ибо структура описывает некоторый объект: игрока, машину, задачу, камеру, элемент интерфейса.

Содержание структур в играх – это состояние игры в данный момент времени. То, что мы видим на экране – это определенные значения в структурах. Поэтому методы могут просто изменять некоторые поля, при этом будет казаться в игре, что они меняют картинку, физику и т.д. В силу этого, можно сделать копию структуры и применять к ней методы, следя, какие поля она изменила и на что изменила при текущем наборе аргументов. Меняться глобальное состояние не должно в силу инкапсуляции.

Пример: у нас есть сущность, у которой есть кости(bones). Сделать возможность просмотра координат этих костей, визуализацию через directx, изменение положения и… сделать программирование через луа работу с этими значениями, возможно перезапись и тд

Стек, локальные переменные

|  |
| --- |
| 0x28: 5-й аргумент |
| 0x20: регистр |
| 0x18: регистр |
| 0x10: регистр |
| 0x08: регистр |
| 0x00: **адрес возврата (rax, rbp)** |
| -0x08: регистр |
| -0x10: … |
| -0x18: локальные переменные |
| -0x20: … |
| -0x28: регистр |
| -0x30: регистр |
| -0x38: регистр |
| -0x40: **адрес возврата** |
| -0x48: … |

Функция 1: sub rsp, 0x38 (место выделяется не только для лок. Переменных, но и для сохранения регистров в функциях, которые вызываются в данной функции + для 5-ого, … аргументов)

Адрес возврата хранится с 0x00 по 0x08.

Часто обращение к локальным переменным происходит через регистр rsp(а не rbp). Смещение относительно адреса возврата можно расчитать так: rbp = rsp + offset (rbp – указывает на адрес возврата, offset – значение в текущем месте), откуда rsp = rbp – offset. Заменяем это в текущей формуле:

[rsp + 0x18] -> [rbp + 0x18 - offset]

offset считается как сумма смещений, производимых функциями sub rsp, push, pop, …

Путь до выявления определенной функции

1) Синтез

2) Анализ

3) Концевая ф-я

Это 2 самых простых случая. Рассмотрим более сложные случаи - их комбинации.

1) Синтез & Анализ

Далее рассмотрим небольшой граф.

gvar

Задача: алгоритмически найти самый простой путь, чтобы разузнать, что данная функция делает. Простота складывается из минимального времени, нужного для реверсинга каждой вершины на этом пути. Дополнительные параметры – избегание вершин-методов некоторого класса или функций, у которых есть/нет определенные теги. Выбирать на пути преимущественно маленькие функции, желательно без виртуальных вызовов. Также учитывать глобальные переменные и прокладывать путь через мосты (синие линии).

На каждой вершине пути можно автоматически юзеру предлагать делать одну из вещей: debug, …

Хранить список пройденных вершин для ускорения.

Необходимо найти лучший из всех путей -> на каждом шаге пробовать все развилки и в итоге выбрать лучшую. Так как алгоритм поиска рекурсивный, то все действия одинаковы для всех вершин. Есть стартовая вершина(красная) и параметры поиска – это вход для алгоритма.

Алгоритм прост: сначала для стартовой вершины сравниваем структуру с одним из 3 базовых шаблонов. Если не совпало с каким-то, пытаемся рекурсивно вызвать данный алгоритм для тех вершин, из-за которых не совпало и т.д.

Находить можно все возможные пути, однако сортировать их в порядке простоты.

Концевые функции и задача синтеза

По сути, эти 2 задачи хоть и являются базовыми, но не являются тривиальными. Концевая функция может:

1. Вызывать API функцию (открыть файл, создать поток, …).

Тут всё просто.

1. Каким-то образом модифицировать входящие аргументы или возвращать какой-нибудь результат (работа со строкой, мат. функция, …).

Нужно просто дебажить или догадываться.

1. Изменять некоторую структуру, т.е. являться методом класса.

Если это метод неизвестного класса, и он изменяет неизвестные нам поля, то:

1. Изменять или читать неизвестную глобальную переменную.

Тогда идем по мосту глобальной переменной в те области программы, где легко выявить, что она обозначает.

Выбор функции автоматически, которую можно отреверсить

Где можно еще применить вышеописанный алгоритм, дак это для поиска следующей функции в программе, которую легко распознать.

Для этого проходимся циклом по всем функциям в программе, для каждой вызывая поиск пути. Заносим их в пул пройденных вершин, где для каждой пройденной функции легко вычислить сложность реверсинга, так как для каждой вершины вызывается поиск пути. Сначала можно вычислить сложности, отсортировать функции, а потом для каждой функции по отдельности вызывать поиск пути уже для пути. Можно также умножать на спец. коэффициент, который нужен для вывода в топ только тех функцией, чей реверсинг поможет на потом (число ссылок, глубина)

Создание структур

1. Найти аллокатор и деаллокатор памяти. Их может быть несколько.
2. Хукнуть их все, при каждом вызове заносить в таблицу адрес, время в нс, кол-во выделенной памяти и адрес возврата (map).

Легко найти: время жизни объекта, конструктор (адрес возврата), деструктор, размер структуры

1. Хукнуть все функции, куда может передаваться аргументом объект или возвращаться. При каждом вызове такой функции проверять наличие объекта в таблице. Для каждого объекта заносить функцию, которая для него вызывается (список ид функций), причем важен порядок. Первая вызванная функция – конструктор, последняя – вероятно деструктор. Также делать ***снимки*** объектов и заносить в буфер, а потом в файл!

Пример:

|  |  |  |  |
| --- | --- | --- | --- |
| Адрес в куче | Размер | **Адрес возврата** | Список функций |
| 0x1000 | 0x10 | 0x40000 | 102, 44, 105 |
| 0x2000 | 0x320 | 0x40010 | 45, 21 |

Адрес возврата – ключевое поле. По нему мы определяем уникальность объекта.

1. Как только вызывался конструктор, у которого тот же адрес в куче, то перенести инфу об объекте в другой массив.
2. После того, как таблица в течение долгого времени не возрастает в плане новых объектов, то мы можем получить что-то типа следующего:

|  |  |
| --- | --- |
| Размер | Список функций |
| 0x30 | **25**, 33, 100, 103, 670 |
| 0x34 | **25**, **100**, 103, 107 |
| 0x40 | **25**, **100**, 105 |
| 0x100 | 56, 89 |

Каждая строка – какой-то объект.

Мы видим, что у первых 3-х строк есть функция 25. Более, того – она вызвалась первой во всех случаях. Значит это конструктор для какого-то базового класса. 100 – так же конструктор базового класса. Базовый класс может быть абстрактным и не иметь конструктора. Тогда здесь не всё так просто, см. далее.

1. Давайте построим иерархию

0x30

0x34

0x40

0x100

107

33, 100, 103, 670

105

Надо также выяснить иерархию конструкторов. Для этого надо проверить, вызывается функция ид 25 из функции ид 33. Если да, то 33 конструктор. Эвристически можно определить, что 25 тоже конструктор, ибо за ним наверняка вызывалась функция аллокатора.

1. После того, как мы определили все классы в программе, надо присвоить им всем функциям. Даже если не все методы вызвались, они легко могут определиться из уже найденных в контексте. Имена функций переименовать в методы тоже или как-то иначе пометить. Можно вызвать специальный мастер, которым легко можно выбрать нужные функции, у которых надо изменить сигнатуру.
2. Далее надо определить оффсеты полей. Для этого придется пройтись по всем найденным методам. Нужно искать чтение и запись в оффсеты структур. Объект может быть получен через аргумент или через другую функцию. Так как мы знаем тип объекта(класс), то можем легко определить поля.

**1 пример**

7ff7896ac61e PUSH RDI

7ff7896ac61f SUB RSP,0x60

7ff7896ac623 MOV **RBX**,qword ptr [**RDX** + 0x10]

7ff7896ac627 MOV RSI,RCX

7ff7896ac62a CMP dword ptr [**RBX** + 0x40],0x0

7ff7896ac62e MOV R9D,dword ptr [**RBX** + 0x48]

7ff7896ac632 MOV R11D,dword ptr [**RBX** + 0x30]

7ff7896ac636 MOVSS XMM0,dword ptr [**RBX** + 0x38]

7ff7896ac63b MOVSS XMM1,dword ptr [**RBX** + 0x20]

7ff7896ac640 MOVSS XMM3,dword ptr [**RBX** + 0x18]

7ff7896ac645 MOV EDI,dword ptr [**RBX** + 0x28]

7ff7896ac648 MOV R8,qword ptr [**RBX** + 0x10]

7ff7896ac64c MOV RDX,qword ptr [**RBX** + 0x8]

7ff7896ac650 MOV ECX,dword ptr [**RBX**]

7ff7896ac652 SETNZ R10B

7ff7896ac656 CMP dword ptr [**RBX** + 0x50],0x0

7ff7896ac65a SETNZ AL

7ff7896ac65d MOV byte ptr [RSP + local\_18],AL

7ff7896ac661 MOV dword ptr [RSP + local\_20],R9D

7ff7896ac666 MOV byte ptr [RSP + local\_28],R10B

7ff7896ac66b MOVSS dword ptr [RSP + local\_30],XMM0

7ff7896ac671 MOV dword ptr [RSP + local\_38],R11D

**2 пример**

7ff789056469 PUSH RDI

7ff78905646a SUB RSP,0x20

7ff78905646e MOV **RDI**,**RCX**

7ff789056471 MOV RCX,qword ptr [**RCX** + 0x370]

7ff789056478 XOR R8D,R8D

7ff78905647b MOV EBX,EDX

7ff78905647d CALL FUN\_7ff789efc6a8

7ff789056482 TEST RAX,RAX

7ff789056485 JNZ LAB\_7ff7890564b0

7ff789056487 MOV RCX,qword ptr [**RDI** + 0x370]

7ff78905648e LEA R8D,[RAX + 0x1]

7ff789056492 MOV EDX,EBX

7ff789056494 CALL FUN\_7ff789efc6a8

**3 пример**

7ff7896bda72 SUB RSP,0x20

7ff7896bda76 CALL GetVehicleById

7ff7896bda7b MOV **RBX**,**RAX**

7ff7896bda7e TEST **RAX**,**RAX**

7ff7896bda81 JZ LAB\_7ff7896bdab7

7ff7896bda83 MOV RCX,qword ptr [**RAX** + 0x20]

7ff7896bda87 MOV ECX,dword ptr [RCX + 0x488]

7ff7896bda8d CALL FUN\_7ff7898d0f20

7ff7896bda92 TEST RAX,RAX

7ff7896bda95 JZ LAB\_7ff7896bdab7

7ff7896bda97 CMP qword ptr [**RBX** + offset DAT\_ffffffffa7da0b90],

7ff7896bda9f JZ LAB\_7ff7896bdab7

7ff7896bdaa1 MOV EDX,0x9

7ff7896bdaa6 MOV RCX,RAX

7ff7896bdaa9 CALL FUN\_7ff7898d0ea0

Алгоритм очень прост. Для этого надо эмулировать работу процессора. У нас есть определенный контекст, в котором находится текущие объекты. Далее мы проходимся по командам и сопоставляем каждому объекту в контексте определенный регистр, в котором он находится. Если есть команда CALL, которая возвращает объект, то заносим объект (его тип) в контекст и сопоставляем его с регистром RAX. Важно: если попалось условие, мы должны разделить проход на два – тут нужно юзать рекурсию. Таким образом очень просто определить типы полей структур.

Для float:

7ff78976360c **MOVSS** dword ptr [RDX + RCX\*0x1 + 0x40], XMM2

Далее надо опять вызвать мастер, в котором выбирать какие типы каким структурам присвоить, особенно в тех местах, где возник конфликт.

1. Зная оффсеты полей структур, их размеры, а также снимки объектов, можно потом точно определить типы.
2. Важно также определить типы полей-указателей! Для этого можно сохранять адреса объектов месте с их размерами, а также много снимков для каждого объекта. Потом на данном шаге сопоставлять поле-указатель по снимку с объектами из таблицы. В таблицу нужно будет заносить также список адресов!

Пример:

|  |  |  |
| --- | --- | --- |
| **Оффсет поля** | **Тип** | **Снимок** |
| 0x0 | Void\* | 0x7FF710120293 |
| 0x8 | int | 34, 78, 12, 33 |
| 0xc | float | 1.099, 2.33, … |
| 0x10 | Void\* | 0xba4455551 *-> class\_564* |

1. Распознать виртуальные таблицы

7ff789623a4b LEA **RAX**,[PTR\_LAB\_7ff78a3054a8] = *7ff7896247e0*

7ff789623a52 MOVSS XMM6,dword ptr [RSP + time]

7ff789623a58 XORPS XMM7,XMM7

7ff789623a5b COMISS XMM6,dword ptr [DAT\_7ff78a21d5a0]

7ff789623a62 MOV qword ptr [**R14**],**RAX**

7ff789623a65 MOV EAX,EBP

7ff789623a67 SETC AL

7ff789623a6a LEA RCX,[**R14** + 0xe0]

7ff789623a71 MOV RDX,RSI

7ff789623a74 MOVAPS xmmword ptr [**R14** + 0xb0],XMM7

7ff789623a7c MOV dword ptr [**R14** + 0xc0],EBP

7ff789623a83 MOV dword ptr [**R14** + 0xc4],EBX

Проходимся также по листингу, если видим присваивание по оффсету 0 некоторого статического адреса, то запоминаем это, т.е. заносим в таблицу. После анализа, когда мы прошлись по всем *конструкторам*, надо определить размер этих виртуальных таблиц.

Тут можно проходиться по всем функциям и делать поиск вирт. таблиц независимо, чтобы потом разграничить вирт. таблицы без использования гидры. Или можно юзать гидру и использовать разграничение по глобальным переменным. Лучше гидру, так как есть undefined функции. И вообще: пройтись по всем глобальным переменным. Если есть ссылка на функцию, то сделать из undefined обычную. Тогда можно хукнуть будет и их. И анализировать также.

Определить размер просто – идем вниз до тех пор, пока не встретим ссылку на другой конструктор. После этого создаем в сда объект виртуальной таблицы с функциями этими. Также у класса надо указать эту таблицу.

Потом можно в гидре самой сделать сигнатуры функций, тогда у нас виртуальные вызовы будут мало чем отличаться от невиртуальных -> больше определения типов.

Декомпиляция асма

Есть множество функций, которое представлено в виде списка функций. Все функции равноправны.

У каждой функции есть точка входа: список входных параметров. Есть выходное значение.

Есть так называемый **поток данных**: у каждого входного параметра есть свой путь преобразования, чтобы впоследствии либо стать входным параметром подфункции или для передачи преобразованного значения глобальной переменной. Пример:

FUN\_1(Entity\* entity, float speed)

Float speed2 = FUN\_2(speed, 2)

Entity->speed = speed2;

FUN\_3(entity)

Здесь входные параметры entity и speed. Entity – составной тип, агрегатор множества простых и составных типов. speed2 – результат функции.

***Нам надо определить все типы всех сущностей в программе так, чтобы мы занимались потом только переименованием и отладкой кода.***

***Надо определить то, что должна делать наша программа и то, что должна делать Ghidra. Впоследствии мы сделаем нашу программу независимой от гидры. Ghidra должна делать только декомпиляцию кода, все остальное – наша прога.***

Сделать поэтапное декомпилирование асма в c++:

1) проходимся по инструкциям асма и формируем первичное абстрактное дерево: есть вызовы функций, присваивания переменных другим переменным, выражения, а также условные выражения(блоки), циклы и операторы goto(прыжок).

1.1) Дизассемблируем и заносим структуры команд в список. Все будет хранится в памяти для скорости.

1.2) Определяем jmp range (диапазоны условий и прыжков). Таким образом заданный асм код разбивается на промежутки(функция – тоже промежуток, от начала до ret с влож. 0), которые выглядят как прямоугольники, которые могут пересекаться друг с другом.

1.3) есть конструкции:

a) if (одна стрелка вперед. Либо то, либо ничего)

b) if else (две стрелки вперед, вторая выходит из блока первой. Либо то, либо то)

c) while (одна стрелка вперед, другая назад. Есть общий блок и тело цикла)

d) do while (одна стрелка назад. Есть только тело цикла)

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

1

if (2) {

3

If (4) {

5

6

} else **{**

8

9

10

If (!11)

Goto 15

12

**}**

13

14

15

16

}

17

18

19

Проектирование

1) преобразование набора байт в список структур асм команд

2) построение на основе этого списка графа. Граф – это карта блоков асм кода, у которых есть связи. Из одного блока может выходить одна или две стрелки.

3) есть специальный контекст исполнения и интерпретатор команд (виртуальный процессор). Контекст – хранилище, интерпретатор – список действий, которые делают команды. На данном этапе мы получаем первичное абстрактное дерево.

Алгоритм:

1) rcx, rdx, r8, r9, xmm0, xmm1, …, stack – входные параметры, rax, xmm0 – выходной параметр.

2) строим первичное абстрактное дерево, никак не связанное с sda сущностями.

MOV R9,qword ptr [RCX + 0x40] => localVar1 = param1->field\_0x40

MOV R8, R9 => *localVar1: R8, R9*

TEST R9, R9 => if(localVar1)

JZ LAB\_111

Ставим соответствия между памятью (ram или регистры) и символами. Пример:

RCX: param1 (size = 0x8, longlong)

R14: localVar1 (size = 0x8, longlong)

R15: localVar1 (size = 0x8, longlong)

Stack[-0x10]: localVar2 (size = 0x4, float)

GlobalVar\_2: localVar3 (…)

Смотрим, как преобразуются эти значения. Пример:

*MOV R10, qword ptr [RDX] (сохраняем в контексте соответствие R10 <=>* localVar1*)*

*SHR R10,1*

*MOV RAX,qword ptr [R8]*

*SHR RAX,1*

*CMP RAX,R10*

*JA LAB\_7ff613e7b528*

localVar1 = param1->field\_0x0 >> 1

localVar3 = localVar2->field\_0x0 >> 1

if ( localVar2 < localVar1 )

А лучше, если нигде больше символы localVar1 и localVar2 не юзаются, сделать:

if ( param1->field\_0x0 / 2 < localVar2->field\_0x0 / 2)

**Важно!** Для большей многослойности упрощения делать уже над готовым первичным абстрактным деревом. На первом шаге просто трансформировать ASM код в дерево. Нужно, если захотим без лишних усилий добавить новую архитектуру процессора.

Также стоит отметить, что есть ветвящиеся штуки, такие как условные выражения и циклы, а есть просто некоторые преобразования и вызовы функций – это и составляет весь наш код. Соответственно у нас есть 2 задачи: задача построения дерева кода из условий и циклов и построения преобразований над символами (математические выражения, копирования, вызовы и т.д).

Сущности те же: условие, цикл – группа нодов, вызов функции – просто нода. Есть выражения – дерево преобразований над символами (переменные, числовые константы).

По сути, все эти шаги направлены на древовизацию линейного списка асм команд в удобную для дальнейшего анализа структуру, в которой легко производить навигацию. То есть все, что описано выше, предполагает только малую долю оптимизации (связанной с символами) и не является готовым абстрактным деревом для c++ кода. Здесь можно сделать нейронку, которая бы сопоставляла данной структуре некоторую высокоуровневый код путем PDB файлов определенного компилятора. Символы хранить в виде дерева как в гидре, сопоставляя каждому объекту свой символ.

Сделать диспетчер, потом несколько классов, которые будут работать с командами(категориями). По одной команде скармливаем интерпретатор, он строит абстрактное дерево, которое надо спроектировать (какие ноды, как что). Есть хелперы. Есть символы (они заменяют регистры, ячейки памяти и т.д). То есть надо разделять ответственность между классами.

Декомпилятор должен быть многоуровневым, чтобы некоторые уровни можно было кешировать. Например, первичное абстрактное дерево не связано с сущностями сда, и может быть сохранено.

Язык Си можно представить очень просто. Пока что исключим из рассмотрения условия и циклы (условный и безусловные переходы). По сути, есть у нас линейный список действий, производимых над символами(данными). Данные пассивны. Некоторые действия сгруппированы в функции, функция – это черный ящик, принимающий что-то на вход и дающий результат на выходе. Есть выражения, которые обязательно в конце присваиваются какому-то символу. Поэтому любую программу внутри операторных скобок можно представить:

Выражение = выражение

**или**

Вызов функции без результата *(предложение: не использовать отдельно вызов функции, а сделать null выражение = вызов функции. Так избавимся от ненужного полиморфизма)*

Например (высокий уровень):

\*((int\*) (&localVar1 + 0x128) + 0x4 \* localVar2) = localVar3 \* pow (localVar4, 2) \* 10.0 + 2.0 + \*(int\*) &localVar5

Надо сделать такое выражение более низкоуровневым. Работа должна быть с байтами.

**Операции**: 1) getValue 2) call 3) арифметические 4) логические

* Хранить для скорости в обратной польской записи => **не надо юзать new**

Нода = (присваивание) содержит ***адрес destination*** и ***значение***. Кол-во байт записи зависит от входящих в выражение символов (как известно, если мы знаем символы - мы знаем все)

* Символы локальных переменных и параметров функции (входных данных) хранятся в общем списке (в гидре, например, они все в начале). *Если они все известны => известна вся функция.*
* Для вызова функции параметры надо указывать, причем каждый параметр – это **выражение**, так же, как и адрес функции.
* Могут быть как ***символы***, так и ***константы***. Например, вызов функции pow – это вызов константы, а вызов виртуальной функции – это вызов выражения, вычисляющего адрес, причем есть обязательно база этого адреса – символ.

**Destination:**

localVar1

0x128

+

0x4

localVar2

\*

+

Для результата вызова функции pow:

localVar6 = pow

**Source:**

localVar3

*CallFunctionContext (передача управления определенному адресу)*

pow *(адрес – по сути тоже выражение)*

localvar4

2

\*

10

\*

2.0

+

localVar5

getValue *//не нужно здесь указывать тип, ибо он уже содержится в localVar5*

+

## Какова цель преобразования асма в такое дерево?

1) сокращается число мало зависимых друг от друга действий, производимых над памятью. Вместо списка команд, идущих в разном порядке, теперь список команд, который имеет какую-то определенную цель (эта цель едина – изменение данных).

2) ближе к тому, чтобы превратить все это дело в понятный человеку си код.

3) на основе этого дерева можно легко вычислять предполагаемые типы, производить другие виды анализа в поисках, например, противоречий. Таким образом, промежуточное представление выгодно в плане скорости, так как не надо производить много вычислений. Это представление заменяет ранее планируемый к разработке так называемый ExecutionContext, когда все это делалось прямо на асме.

Такое дерево можно хранить в памяти, то есть в БД. Не надо снова преобразовывать, можно хранить в кеше.

Важно при разработке структуры данного дерева не переборщить: удобное визуальное представление для человека – это одно, удобство для анализа – это другое. Тут не перейти грань главное.

Условные переходы в каждом блоке только 1 штука => по ним можно определить знак сравнения, перед ними всегда должно быть test или cmp, которые меняют состояния флагового регистра. Таким образом за некоторыми блоками может быть закреплено условное выражение.

**Сделать подписку** на интерпретатор обработчиков для поиска функций (а вернее, какие регистры юзаются для передачи параметров. Так мы решим проблему и с кастомными JMP)

**Можно** сделать эмулятор процессора, чтобы лучше потом дебажить. Сразу на асме выполнять. После каждой команды вызывать функцию сохранения контекста (состояния всех регистров) в стеке для последующего чтения. По значениям параметров функции, которые были сохранены во время работы триггера, можно дебажить код в оффлайне. Также можно сделать дамп памяти и дебажить с ним, правда надо решить проблему с RIP и потоками. Прикол здесь в том, что дамп памяти можно быстро из игры сделать и потом быстро загружать и т.д. в память. Для удобства дебага можно при каждом шаге делать снапшоты, но только тех областей, что изменены. Снапшоты делать триггерами, которые срабатывать будут по событию.

Символы могут быть временные (в контексте) и постоянные (в общем списке).

Символ – это поименованный адрес в памяти: функция, регистр, адрес в памяти для локальной и глобальной переменных, параметры функции (что и локальная, но с флагом каким-нибудь). Если увидим адрес на место за пределами данной зоны декомпиляции, то помечаем его сразу символом либо функции, либо глобальной переменной.

1) сделать контекст выполнения.

* Список временных символов (найденных)
* Список соответствия участкам памяти этих символов. Нашли новое место – создаем символ. Если это RCX – то это входной параметр. Если rax, то, наверное, выходной от последней вызванной функции, которая возвращает результат.

2) сделать интерпретатор, который будет изменять контекст. Сделать возможность клонирования контекста.

3) сделать структуру – группу нодов дерева. Туда интерпретатор будет складывать готовый результат декомпиляции. Также сделать списки готовых символов: глобальные (функции, gvar) и локальные (параметры, локальные переменные).

**Важно**: данное первичное дерево может меняться только если изменится асм, а **не**, например, сигнатура функции.

Вызовы и присваивания – это опорные точки, когда из контекста мы уже строим дерево (ноду).

rcx => expression (param1)

**rax** => expression (param1 >> 1)

ebx => expression (getValue(param1, 0x4))

ebx => expression (getValue(param1, 0x4) \* 2)

edx => expression (getValue(param1, 0x4) \* 2) *//выражение копируется*

call func (expression(param1), expression (getValue(param1, 0x4) \* 2)) *//ссылки на выражения*

**rax** = expression (param1 >> 1 + 1) **or** expression (func + 1)

**2 проблемы**: большие одинаковые выражения повторяются; вызов функции даст или не даст результат. Решение – временные символы (регистровые), отдельная нода для них.

# Решение первой проблемы

Сделать map (string, Expression), где string – адрес: rax, ram:0x103, stack:-0x2. Каждый map содержит объект Expression, который содержит список операций. Допустим у нас есть список функций, которые последовательно вызываются. Неизвестно какая из них какое значение возвращает.

Func\_1

Func\_2

Func\_3

Мы делаем для каждой из них временный символ. Получается отдельная нода для каждой функции.

retVal1 = Func\_1

retVal2 = Func\_2

retVal3 = Func\_3

Допустим изначально регистр rax содержал след. выражение: Expression (param1, 2, \*)

После он будет содержать: Expression (***retVal3***, ***retVal2***, ***retVal1***, param1, 2, \*)

Для xmm0 тоже самое!

Тут соблюдается приоритет. Ибо если все возвращают значение, то заюзано будет у последней вызванной функции. Если, например, возвращает только Func\_2, то при трансформации в высокоуровневое абстрактное дерево добавить в список невалидных символов символы retVal1 и retVal3.

У retVal в списке можно сделать флаг, что они прерывают цепочку, и список операций дальше относится только к определенному символу в этом списке.

# Решение второй проблемы

rbx => Expression (param3, 5, +, 2, \*)

getValue(param1, 0x8) + 0x10 = rbx

getValue(param1, 0x8) + 0x18 = rbx

**Будет:**

\*(int\*)(\*param1 + 0x10) = (param3 + 5) \* 2

\*(int\*)(\*param1 + 0x18) = (param3 + 5) \* 2

**А надо:**

localVar1 = (param3 + 5) \* 2 *//зависит от длины выражения*

\*(int\*)(\*param1 + 0x10) = localVar1

\*(int\*)(\*param1 + 0x18) = localVar1

Внимание! Эта проблема должна решаться для высокоуровневого абстрактного дерева, ибо это удобство не для анализа, а для визуализации. Однако…

Неплохо было бы знать, как строилось то или иное выражение. Обязательно ставить соответствия команды ассемблера тому или иному выражению! Например: оффсет команды ⬄ нода

Поэтому храним отдельно вектор выражений. В выражениях по ид делаем ссылку на подвыражение в этом списке выражений:

rcx => expression (param1) *//expr1, offset 0x10*

ebx => expression (expr1, getValue(0x4)) *//expr2, offset 0x14*

ebx => expression (expr2, 2, \*) *//expr3, offset 0x19*

Тогда получается уже граф. Плюс графа по сравнению со списком (обратная польская нотация) – это возможность потом быстро решить эту проблему. Для каждого выражения по ид можно вычислить его сложность. Можно создать новое выражение на базе повторяющегося, присвоить его переменной, а все вхождения заменить ид этой переменной (все это на высоком уровне делаться будет).

# Реализация

1) У нас есть операнды у команд. Сначала распарсим операнды. Примеры:

mov qword ptr [rdx], rax

mov byte ptr [rsp+0x10], bl

mov rax,rsp

sub rax,0x10

mov word ptr [rax+0x2], bx

mov dword ptr [0x1000000], eax

mov al, 0x1

mov ebx, dword ptr [rsp+0x10]

mov al, byte ptr [**edx** \* 0x8 + 0x100] *//в отдельный метод*

Если идет запись куда-то или изменение памяти, кроме регистра. Или вызов функции.

Reg:rax

Stack:0x8

0x200:0x100000

Регистры rsp, rip считать сразу

## Вариант 1

Во время присваивания делать касты через and.

mov al, 0x1

mov rax, 0x10235555 (rax > al => перезапись)

mov ah, 0x2

mov bl, al

mov cl, ah

mov byte ptr [0x1000], cl

mov dword ptr [0x1008], eax

al = 1

rax = 0x10235555

ah = 0x2 **and** rax = 0x10235555 & ~0xFF00 | (0x2 << 8) *//в зависимости от маски. Перезапись объемлющего регистра. Для ah это: rax, eax, ax*

bl = 0x10235555 & ~0xFF00 | (0x2 << 8) & 0xFF

cl = 0x2

## Вариант 2

mov rax, 0x10235555

mov rbx, 0x55

mov ecx, 0x100000

mov cl, al

mov dl, ch

sub rax, rbx

sub cl, bh

rax = 0x10235555

rbx = 0x55

mov byte ptr [0x1000], cl

cl = 0x10235555 & 0xff **and** ecx = 0x100000 & ~0Xff | (0x10235555 & 0xff << 0)

dl = undefined

rax = 0x10235555 – 0x55

cl = 0x10235555 & 0xff – (0x55 & 0xFF00 >> 8) **and** ecx = 0x100000 & ~0Xff | (0x10235555 & 0xff) & ~0Xff | (0x10235555 & 0xff – (0x55 & 0xFF00 >> 8) << 0)

## Вариант 3

mov rax, 0x10235555

mov dh, ah

mov cl, dh

mov byte ptr [0x1000], cl

rax = 0x10235555

dh = 0x10235555 & 0xFF00 >> 8

cl = 0x10235555 & 0xFF00 >> 8

## Вариант 4

mov rax, qword ptr [**edx** \* 0x4 + 0x100]

add ecx, dword ptr [**edx** \* 0x4 + 0x200]

mov edx, 0x1

inc edx

mov byte ptr [**edx** \* 0x8 + 0x100], ah

rax = read(edx \* 0x4 + 0x100, 0x8)

ecx = ecx + read(param2 \* 0x4 + 0x200, 0x4)

edx = 0x1 + 1

(0x1 + 1) \* 0x8 + 0x100 **=** read(edx \* 0x4 + 0x100, 0x8) & ~0xFF00 >> 8

## Вариант 5 (movsx, movzx)

mov rax, 0x10235555

mov rbx, 0x10000010

movsx eax, bh **or** mov ah, bh

mov qword ptr [0x1000], rax

**Важно**: регистры на аргументы заменять потом!

Вызов функции – отдельная нода с выражением адреса и списком выражений параметров.

Решить проблему с 5-м и так далее параметром. А также рассмотреть передачу вектора. Также сделать возможность указания кастомных регистров-параметров.

Внимание! Дизасм все за нас делает, мы только классифицируем операции и все.

**Сделать** потом компиляцию кода из полученного дерева снова в асм.

# Условия

Команды cmp и test могут идти вместе. Каждая из них изменяет только определенные части флагового регистра.

## Пример 1

mov rax, 0x1000

test rax, rax

jz 0x10

zf = 0x1000 & 0x1000 > 0

test => left = rax, right = 0 *//любые манипуляции с флагами стирают это из контекста*

## Пример 2

mov rax, 0x2000

mov rbx, 0x1111

cmp rax, rbx

ja 0x10

zf = (rax – rbx) == 0

cf = (rax – rbx) < 0

cmp => left = rax, right = rbx

ja => !((rax – rbx) == 0) && !((rax – rbx) < 0) *//если непосредственно*

ja => (rax **<=** rbx) *//уже намного лучше*

В любом случае лучше обработать эти команды сразу и сделать в конце для условных прыжков в блоке специальную ссылку на condition объект, который наследуется от ExprTree::Node.

if ((value == false) || (isInit != false))

## cmovnz – тернарный оператор

XOR EDX, EDX

MOV RCX, RDI

CALL FUN\_7ff612df28f8

CMP EAX, 0x1 *//всегда вычислять условное выражение*

CMOVNZ ESI, EAX

eax = fun(rdi, 0x0)

cmp => left = eax, right = 0x1

esi = (eax == 0x1) ? eax : esi *//тернарный оператор – это операция с 3-мя операндами: condition, node, node*

mov rax, 0x10235555

mov rbx, 0x10000010

cmp rax, rbx

**cmovz** rax,rbx

mov qword ptr [rsp + 0x30], rax

**Важно**: любое подвыражение (а именно OperationalNode) – это тоже может быть символ, который можно установить просто через флаг. Не надо создавать отдельно объект Line, его можно потом создать на пост-обработке или вообще не создавать.

# Стек и локальные переменные

MOV qword ptr [RSP + 0x18], RSI *//в объемлющей функции под это выделено памяти*

PUSH RDI

PUSH R15

SUB RSP, 0x30

MOV dword ptr [RSP + 0x38], EAX

MOV RBX, 0x2

MOV RAX, RSP

ADD RAX, RBX

MOV word ptr [RAX + 0x8], 0x10

\*(uint64\_t\*)(stackTop + 0x18) = rsi

\*(uint64\_t\*)(stackTop – 0x8) = rdi

\*(uint64\_t\*)(stackTop – 0x10) = r15

\*(uint32\_t\*)(stackTop – 0x40 + 0x38) = rdi

\*(uint16\_t\*)((stackTop – 0x40) + 0x2 + 0x8) = 0x10

И так далее. Повторный проход, но уже по первичному дереву, позволит вычислить оффсеты для стека. Символы для локальных переменных можно будет сделать потом. Сейчас нет необходимости в них => карту можно сделать как reg ⬄ expression

Входные регистры (rsp, rcx, rdx, …) можно использовать как стартовые.

**То же касается** и \*(uint64\_t\*)(RIP + 0x18) и всяких просто констант \*(uint64\_t\*)(0x1000 + 0x18).

## Массивы

Массивы могут в памяти реализованы быть по-разному.

Int [2]: 1, 2 dword ptr [rax + rbx \* 4]

Int [2][3]: 1, 2, 3, 4, 5, 6 dword ptr [rax + rbx \* 12 + rdx \* 4] //rax[rbx][rdx]

Int [2][2][2]: 1, 2, 3, 4, 5, 6, 7, 8 dword ptr [rax + rbx \* 16 + rdx \* 8 + rcx \* 4]

Int [2][2]: addr1 => 1, 2, addr2 => 3, 4 dword ptr [*dword ptr [rax + rbx \* 8]* + rdx \*

array[rcx >> 1][rdx] = 2;

# Оптимизация дерева

((0x1 + 1) \* 0x8) + 0x100 **=** (read((edx \* 0x4) + 0x100, 0x8) & ~0xFF00) >> 8

Выражение должно обладать минимальной избыточностью и должно позволять выдавать тот же результат, который был бы при вычислении на асме – это некоторое битовое число.

Решить также задачу уменьшения выражения (а лучше – удобного чтения). Для этого надо сосчитать длину.

## Проблема каста в выражениях

mov rax, 0x10235555

mov rbx, 0x10000010

movsx eax, bh *//если* ***bh*** *– отрицательный, то вторая часть* ***eax*** *заполнится* ***1***

mov [0x0000000000001000], rax *//что если вместо* ***rax*** *будет* ***eax/ax****; вместо* ***mov*** *-* ***movsx***

\*(uint\_64t\*)0x1000 = ((0x10235555 & 0xffffffff00000000) | ((0x10000010 & 0xff00) >> 0x8))

**Важно**: здесь | означает прибавку только определенной части регистра набора бит левого операнда, однако, если предположить, что левый операнд будет отрицательным, непонятно сколько старших бит надо заполнить единицами. Каждому операнду инструкции надо указать размер в виде числа байт, чтобы потом использовать соответствующие касты при вычислении выражений. Или просто указать truncate как 0Xffffff… и все. Тогда код будет унифицированным. Хранить число байт все таки нужно!

Что касается умножения операндов:

int a = -5;

uint64\_t b = 1;

b = b \* a;

то здесь, если **a** – это unsigned тип, то результат будет большое число. Иначе -5, так как до самого старшего бита будут 1, что сигнализирует об отрицательном числе.

Крч, эти касты нужны, где нам явно нужно прибавить только определенную часть регистра, то есть явно обрезать регистр. Причем отсечение связано с любой арифметической операцией.

Там где в асме есть разделение на знаковые и беззнаковые операции, там и ставим для выражения signed = true;

# Битовые представления чисел

Мы должны представлять числа как набор бит при разработке декомпилятора.

Числа (n-битные) бывают:

1) Беззнаковые: от 0 до 2^n

2) Знаковые: от -2^(n-1) до 2^(n-1)-1 (старший бит - знаковый)

Знаковые представлены в дополнительном коде:

Положительная часть одинакова всегда для всех видов кодов (прямой код).

Пример: -1 = 0b111 -2 = 0b110 2 = 0b010

Есть также разрядное знаковое расширение: 0b110 => 0b111110, 0b010 => 0b000010

# Операции

Здесь стоит обратить внимание на флаги переноса (CF) и переполнения (OF).

## Беззнаковые

Важен только флаг переноса (CF)

0b0111 + 0b1 => 0b**(0)**1000 *//переноса нет*

0b1111 + 0b1 => 0b**(1)**0000 *//перенос есть*

## Знаковые

Важен только флаг переполнения (OF). Вычисляется как **CF ^ Y**, где **Y** – перенос в старший рязряд

0b1111 + 0b0001 = -1 + 1 = 0b**(1)**0000 *//переполнения нет, хотя есть перенос;* ***1 ^ 1 = 0***

0b1111 + 0b1111 = -1 + (-1) = -2 = 0b**(1)**1110 *//переполнения нет, хотя есть перенос;* ***1 ^ 1 = 0***

0b0111 + 0b0001 = 7 + 1 = 0b**(0)**1000 *//есть переполнение, так как* ***0 ^ 1 = 1***

0b1000 + 0b1000 = -8 + (-8) = 0b(1)0000 *//есть переполнение, так как* ***1 ^ 0 = 1***

# Общая структура кода

К первичному дереву без символов можно потом в разном порядке применять или не применять специальные преобразующие классы: оптимизатор, создатель символов (можно получить список символов), …

# Общая задача

1) Сделать оптимизацию деревьев, вычислив все константные выражения, и приведя выражения к единому виду (2 выражения, дающих один результат, должны быть одинаковы). Можно делать во время построения дерева (так будет даже лучше), хотя бы только для вычисления констант. Любое вычисление начинать, если: запись в память, вызов функции

2) Все выражения с RIP сделать с оффсетом относительно начала функции.

3) Функциям надо передавать инфу о том, какой регистр является параметром, а какой выходным значением. Для этого предлагается, когда встретится функция, попробовать сразу вычислить дерево с адресом для этой функции. Можно сделать подписку на обработчика для функций и в самом обработчике вычислять оффсет функции (так лучше). Если вычислить не удалось глобальный оффсет, то можно искать по оффсету инструкции CALL где-то эту инфу о регистрах. Сама структура проста: массив регистров. Что важно: если передача через стек параметров, то это не надо указывать.

4) Сделать класс преобразователь для первичного дерева, задача которого присвоить символы: для локальных переменных, для глобальных переменных, для функций. Список этот можно получить. Тут возможно может пригодиться замена, а не только удаление ноды из дерева.

# Оптимизация деревьев выражений

Так как компилятор уже оптимизировал выражения, то нет особой необходимости пытаться разработать алгоритмы для оптимизации математических выражений. Также безуспешны попытки поиска универсального решения, способного привести любое выражение к единой форме. Здесь проблема заключается в том, что мы, будучи несведущими в том, какие выражения нам будут попадаться при реверсинге, пытаемся найти общий алгоритм, используя очень примитивные методики для его разработки, и обладая небогатым багажом знаний в этой области.

**Важно**: нельзя пытаться решить проблему сразу, делая большой общий алгоритм. Лучше не делать ничего, и по мере того, как мы будем находить частные случаи, на этих частных случаях и основываться. Накопив много частных случаев, мы сделаем алгоритм, который будет покрывать большой кол-во кода. Из практических соображений это намного выгоднее.

Оптимизация состоит из след. этапов:

1) Поиск нулевых выражений. Примеры: xor rcx, rcx

2) Вычисление константных выражений. Примеры: (5 + 2) \* 3

3) Приведение всех констант в выражениях к виду: слева что-то, справа – константа. Также замена минуса на плюс. Примеры: -5 + rcx = rcx – 5; edx – 2 = edx + (-2)

4) Вычисление повторяющихся операций. Примеры: (((rcx & 0xFF) + 5) +3) = ((rcx & 0xFF) + 8)

Этого будет достаточно для многих случаев. Но почему не стоит раскрывать скобки? Потому что компилятором и в исходном коде программы так задумано. Возможны случаи:

\*(word\*)(rcx + (rdx - 2) \* 0x2) => arr[rdx - 2] (of word)

Как анализировать эти выражения? Сначала ищем **базу** – адрес массива. Как правило, это регистр, на который ничего не умножается. Опционально ищем *смещение*, ибо база может указывать на структуру, в которой есть массив. Дальше ищем умножение. В этом умножении есть два множителя: размер элемента и индекс. Пример: ((**rcx** + ((rdx - 2) \* 0x2))) + *0x108*)

Важно заметить, что rcx может быть выражением. Однако тип “массив” мы можем присвоить только символу.

# Преобразование графа в условия и циклы

Все нижеприведенные фигуры являются фрактальными. Внутри них можно описать те же фигуры.

Блок if-else выглядит так:

Блок if выглядит так:

Множественный if-else:

**If()** {

} else {

**If()** {

} else {

**If()** {

} else {

}

}

}

If-else-if-else

If-else-if

Во всех случаях из одного блока может выходить либо **одна**, либо **две** стрелки, *либо не одной, но тогда это равносильно выходу из функции (инструкция ret)*.

## Исключительные ситуации и их интерпретация

A

B

C

D

E

A

If() {

B

If() {

E

return;

}

} else {

C

}

D

A

B

C

D

E

A

B

C

D

E

A

If() {

B

If() {

E

}

} else {

C

}

D

Выполнение кода можно сравнить с потоком воды по трубам. Соответственно, данный рисунок надо рассматривать как водопроводную систему. Мы имеем дело с ориентированным графом.

Первые эвристические предположения:

1) Если из блока выходят два пути, и потом они вместе смыкаются => это блок if

2) Если из блока выходят два пути в два независимых блока, а от них в один общий => это if-else

3) Если в блок входят пути, но из него не выходят => там есть инструкция ret => выход (return;)

4) Если блок if не смыкается, то использовать операторы goto

5) Любой код в блоке while, if, else может как выполниться, так и не выполниться. Есть родительский контекст read-only, и есть скопированный от родительского текущий контекст с регистрами. Если в данном блоке идет запись в регистр, который есть в родительском контексте, то мы создаем символ-переменную и делаем addLine в родительском блоке. В родительском и текущем контексте заменяем выражением в данном регистре на эту переменную.

На картинке выделен красный круг – это зона объемлющего if-else, у которого есть исток и сток. Есть if, который вложен в данный. От него стрелка выходит за пределы отмеченной зоны => нужно использовать goto.



Взглянем на след. рисунок



**If** () {

**If** () {

**goto**

} **else** {

**goto**

}

} **else** {

**If**() {

**goto back**

} **else** {

**goto**

}

}



Алгоритм:

1) Проходимся по графу, назначаем каждому блоку число как в волновом алгоритме. Это число – максимальный путь от начального блока до данного блока. Таким образом мы строим фигуру из блоков, которую очень легко анализировать (особенно визуально).

1 уровень

2

3

4

5

6

Начало петли

Конец петли

Начало петли

Конец петли

Начало петли

Конец петли

Петля – это главное в алгоритме приведения графа в дерево условий и циклов.

2) Далее проходимся по графу рекурсивно. Каждый блок, в который входит два и более путей, является своего рода ***гейтом***, через который можно пройти тогда, когда к нему будет подступ со всех входящих путей (резервуар с водой и клапаном). Также ищем петли. Следующий блок, в который надо идти, должен быть как можно ближе к данному (на рисунке красной стрелкой отмечен желательный путь), чтобы избежать проблем с поиском петли.

Алгоритм поиска петли:

1. Спускаемся от условного блока до первого ***гейта***

2. Поднимаемся снова вверх до условного блока

3. Заносим в стек текущий условный блок и спускаемся по другому пути

4. Как только попадаем в гейт, получаем с вершины стека последний условный блок – это и будет наша петля в итоге.

Петли могут быть вложенными друг в друга. Однако если петли пересекаются – это означает наличие оператора goto (может быть return;). На рисунке это видно (2 последние петли). Петля – это два числа: верхний и нижний уровень.

3) Каждый блок в графе может быть декомпилирован независимо друг от друга за одним НО: если есть чтение какого-то регистра, которого нет в текущем блоке, но зато он есть в других блоках наверху. Если регистр определяется неоднозначно, необходимо сделать специальный символ и работать в дальнейшем с ним.

1 уровень

2

3

4

5

6

чтение rax

запись rax

запись rax

запись rax

3/4

1/2

1/4

1/1

1/1

На рисунке сверху в последнем блоке мы должны прочитать регистр rax. Для этого надо его найти. Поиск делается так:

1) Сначала ищем регистр в текущем блоке. Если не нашли -> определяем тип блока:

a. Блок с одним входом -> идем наверх

b. Блок с ни одним входом(начальный) -> заканчиваем поиск

c. Блок с несколькими входами -> значит он является концом петли или петель. ~~Для этого надо найти систему из пересекающихся петель такую, чтобы она была как можно больше и захватывала самый высокий уровень~~ *~~(красными линиями отмечена такая система на рисунке)~~*~~. Далее собираем в мешок все блоки, где встречается этот регистр~~ *~~(на рисунке эти блоки отмечены желтым)~~*~~. После этого идем на самый верхний блок этой системы объединенных петель и продолжаем искать регистр, но уже не внутри какой-либо петли.~~

**Есть проблема**: искать пересекающиеся петли так нельзя, ибо можно в другой ветке найти, если финальных блоков несколько. **Решение**: использовать метод водопроводных труб. Для этого надо рекурсивно идти вверх, пока не встретим блок, в который вся идущая в начале вода вытекает. Применять метод гейтов. Таким образом используем принцип локальности. На рисунке отмечены дробные числа – кол-во воды.

2) Если мешок не пуст, значит мы имеем дело с неоднозначностью в присваивании значения данному регистру. Для этого надо создать символ и использовать затем его.

TODO: сделать провайдеры в виде разных компиляторов: MSVC, Intel, …

1

If

2b

*goto 4e*

else

2a

If

3b

If

*goto 4e*

else

4c

*goto 7*

else

3a

If

4a

if

*goto 8*

else

*goto 9*

else

4b

If

5a

*goto 7*

else

*goto 7*

*goto 8*

*goto 7*

4e

If

5b

If

*goto 6*

else

*goto 7*

else

5c

goto 6

6

7

8

9

If

else

13

1

2a

2b

3a

3b

4a

4b

5a

4c

5b

4e

5c

6

7

8

9

10a

10b

11b

12b

13

11a

12a

1

If

2b

goto 4e

2a

If

3b

If

4e

If

5c

Else

5b

If

goto 7

6

Else

4c

Else

3a

If

4b

If

5a

Else

4a

If

goto 8

Else

goto 9

7

8

9

If

else

13

9

If

10b

If

11b

goto 12b

12b

goto 13

else

10a

If

11a

goto 12a

12a

13

14

1

2b

2a

3b

4e

5c

1

2b

2a

3b

4e

5b

**Важно**: хранить итог декомпиляции как в виде графа, так и виде дерева условных выражений. Алгоритм преобразования из графа в дерево нужно применять к исходному недекомпилированному графу! Таким образом мы имеем: недекомпилированный граф, преобразованный недекомпилированный граф в дерево, декомпилированный граф. Хранить все в виде блоков и соответствий. Отсутствие украшательства кода.

**Декомпилятор** ставит соответствие между блоком асма и блоком первичного дерева.

## Алгоритм:

1. Начиная с начального блока идем до самого конца так: если встретилась петля, перепрыгиваем и делаем пустую конструкцию if-else, иначе делаем список блоков. Мы получим первичную структуру блоков 1-ого уровня. *1, 4e, 6, 7, 8, 9, 13 – использованные блоки*

1

**If**

**Else**

4e

**If**

**Else**

6

7

8

9

**If**

**Else**

13

2. Заполняем пустые конструкции if-else структурой 2-ого и т.д. уровня. Делаем так: если ссылка на блок, которого нет в списке использованных и номер которого наиболее близок к текущему блоку (желтая стрелка на рисунке), то переходим к шагу 1, делая ту же структуру. Иначе делаем прыжок на этот блок оператором goto.

Условный переход: делаем пустую конструкцию if-else

## Пример 1



1

If

2

If

3

4

5

If

6

goto 7

Else

goto 7

Else

goto 7

Else

goto 7

7

# Примеры циклов

## 1 пример

E2

**If**() {

D9

**If**() {

0A

} **else** {

DE

**goto** E2

}

} **else** {

E7

}

EA

Есть переход назад => это **цикл**. Условие в самом начале => это цикл типа **while**. Так как тут есть else блок, предлагается ввести цикл while-else. Первое условие продолжения цикла находится в начале, второе условие продолжения – внутри цикла. Во всех остальных случаях происходит выход из цикла => использовать оператор break внутри тела цикла. Телом цикла обозначить тело первого if.

*E2*

**while**() {

*D9*

***If****() {*

*0A*

*break;*

*}*

*DE*

***goto*** *E2*

} **else** {

E7

}

EA

## 2 пример

**Очень важно заметить**, что циклы и условия содержат участки кода, которые могут увеличивать значение регистра или как-то его изменять. Модификация регистров в таких местах должна выносить сами регистры в отдельные переменные (создаются отдельные символы для этого).

E2 *//это в основном блок, связанный с вычислением истинности услови*я

**If**() { F0 } *//можем быть в виде тернарного оператора (то, что выполняется в условии цикла)*

**If**() {

F1 *//начинается тело цикла*

**If**() {

D9

**If**() {

0A

} **else** {

DE

**goto** E2 *//можно заменить на* ***continue***

}

F2 *//перемещаем в место после* ***0A***

} else {

F3

}

F4 *//выход из цикла здесь =>* ***break****; можно как и* ***F2*** *переместить после* ***F2, F3***

} **else** {

E7 *//можно по сути игнорировать*

}

EA

Таким образом, можно заменить **goto** на **continue**, а можно сделать нормальный цикл: для этого надо переместить F2 и F4, инвертировать два if внутри цикла так, чтобы **goto** был самым последним в теле цикла. Поставить соответствующие **break** после **F2** и **F4**. Однако добавятся лишние break вместо одного continue.

# Новый алгоритм поиска регистра

Если встречается регистр в команде, надо его найти. Искать надо начиная с текущего блока и продолжая по цепочке вверх.

Суть поиска заключается в постепенном строительстве запрашиваемого регистра. Если мы, например, запросили регистр eax, то он может состоять из rax, ax, ah, al. Соответственно, ищем его части в каждом блоке последовательно идя вверх.

Не стоит забывать о флаговом регистре и об условиях.

Алгоритм:

На вход подается первый блок: ищем регистр eax, находим в первом блоке ah и al, умножение масок дает не нулевой результат => продолжаем поиск.

Переходим ко второму блоку: находим rax. Добавляем его к результату предыдущему, пересечение дает ноль => выражение построено.

Может встретится петля. Тогда надо еще добавить и символ. Надо также учитывать одинаковость выражений в разных участках петли. Если все выражения одинаковы => символ создавать не надо.

rax

**eax**

ax

ah

al

Сделать проверки со статической либой.

8

9

10a

10b

12b

13

11a

12a

11b

Запрос на **eax**

Тут **rax**

Тут **eax, ax, al**

Тут **ah**

Тут **al**

## Пример:

mov rax,rdx

mov eax, 0x12345

mov ah,ch

cmp eax, 0x10

jne lab1

mov ax, 0x1

jmp lab2

lab1:

mov eax, ecx

mov al, 3

lab2:

mov qword ptr [rsp+0x10], rax

## Хранение выражений регистров

Каждому регистру в контексте выполнения соответствует определенное выражение в словаре m\_registers. Если перезаписана какая-то определенная часть регистра, то эта часть хранится отдельно (например, al = 0x1). При получении, например, регистра eax формируется выражение путем склейки младших регистров. Именно поэтому для чтения регистра нужно кешировать отдельно, причем кеш только в пределах одного блока. То есть реализовано ленивое склеивание, по запросу.

## Еще пример:

mov rax,0x123456789A

jmp lab1

lab1:

mov eax,0x12345678

cmp ax, 0x12

jne lab2

add eax, 0x2

jmp lab3

lab2:

nop

lab3:

nop

jmp lab4

lab4:

add ax, 0x3

cmp al,0x1

jge lab5

mov rax, 0x10000023

jmp lab6

lab5:

nop

lab6:

add rax, 0x4

mov qword ptr [rsp+0x10], rax

## Проход по графу с учетом циклов

Рассмотрим блоки как резервуары с водой, а связи между блоками – как каналы. Изначально задан блок (входной), который заполнен 100% водой. Далее, как в алгоритме поиска в ширину в графе, мы по каналам от данного блока передаем в равном отношении воду соседним связанным блокам. Во избежание утечки воды вверх до конца, наша задача поднимать всю воду снизу вверх, поэтому будем делать итерации.

n-ая итерация заключается в том, чтобы найти самые нижние блоки на одном уровне, которые будут иметь конечное значение воды, передать воду соседним блокам и эти нижние блоки сразу обработать и занести в список обработанных блоков, при этом эти блоки продолжат участвовать в распределении воды. После этого, начать n+1 итерацию.

Суть в том, чтобы при обработке было известно, 100% или не 100% воды в нем, ведь от этого зависит способ обработки, а также надо пройтись только по нужным блокам.

Надо учитывать следующее:

*If() mov eax, 1 else mov eax, 2*

Проблема порядка в swap

1

2a

2b

3a

3b

4a

4b

5a

4c

5b

4e

5c

6

7

8

9

10a

10b

11b

**12b**

11a

12a

13

## Другой способ LinearView

Методом воды и труб определяем в какой блок на нижнем уровне стекается вода. Выбираем преимущественно самый верхний и с коэффициентом 0,95-1 (максимум воды стеклось). Придумать эвристическую формулу, позволяющую взять блок как можно выше, но с как можно большим коэффициентом. Создаем блок Condition, и в его двух ветках повторно делаем те же вычисления.

**Необходимо**: как можно более линейный список условий и блоков, как можно меньше операторов goto (это важнее всего), операторы goto должны вести вниз (обязательно).

Алгоритм похож на уже реализованный, только не надо вычислять петли. Если блок ссылается на блок, разница уровней с которым больше единицы, то сделать на такой блок ссылку и не добавлять в BlockList. В данном случае необходимым, но не достаточным условием существования оператора goto является переход с разницей больше 1

# Оптимизация LinearView дерева

Оптимизация должна производиться над графом после того, как будут декомпилированы блоки и будет известно, какой блок содержит код (работа с памятью), а какой нет.

Алгоритм:

1) Условие в каждом блоке должно соответствовать переходу на ближайший блок. Если это не так, то инвертировать условие и поменять местами ссылки на ближний и дальний блок. Сделать эту процедуру перед оптимизацией сразу для всех блоков.

2) После этого начать оптимизацию с первого блока. Проверяется ближний блок, который должен ссылаться на дальний блок текущего блока. Идет проверка на наличие в этом блоке кода (работа с памятью) – если его нет, то блок удаляется, условие объединяется в сложное И выражение. Дальше для текущего блока выполняется шаг 1), при этом при инвертировании сложного условия можно просто поставить флаг inversed или сделать рекурсивную инверсию знаков: && на ||, == на != и т.д.

1

2

3

4

5

b == 10 || b == 20 || b == 30

!= 10

!= 20

!= 30

Несмотря на условие ИЛИ мы смотрим это как на инверсию сложного И условия. После преобразования:

1

4

5

!= 10 && != 20 && != 30

1

4

5

== 10 || == 20 || == 30

→

Заметим, что ближний блок всегда находится справа.

С ссылками на блоки должен быть строгий порядок:

1. Условие перехода относится к ближнему блоку (отсутствие прыжка).

2. Ближний переход

Этот порядок должен сохраняться и после оптимизации графа.

## Создание while вместо условий

Во-первых, не нужно добавлять пустые блоки *(блок 46)*. Если на них кто-то ссылается, то ссылку на этот блок заменить ссылкой, которая идет после данного блока *(ссылка на блок 12)*.



Блок else в условии, заменяемом циклом, должен переместится вниз на уровень выше.



# Символы

Символ – это придание имени определенному участку памяти, которая может являться частью кадра стека или регистром. Каждый символ имеет адрес (абсолютный или относительный) и размер, указываемый в байтах. На более высоком уровне символ будет иметь ещё и тип.

Есть несколько типов символов:

## 1. Символы для регистров

Создаются как неразрешенные внешние символы в момент исполнения ассемблерных инструкций, когда в команде есть запрос на получение значения из регистра. Если никаких проблем с кодом и декомпилятором нет, то по-хорошему для каждого регистра должен встретится только один такой символ, остальные вхождения регистров заменяются выражением, содержащем этот символ, или другим типом символа (см. далее).

**Важно**: эти символы содержат сохраненные значения с того места программы, где они впервые появились.

Как правило, это входные регистры и их производные: rcx, rdx, r8, r9. Если есть эти же самые символы, но встреченные повторно в середине, значит **проблема** в неправильной сигнатуре вызываемой функции, её надо поменять.

## 2. Символы для неоднозначных регистров

Этот вид символа создается тогда, когда при запросе на значение из определенного регистра неясно, какое именно выражение оно содержит (это происходит из-за блока if-else или цикла). Поэтому создается символ в тех местах программы, в которой идет присваивание выражения запрошенному регистру.

## 3. Символы локальных переменных в стеке

Этими символами заменяются обращения к стековой памяти в виде смещения относительно сохраненного адреса возврата.

**Важно**: создавать символы только для тех смещений, из которых производится чтение, так как может быть только запись для аргументов вызываемых функций и для возвращаемого значения текущей функции. При этом важно смотреть на контекст вызываемой функции, ведь она может возвращать значение в стеке, тогда здесь будет и чтение (правда, это только для 32-битных приложений.)

## 4. Символы параметров

Могут обозначать входные регистры или стековые входные переменные.

## 5. Символы глобальных переменных

Тоже самое, что и предыдущее, но здесь идет абсолютная адресация.

## 6. Символы функций

Это немного другой символ, он не имеет размера и типа. Он просто заменяет числовой адрес функции именем.

## ~~Создание символов локальных переменных~~

**Важно**: делать эти символы на этапе преобразования в С/С++.

Есть функция создания, есть карта соответствия смещениям и символу (чтобы один экземпляр объекта был). Для отрицательных смещений относительно адреса возврата надо создать входные параметры, для положительных, но в пределах кадра стека локальных переменных, надо создать символы локальных переменных. Есть еще область в стеке под параметры вызываемых функций.

Могут быть массивы: \*(uint32\_t\*)((rsp + 0x20) + (ecx\*0x8))

Одно и то же место в памяти может считываться с разным размером. Решение – добавление умножающей маски при чтении.

При записи значений параметров вызываемой функции в стек, впоследствии удалять присваивания и заносить в контекст вызова функции выражение.

# Сложные объекты: работа с ними и их представление в памяти

Работа с объектом подразумевает наличие адреса начала массива байт этого объекта. Сами байты могут быть расположены:

1) В стеке

2) В глобальной области

3) В куче

В коде на C/C++ мы можем передавать объекты из одной функции в другую, копировать их из одного участка в памяти в другой. Рассмотрим передачу аргументов. Пускай у нас имеется объект **a** класса **A**. Есть функция **fff**, куда мы передаем этот объект.

## Передача объекта по значению, возврат по значению



**Красная стрелка** – входной параметр (второй, rdx), **синяя стрелка** – выходной (первый, rcx).



**Фиолетовая стрелка** – процесс копирования через SIMD регистры. **Зелёная** – запись в стековый объект значений из функций rand. **Жёлтая** показывает, что объект, возвращаемый по значению, в данном случае возвращается как ссылка на объект, который находится в той области стека, которая была указана первым параметром.

**Выходной параметр** всегда есть у функции (он первый), где производится возврат объекта по значению: мы говорим функции, куда она должна передать результат, ибо ей будет попросту непонятно (так как результат – множество байт).

**При этом**, если мы поменяем входной параметр передачи объекта на ссылочный тип (&), то ничего не поменяется внутри функции **fff**: и там, и там объект передается по адресу, просто в **первом случае** (по значению) мы перед вызовом функции копируем объект в стековую область памяти вызывающей функции, которая отведена специально для вызываемой функции, и передаем адрес на эту область, а **во втором** случае мы передаем адрес непосредственно на сам объект.

# Адресация разных объектов

Адресацию любого объекта (числовой переменной, массива или объекта) можно представить в виде: [addr] = [base] + [offset] \* [scale] ([size] – число байт, которые идут после [addr])

[base] – адрес начала объекта или какое-то значение *(выражение)*

[offset] – смещение поля объекта или индекс массива *(выражение)*

[scale] – размер единицы массива *(всегда число)*

1) Переменная: [offset] == 0

2) Массив: [offset] != 0 and [scale] != 0

3) Объект: [offset] != 0 and [scale] == 1

{[base], [offset], [scale], **[read\_size]**, [type]} – универсальное представление, где:

[read\_size] – число прочитанных байт. 0 — это ссылка. [read\_size] = sizeof([type])

[type] – тип, который интерпретирует байты, идущие после данного адреса. Если [base] – символ с типом, который равен типу каста, то не делать каст.

[base] – если символ, то по факту это какое-то число. То есть не надо делать чтение символа, надо делать чтение по адресу, который обвернут этим символом.

Примеры:

1) {[rsp], 0x10, 0x1, **0x0**, obj1} -> &stack\_0x10

2) {{[rsp], 0x10, 0x1, **0x0**, obj1}, 0x8, 0x1, **0x0**, obj12} -> &stack\_0x10.field\_0x8

3) {[rsp], 0x10, 0x1, **0x18**, obj1} -> stack\_0x10

4) {{[var1], 0x0, 0x0, **0x0**, obj1}, 0x4, 0x1, **0x4**, int} -> var1->field\_0x4

5) {{[var1], 0x0, 0x0, **0x0**, obj2}, 0x4, 0x1, **0x4**, int} -> ((obj2\*) var1)->field\_0x4

6) {{[var1], 0x2, 0x8, **0x8**, int\*}, 0x3, 0x4, **0x4,** int} -> ([var1])[2][3]

7) {{{[rsp], 0x10,0x1, **0x0**, int[2][3]}, *[var1]*, 0xC, **0x0,** int[3]}, *[var2]*, 0x4, **0x4**, int} -> stack\_0x10[v1][v2]

&(&stack\_0x10)->field\_0x8 ⬄ &stack\_0x10.field\_0x8

## Указание символов уровня SDA символам уровня декомпилятора

1) Стековая область (как и в классе у нас есть линейный список “полей” с типом и названием)

2) Глобальная область (точь в точь как и стековая область, да и любая другая область памяти)

3) Сигнатура функций (как в гидре, где соответствие param/ret регистру или области стека)

4) **Указание символов смещениям инструкций**

Пользователь может определить, какое выражение сделать символом. По умолчанию символы для выражений создаются автоматически и могут при каждой декомпиляции меняться. Есть 2 типы:

**1.** Символы неоднозначных регистров – это когда одному символу соответствует множество выражений.

**2.** Символы повторяющихся выражений – это когда одному символу соответствует одно выражение, но которое в единственном экземпляре или повторяется во многих местах программы.

Мы как бы называем непонятные нам ноды некоторым именем и придаем им тип, правда размер типа фиксированный.

Есть искусственно созданные ноды *(символы неоднозначных регистров)*, а есть ноды, непосредственно связанные с ассемблерной инструкцией. Таким образом, вторые легко идентифицировать (всего лишь смещением), а первые идентифицировать можно по тем нодам, с которыми они связаны некоторым отношением и которые связаны с ассемблерной инструкцией. Можно использовать множественные идентификаторы: если один или несколько пропадет, останутся другие.

**Итог**: делаем список связей смещений (тип - число) инструкций в ассемблере, соответствующих выражениям, с символами уровня SDA. При этом связь может быть двух типов: **непосредственная** *(повторяющиеся выражения)*, и **косвенная** *(символы повторяющихся выражений)*. Что важно, одному смещению может соответствовать 2 связи разных типов, поэтому делаем соответствие каждому символу уровня SDA списка смещений. Таким образом, когда у нас есть выражение, мы можем получить его смещение, а если у нас есть символ неоднозначного регистра, то получить список выражений, из которых потом получить список смещений, а по этому списку смещений определить символ уровня SDA. Всё просто.

### Проблема амперсанда – получения адреса символа

Тут трабл в том, что для символов памяти проблем особо нет. Но проблемы есть с символами параметров функции, ибо передача идет через регистр.

[rsp] - 0x30 -> &stack\_0x30

[rip] + 0x2340 -> &gvar\_0x2340

[rcx] = 5 -> param1 = 5

(read:1)[rcx] = 6 -> \*param1 = 6 *(byte param1)*

***stack\_0x30***[param1][param2 + abs(stack\_0x10)].m\_field0x10[param3]->m\_field0x1020 – здесь важно то, что цепочка адресов непрерывна.

Важные заметки:

1. Базой такого выражения является символ.

2. Учитывать адрес вплоть до битов, прочитать о битовых полях структур.

3. Учитывать, что чтение происходит в отдельную переменную.

4. Представлять структуру УОАП как линейку.

5. [1] = \*, [n] = [n] при n > 1

## Алгоритм (новый, с учетом массивов)

Итоговый адрес, будь то смещение или многомерный массив, всегда является линейным выражением с n переменными. Более того, базой такого адреса всегда является символ, который имеет обязательно некоторый тип и название. Наша задача преобразовать любое линейное выражение, представляющее адрес, в структуру УОАП.

**Адрес**:

Y = Base + Offset + X1 \* S1 + X2 \* S2 + … + Xn \* Sn

*или с учетом вложенных классов друг в друга*

Y = Base + (Offset1 + Offset2 + … + Offsetn) + X1 \* S1 + X2 \* S2 + … + Xn \* Sn

Чтобы не спутать выражение адреса с любым другим выражением, нужно ориентироваться на слагаемое Base, которое является символом. Если мы встретили символ, необходимо понять его тип:

1. [rsp] или [rip] – некоторая область памяти

2. [rcx], [rdx], … или [var1], [memVar1], … – входной параметр или другой символ

Если это – область, то мы не можем определить символ SDA и соответственно тип, но мы сразу знаем базовый адрес и можем узнать оффсет, который даст возможность определить символ. Если это входной параметр или другой символ, мы можем сразу определить символ SDA, потом тип. Если тип – это указатель, то мы знаем Base.

Таким образом в 1 типе, мы отталкиваемся от Base + Offset1 (&stack\_0x30, &gvar100), а во 2 типе от Base(param1). В итоге мы пришли к одному и тому же: у нас есть Offset2 + … и, может быть, линейное выражение массива. У нас уже есть GoarSymbolBase и мы начинаем строить GoarNode.

Мы построили выражение первого уровня, когда шли вверх по дереву, и если узел-родитель – это чтение по адресу, необходимо данному GoarNode присвоить readSize, тогда получится итоговое выражение без амперсанда &. Но если прочитанное значение – это адрес, то необходимо продолжить строительство.

Y1 = Read(Base + Offset1 + X11 \* S11 + X12 \* S12 + … + Xmn \* Smn)

Y2 = Read(Y1 **->** **GoarNode** + Offset2 + X21 \* S21 + X22 \* S22 + … + Xmn \* Smn)

…

Y1 теперь ссылается во второй строчке на GoarNode.

Нужно понять, что делать, если не указан символ SDA пользователем -> использовать лист не с символом SDA, а с типом, вычисленным автоматически, при этом SdaSymbol и AutoSdaSymbol обобщить общим классом родителем. Если мы ищем символ SDA в областях и среди параметров функции и не находим, то заменяем AutoSdaSymbol с автоматическим названием и типом: параметр, глобальная переменная, функция и т.д., то есть присваиваем тип из AbstractSymbol::Type.

Надо сделать вычисление наиболее подходящего типа SDA для символов и выражений в зависимости от контекста операций и других символов, тип которых определен юзером. Основная задача – минимум кастов выражений.

## Вычисление типов SDA. Алгоритм

Пусть у нас есть следующие строки:

[var\_1\_64] = ([rcx] + 0x10) \* ([rdx] – 0x2)

…

[var\_1\_64] = GetVehicleById(1);

1. Сначала идем снизу вверх для каждой строчки и вычисляем для каждого выражения тип в соответствии с правилами C++. Если мы доходим до выражения, тип которого жестко определен пользователем или автоматически, то дальше отталкиваемся от него. Тут появляется место каста.

2. Дойдя до вершины, смотрим, чем она является. Если символом, то добавляем символ в карту, где ему будет соответствовать список вычисленных типов. Впоследствии в порядке приоритета мы жестко определим типы всех этих символов. Если вершина – это запись по некоторому выражению, то делаем для него расчет как в пункте 1. Если [dstType] != [srcType], то делаем каст [srcType], но в пункте 3.

3. У нас определены типы для вершин, которыми являются левые ноды (dstExpr) присваиваний. Зная типы для вершин, мы теперь можем спускаться вниз, вычисляя типы для нижних выражений, избегая по возможности лишних кастов.

***Необходимо разработать следующие функции:***

1. Для операции с некоторыми выражениями X и Y, у которых вычислены типы, определить новый тип, руководствуясь правилами языка C++ (можно потом будет другие правила юзать, если надо).

2. Для ноды, для которой есть вычисленный тип заранее и текущий тип, определить необходимость каста и, если надо, сделать его для текущей ноды или для child нодов. При этом, если мы сделали каст для текущей ноды, надо все равно продолжить проход по child.

3. Сделать сравнение приоритетов двух типов. Например, uint < int или int32\_t < int64\_t. Можно сделать проверку эквивалентности типов: char = int8\_t.

Все выражения имеют родителя и в итоге куда-то присваиваются: dst = expr

Зачеркнутые шаги 1 и 2 надо объединить в один шаг, чтобы избежать излишних вычислений.

1. Сначала мы проходимся снизу вверх, чтобы построить УОАП и узнать тип для символов (которые используются в нескольких местах) слева. При этом создаем автосимволы только для входных параметров, [rip] и [rsp]. Символы [var] и [mem] не меняем.

2. Потом проходимся справа налево для присваиваний вида символ = символ, чтобы разузнать тип для правых символов (можно снизу вверх по строкам идти).

3. У нас построится карта символов: символ -> тип. Также можно построить карту функций: оффсет | номер параметра -> тип.

4. Вернуться к шагу 1. Если УОАП больше не создается, то перейти на шаг 5.

5. Пройтись сверху вниз, делая касты (ставя флаг). Если в конце дойдем до символа, который находится в карте, то меняем тип в соответствии с приоритетом и возвращаемся к шагу 1.

6. По карте символов сделать автосимволы для [var] и [mem].

**Пример**:

[mem1] = \*(uint64\_t\*)([rcx] + 0x10)

[var1] = (([mem1] + 0x100) + ([rdx] \* 0x4))

[funcResult1] = func1(*p1 =* 0x3, *p2 =* [mem1], *p3 =* [rcx], *p4 =* (([rsp] – 0x30) + ([var1] \* 0x4)))

[var2] = [funcResult1]

[var3] = [var2]

…

[var3] = 0x10

…

return *r =* [var1]

Примечания:

1. На первом шаге можно не создавать автосимволы, но тогда типы для символов стека и глобальной области будут разными в зависимости от контекста, ведь их не получится просто расположить в карте. Для параметров, [var] и [mem] проблем нет, для них можно автосимволы создать в конце, заменяя [rcx], … на param1, … с типом, который в карте. Для стековых и глобальных переменных. Чтобы избежать [rcx] + 0x10 и т.д., когда тип является структурой, то нужно вернуться к шагу 1, когда [rcx] – указатель на стркутуру. Возвращаться в конце, потому что указатель на указатель никак не повлияет, если несколько замен произойдет одновременно.

**Важно:**

*Возврат к шагу 1 нужен только для создания УОАП или изменения приоритета символов*. ***Основная задача*** *– определить самые приоритетные типы для всех символов*. *Очень похоже алгоритм на поиск частей регистра. Здесь есть начальное состояние – все символы с наименьшим приоритетом символа. После нескольких проходов состояние меняется в большую сторону.*

*Шаг 1-3 служат для обновления карты символов с учетом текущей карты символов, то есть шаги 1-3 увеличивают приоритезацию, таким образом state1 -> state2, где Prior(State2) > Prior(State1). Шаги 3,4,5 можно объединить. Таким образом шаги 1-3 будут крутиться в цикле, пока все не придет к высокоприоритетному состоянию. В этот или рядом надо вклинить создание автосимволов. [rip] и [rsp] с оффсетами должны быть тоже в карте, можно сделать 2 карты для rip и rsp. Для них лучше создавать массивы в конце.*

*Нужна оптимизация: (([rsp] + ([rcx] \* 4)) + [rdx] \* 8) + 0x30 -> ([rsp] + 0x30) + (…)*

*вместо текущей оптимизации для 2-х вершин: ([rcx] + 2x) + 3x.*

*Важно понимать, что этот алгоритм потом пригодится для создания реальных SDA символов, когда сначала исследуются концевые функции (у которых нет вызовов других функций), определяются типы параметров и возвращаемого значения, и т.д.*

**Ещё**:

Надо учесть битовые поля, также сделать дефолтное поле, ибо может быть ссылка на несуществующее поле.

**TODO**:

~~1. Нужно построить либо SdaSymbol, либо Goar. Первый случай – когда есть сверху чтение и если нет оффсета (кроме [rip] и [rsp])~~

~~2. GoarBase – либо символ Sda, либо символ декомпилятора через SdaNode. Учесть [rsp], [rip].~~

~~3. Для любого символа декомпилятора можно получить тип из карты или создать для него символ SDA. sdaSymbol = getSdaSymbol(symbol, offset = 0x0)~~

4. Разыменовывание. Если [rcx] получила в карте тип float\*, то [rcx] + 0x4 – это Goar.

5. Унаследовать от SymbolLeaf с интерфейсом ISdaNode для удобной замены в конце символами.

6. Карта: символ -> SdaSymbolLeaf (который либо пользовательский, либо авто), то есть автосимволы создаются не в конце, а сразу. Так проще сразу определить его тип: параметр, …

7. Сделать сразу класс автосимвола, у которого можно будет получить тип символа SDA.

8. Для контекстов вызовов функции сделать список регистров, которые функция требует.

9. Сделать саму функцию-вершину уровня SDA, с символом, параметрами, возвр. значением и т.д. Автосимволы делать для параметров и этой функции. Сделать автоопределение регистров-параметров. Для этого нужны режимы декомпиляции.

**Можно**:

1. Создавать все символы SDA (как авто, так пользовательские) в конце. В начале делать в карте инициализацию типов по символам SDA наравне с обычными автосимволами.

ИЛИ

2. Унаследовать от AbstractSymbol, сделав автосимвол; ~~можно, но не нужно убрать SdaSymbolLeaf, вместо этого сделать SdaSymbol~~;

Итог: [rcx] -> param1, [rsp] – 0x30 -> &stack0x30, [rsp] – 0x50 -> &stack0x40 + 0x10

Идея в том, чтобы для нас param1 и &stack0x30 были одинаковыми – это указатели по типу. ~~Можно сделать поле второго символа с дополнительным уровнем указателя, чтобы определить легко амперсанд, а можно через флаг.~~ Убрать GoarSymbolBase. Напечатать амперсанд легко через использование printDebugGoar в иерархии SdaNode. Карта будет не нужна, кроме параметров. Учитывать в качестве top вершин параметры функции, можно заранее сделать функции. Сама функция может быть глубоко внутри выражения.

# Алгоритм трансляции первичного дерева в C/C++ вкратце

Во-первых, создаем структуры данных, представляющих Си код. Здесь нет раздельного представления условных выражений и числовых выражений, всё в одной куче.

## Преобразование выражения в универсальное представление

Основная проблема – вычисление, какому объекту в системе вложенных объектов принадлежит абсолютный адрес: [rsp] – 0x34 => &stack\_0x10.field\_0x18.field\_0x4

Алгоритм должен работать за один проход: он проходится по первичному дереву, и встречая выражение *([rsp] – 0x10)* или символ *([var\_1\_32])* должен строить универсальное объектно-адресное представление (УОАП), описанное выше. Например, пусть *[var\_1\_64]* имеет тип obj1\* с полем field\_0x8, имеющем тип obj2\* с полем field\_0x10, имеющем int.

\*(uin32\_t)\*(\*(uin64\_t)\*(*[var\_1\_64]* + 0x8) + 0x10) -> {{{*[var\_1\_64]*, 0x0, 0x1, **0x0**, obj1}, 0x8, 0x1, **0x0** -> **0x8**, obj2\*}, 0x10, 0x1, **0x0** -> **0x4**, int}

#### Из первичного дерева в УОАП:

1. Видим *[var\_1\_64]* и знаем его тип – obj1\*.

2. Поднимаемся выше, видим прибавку смещения 0x8. Так как мы находимся в состоянии построения УОАП, то смотрим, что у нас означает последнее построенное УОАП: в данном случае – адрес начала объекта obj1. Значит прибавка 0x8 означает адресацию поля field\_0x8 объекта obj1. Значит берем предыдущее УОАП и оборачиваем его в новое УОАП с указанием адреса поля. Но чтение никакого не производим.

3. Поднимаемся выше, видим операцию чтения 8 байт по адресу, который является последним построенным УОАП. Изменяем в УОАП значение [read\_size] с 0x0 на 0x8.

4. Поднимаемся выше, видим прибавку числа, а так как последний УОАП – это адрес начала obj2, то ищем в классе obj2 поле со смещением 0x10 – это field\_0x10. Оборачиваем последний УОАП новым УОАП с указанием адреса этого поля.

5. Поднимаемся выше, видим чтение, изменяем в последнем УОАП значение поля [read\_size] с 0x0 на 0x4.

#### Из УОАП в C/C++:

1. Видим {*[var\_1\_64]*, 0x0, 0x1, **0x0**, obj1} – адрес начала объекта obj1, пишем var1

2. Поднимаемся выше, видим {…, 0x8, 0x1, **0x0** -> **0x8**, obj2\*} – чтение поля объекта var1, пишем var1->field\_0x8 . Операция ->, а не точка, потому что у нас чтение поля и подобъект - ссылка.

3. Поднимаемся выше, видим {…, 0x10, 0x1, **0x0** -> **0x4**, int} – чтение поля объекта obj2, пишем var1->field\_0x8->field\_0x10. Так как у нас чтение 4 байт, & не ставим.

**УОАП** может быть представлен как **вложенное выражение**:

((((var1)->field\_0x8)[2])->field\_0x10)

При этом, если нас интересует адрес, то & ставится в самом конце:

&(((var1)->field\_0x8)->field\_0x10)

В самой УОАП можно хранить доп. инфу (ссылку на поле объекта). Не забывать про дебаг!

## Вычисление типа выражения в C/C++

По умолчанию для констант, если это возможно: int32\_t, int64\_t, uint64\_t

В бинарной операции итоговый тип выражения унаследует тип того операнда, у которого:

1) Указатель, если другой операнд – число и, если операция сложения.

2) Больше размер

3) Беззнаковый тип, если размеры одинаковы

4) Размер 32 или 64 бита.

В противном случае (когда оба операнда 8 или 16 бит) тип выражения будет int.

## Проблема 1. Вычислять тип выражения самому или из маски

Маска гарантирует верный размер типа. Также наличие CastNode может говорить о знаковом типе, то есть по крайней мере использование знакового вместо действительного беззнакового не приведет к противоречию -> всегда юзать знаковый тип, если возможно. Противоречие может возникнуть только при расширении числа до большего размера или при операциях sdiv, srem.

CastNode можно преобразовать в SdaCastNode.

Есть:

1. Арифметика указателей

2. Целочисленная арифметика

3. Арифметика float чисел

### Итоговый тип выражения:

1. Если есть указатель (Car\*, int\*, float\*, long\*\*), то итоговый тип – этот указатель (8 байт)

2. Если есть пометка IsFloatingPoint, итоговый тип - либо float, либо double

3. Иначе создать беззнаковый тип по умолчанию размером, равной размеру маски

#### Мысли:

- использовать везде беззнаковые типы, и только в знаковых расширениях и операциях приводить операнды к знаковому типу, но результат потом снова к беззнаковому (концепция универсального хранилища uint64\_t)

### TODO:

1. SdaNode -> SdaCastNode & SdaNode

2.

## Дерево символов

Нужно для того, чтобы, изменив тип и имя символа в одном месте программы, он изменился и в другом и был там по умолчанию. Также может использоваться для поиска противоречий, например несоответствие знака, размера типа или самого типа (не объект, а массив).

Че должно быть сделано? Если мы изменим тип какого-нибудь одного параметра одной функции, то должен быть определен тип для целой цепочки выражений и символов, значение по которой, трансформируясь, идет от данного параметра. Можно показать окно, где есть противоречия и сделать что-то типа коммита.

# Символы уровня SDA

Это доменные объекты, которые характеризуют некоторый объект в памяти. Если все символы известны -> вся программа отреверсина. Символы прямо влияют надо код, показанный декомпилятором. Символы часто можно получить из объектов, к которым они относятся. Символы стоит рассматривать как большое линейное множество, а потому есть Менеджер символов с загрузчиком этих символов.

### Бывают 4-х видов:

1. Глобальные переменные

2. Локальные переменные (подразделяются на стековые, параметры и другие)

3. Сами функции

Есть так называемые области, которые похоже на области полей класса. В них расположены символы по определенным смещениям. Есть стековая область и глобальная. Стековая область нужна для каждой функции, чтобы определить все локальные переменные. Глобальная область одна и нужна для идентификации глобальных переменных и функций (которые как массивы байт кода).

Символы связаны между собой через декомпилированный код на определенном уровне абстракции (без оптимизации для красивого отображения юзеру), который может хранится в памяти и как правило не изменяется (ибо символы декомпилятора должны иметь постоянный размер). При изменении одного символа должны меняться все зависимые символы. Некоторые символы привязываются к инструкции (к адресу в памяти кода).

Порядок создания:

~~1. Делаем доменные объекты символов~~

~~2. Делаем доменные объекты областей, в которых будут эти символы~~

~~3. Делаем мапперы для символов и областей, делаем загрузку, пишем юнит тесты~~

~~4. Битовые поля структуры и класса~~

5. В разделе символов декомпилятора делаем символ, который ссылается на символ SDA

6. Делаем в разделе преобразования декомпилированного графа функцию, которая принимает на вход глобальную область и область стека, и по ним должны в дереве появится символы-ссылки на символы SDA

# Оптимизация Human View Optimizer

В нашем конвейере эта оптимизация находится после построения дерева и его оптимизации, но до символизации этого дерева. Хотя это не должно быть именно так, должна быть возможность проводить данную оптимизацию и над символизированным деревом (больше возможностей, когда есть типы). Для этого нужны интерфейсы.

## Основные задачи

1. Удаление временных переменных, которые используют стек

2. Поиск стековых/глобальных параметров

3. Удаление отдельных строк с чтением и оптимизация их порядка

4. Объединение раздельных переменных памяти, избежав конкатенации

Все эти задачи объединяет работа с памятью. Память используется так же, как и регистры, например хранить результат операции. Основное отличие от регистров в том, что адресация регистров четко определена (номер регистра и маска), тогда как адресация памяти куда сложнее (выражение для определения ячейки, если это многомерный массив). Если бы можно было определить ячейку памяти, тогда их, как и регистры, можно было бы использовать, обобщив регистры как тоже некоторую ячейку памяти.

## Пример

\*{uint32\_t\*}([rsp] – 0x30) = 0x1

[memVar1] = \*{uint32\_t\*}([rsp] – 0x30)

[memVar2] = \*{uint32\_t\*}([rsp] – 0x34)

*[memVar5] = \*{uint64\_t\*}([rsp] – 0x30) //новое отображение*

[funcVar1] = func(0x50)

*//если тут вызов функции (стенка), то очищаем контекст*

*//если тут массив, то очищаем область (глобальная или стек), которую он занимает (отрезок [a, b] или полуинтервал [a, +inf))*

If() {

\*{uint16\_t\*}([rsp] – 0x30) = 0x0

[localVar1] = [memVar1]

[localVar2] = [memVar2]

…

[localVar3] = [localVar1] *//пересылается сюда [memVar1]*

[localVar4] = [localVar2]

[localVar5] = CONCAT([localVar2], [localVar1]) *//новое присваивание ~~(можно в конце блока)~~*

} else {

[memVar3] = \*{uint32\_t\*}([rsp] – 0x60)

[memVar4] = \*{uint32\_t\*}([rsp] – 0x64)

[localVar3] = [memVar3]

[localVar4] = [memVar4]

[localVar5] = CONCAT([memVar3], [memVar4]) *//новое присваивание*

}

*//если тут вызов функции (стенка), то блокируем выход поиска отсюда к высоким блокам*

[memVar10] = \*{uint32\_t\*}([rsp] – 0x30)

func(CONCAT([localVar4], [localVar3]), …, [memVar10]) *//CONCAT – соединяет два выражения (возможна вложенность)*

### Алгоритм прохода

1. Сначала вычисляем значения для [memVar] в промежуточных состояниях контекста памяти. Мы можем получить:

a) [memVar32] -> [memVar32] *(если встретилась стенка)*

b) [memVar32] -> \*{uint32\_t\*}([rsp] – 0x30) *(если все хорошо)*

c) [memVar32] -> ([memVar32] & 0Xffff0000) | 0x333 *(если вместо стенки модификация. Такое плохо для решения задачи о непрерывном присваивании объектов. Лучше исключить, и желательно флагом)*

d)

2.

Встретив CONCAT пытаемся соединить воедино (используем как и везде **принцип парности**). Видим localVar4 и localVar3. Если их можно объединить (последовательное присваивание), создаем новый символ и везде, где было присваивание, добавляем новое с CONCAT левых операндов заменяемых присваиваний. Повторяем этот итерационный алгоритм N раз. Сами объединенные пары запоминаем. ~~Нежелательно, если заменяемые символы (это 3 и 4 выше) не уберутся, поэтому проверяем этот момент: можно один раз пройтись и в конце определить необходимость замены.~~ Все это делаем в начале. В итоге, решим задачу 4.

Решаем задачи 1 и 2. Необходимо учитывать стенки. Чтобы можно было применить и к символизированному дереву (там больше инфы о массивах), можно юзать интерфейсы. Например: IAssignmentNode -> getSrcNode() -> ISymbolLeaf -> ISymbol (size, location)

Потом удаляем все ненужные строки.

Важное отличие от декомпиляции состоит в отсутствии необходимости создания символов неоднозначных регистров.

Чтобы было соответствие между символизированным и обычным графом, надо клонировать выражения (например для memVar), которые будут модифицироваться.

**TODO**: потом сделать хранение для каждой функции список affected global vars.

# XMM/YMM векторы, float/double значения, множественные операции

**xmm/ymm** регистры – это векторные регистры, в которых может хранится несколько float или double значений. Операции совершаются независимо для каждой части регистра. Также эти регистры используются для быстрого копирования содержимого одного участка памяти в другой.

## Пересылка байт

При оптимизации компилятором пересылка объекта производится с помощью xmm регистра (хотя есть возможность и через ymm, используя AVX).

**XMM** регистр – это 16 байтовый регистр (**0x10** байт). Он умеет копировать и вставлять байты по адресам, выровненным по 16-байтовой границе: 0x10, 0x20, … Размеры объектов округляются до 8 или до 16 байт. Внизу показана часть функции UI\_DisplayDataSlot



Стоит обратить внимание, что объекты копируются чаще всего целиком. Наша задача – суметь определить все части объекта, чтобы на высоком уровне сделать одной строчкой: stack10 = \*obj1

В общем случае объект копируется так:

1) Берем n-ю часть объекта, заносим байты этой части в регистр EX/RX/XMM/YMM. Таким образом мы ставим соответствие: **EX** = read <4 bytes> (addr), **XMM** = read <16 bytes> (addr)

2) Теперь перемещаем содержимое регистров в другую область памяти: read <4 bytes> (addr) = **EX**

Таким образом в первичном дереве мы будем иметь следующие строки:

### Пример 1

2 объекта, у которых поле инкрементируется на 1.

\*(uint128\_t\*) ([rsp] - 0x30) = \*(uint128\_t\*) ([rsp] - 0x50)

\*(uint64\_t\*) ([rsp] - 0x28) = \*(uint64\_t\*) ([rsp] - 0x48) + 1

\*(uint64\_t\*) ([rsp] - 0x18) = \*(uint64\_t\*) ([rsp] - 0x38) + 1

\*(uint128\_t\*) ([rsp] - 0x20) = \*(uint128\_t\*) ([rsp] - 0x40)

### Пример 2

Если копируется несколько объектов, их части могут быть перемешаны (см. рисунок выше).

\*(uint128\_t\*) ([rsp] - 0x50) = \*(uint128\_t\*) ([rsp] - 0x100)

\*(uint128\_t\*) ([rsp] - 0x30) = \*(uint128\_t\*) ([rsp] - 0x80)

\*(uint128\_t\*) ([rsp] - 0x40) = \*(uint128\_t\*) ([rsp] - 0x90)

\*(uint128\_t\*) ([rsp] - 0x20) = \*(uint128\_t\*) ([rsp] - 0x70)

**Решение**: Так как объекты располагаются в памяти непрерывно, можно оптимизировать порядок пересылки байт.

Две строки записи в память можно поменять местами (условие сортировки), если предыдущая строка не влияет на следующую строку, т.е. они независимы.

Зависимость строк:

1) Первая строка меняет область памяти X, а вторая строка меняет область памяти Y, и эти области пересекаются X \* Y != 0, но не равны X != Y.

2) Первая строка меняет область памяти X, а вторая строка производит чтение из области памяти Y и запись в некоторую область памяти Z, и области X и Y пересекаются X \* Y != 0 и могут быть равны X = Y.

Две зависимые строки образуют блок (группу).

**Проверять надо четкую независимость**: для этого вычитаем одно выражение из другого, если получается число, то оно является расстоянием между участками, а так как размеры участков – числа, легко вычислить пересечение участков памяти, а значит и зависимость.



На рисунке зелеными стрелками показаны строки, которые можно поменять.

Сначала избавляемся от символов; при трансляции, зная типы, пытаемся соединить строки присваивания одного объекта вместе. Необходимо помнить основную задачу: из нескольких строк мы должны сделать одну строку – непрерывную непосредственную пересылку n байт из одного места в другое.

# Алгоритм удаления символов и поиск непрерывного присваивания

**[var\_1]** = [0x100]:8

[var\_4] = [0x110]:8

**[var\_2]** = [0x200]:8

[var\_3] = [0x104 ~~+~~ **~~[var\_2]~~**]:8

[0x104]:4 = 0

[0x50]:8 = **[var\_1]** + [var\_3]

[0x40]:8 = [var\_2]

[0x44]:4 = [var\_3] + **[var\_1]**

Перед нами конструкция, состоящая из подвижных стенок – оранжевых прямоугольников. Есть несколько моментов:

1) Каждая такая стенка может быть во взаимосвязи с другой стенкой, тогда их очередность по отношению друг к другу не должна меняться.

2) В начальном состоянии каждая стенка – это либо чтение из памяти (запись в символ), либо запись (запись в память), причем может быть смесь

3) Если идет подряд 2 стенки, где в первой идет чтение, а в другой запись одного и того же символа, тогда происходит замена символа во второй стенке и смена порядка этих 2 стенок.

В общем случае задача такова: определить, есть ли путь от стенки A до стенки B, то есть можно ли их склеить вместе. Тут сразу работает следующее полезное свойство транзитивности: если есть путь от A до B и от A до C, то есть путь от B до C. Этим свойством мы воспользуемся, когда будем собирать вместе непрерывное присваивание объекта, собирая все нужные стенки вместе.

**Алгоритм таков**. Нам дана стенка A и стенка B. Определим наличие пути между ними. Толкаем стенку A до стенки B, проверяя, можно ли двигать. Движение – это swap между стенкой A и стенкой A + 1, при этом между A и A + 1 не должно быть взаимосвязи. Если появилась стенка, через которую дальше пройти нельзя, начинаем рекурсивно с данной конфликтной стенки, запоминая текущую в стеке рекурсии. Как только конфликтная стенка прошла сквозь стенку B, возвращаемся на предыдущую стенку и продолжаем двигать её, пока она не будет рядом.

Чем-то похоже на сортировку, где есть вспомогательная функция, сравнивающая два объекта. Так и здесь, у нас есть 2 объекта и нам надо определить, делать ли своп или нет.

# Оптимизация работы с памятью

TODO:

1. Сначала вычислить все взаимозависимости, потом работать с ними

2. Пользователь может указать, что, например, параметр-указатель не ссылается никогда на стек и нет пересечения областей памяти стека и кучи, тогда нет взаимозависимости между строками. Надо дать сообщить декомпилятору как можно больше информации о взаимозависимостях, чтобы получить более лучший код.

3. Решить следующее:

a) сохранение регистров не отображать – фиктивная операция на высоком уровне представления

**Решение**: убрать строчки с неизвестными регистрами (не входными)

б) убрать временные переменные в стеке

в) убрать нули в команде MOVSS, т.е. убрать лишние операции с перезаписью памяти

**Решение б и в**: можно решить с помощью контекста исполнения после всей оптимизации.

Итак. Есть запись в некоторую область памяти некоторых значений. Наша задача – убрать лишние строки. Надо сделать шаблонный класс работы с памятью, где объекты могут накладывать друг на друга.

4. Убрать лишние присваивания такого рода:



**Важно**: перенести эту оптимизацию на более высокий уровень представления, ибо мы будем иметь информацию о типах, то есть о всей программе. Это касается в основном оптимизации порядка пересылки байт, где надо знать, есть ли зависимость между строк. На этапе первичного дерева это определить с высокой точностью невозможно.

## Хранение объектов в регистрах RX/XMM/YMM и их обработка

В игре GTA5 часто используется векторная математика. Вектор представляется обычно и 4-х компонент типа float и обрабатывается он в регистрах XMM. Матрицы – это 4 вектора и, соответственно, 4 регистра XMM.

**Важно**: mm – тоже векторный регистр (MMX), причем к нему относится большое кол-во команд с приставкой P, которые могут работать с 16 и 8 битными частями.

### Пример 1. Взятие первых 3-х компонент X, Y, Z из XMM

0x55 = 0b010101**01**

0xAA = 0b101010**10**



Таким образом, в первой части регистров XMM6, XMM7, XMM8 оказывается компонента вектора X, Y, Z соответственно.



# RISC архитектура, микрокоманды Ghidra

Команд в x86 очень много, особенно учитывая расширения современных процессоров (SSE 4.2, AVX 2, …). Удобно команды программировать микрокомандами, особенно, если это касается векторных регистров. При этом набор микрокоманд лучше взять

<https://ghidra.re/courses/languages/html/pcodedescription.html>

### Реверс-инжиниринг виртуальной архитектуры набора команд Ghidra:

1. Команды работают с одним или несколькими регистрами, которые могут быть разного размера. Например, команда add eax, eax – работа с одним регистром 4 байт, а add xmm0, xmm0 – работа с 4 регистрами размером 4 байт.

2. Арифметические операции могут быть с INTEGER или с FLOAT, причем указывается размер.

3. A **varnode** is a generalization of either a register or a memory location.

4. All p-code operations are associated with the address of the original processor instruction they were translated from.

5. Копирование из памяти или в память регистра XMM надо делать через 4 составляющие, ибо они минимальны.

6. Юзать символику гидры (pCode, varnode, …), преобразование в pcode делать в самом начале до асм графа, каждому адресу ставить в соответствие список инструкций pCode

7. Сделать класс, в котором для каждой команды архитектуры программировать pCode. Чтобы кода было не так много, использовать шаблонные вставки (операнды и т.д.)

8. По сути при переходе к pCode ничего особо не поменяется, зато для векторов все станет по-другому.

9. В контексте должно быть соответствие между варнодом и выражением, причем варнод – это регистр или переменная. Вместо карты сделать список, и, передавая варнод, по его полям определить варнод в контексте.

По сути, команда дробится на микрокоманды, которые почти все (кроме RETURN, …) возвращают свой результат во временную переменную или регистр, т.е. varnode:

[varnode1] = cmd([varnode2], [varnode3], …)

[varnode] может быть символом (enum: sym\_a, sym\_b, …), числом, регистром (флаговым регистром).

Необходимо объявить список регистров и информацию о них. Есть так называемые родственные регистры – регистры, разделяющие одну область памяти:

1. RAX/EAX/AX/AH/AL – делят область памяти RAX

2. EAX и AX пересекаются, а AH и AL не пересекаются, но все они являются явными

3. XMM0\_Da и XMM0\_Db не пересекаются и не являются явными. Они используются в пределах микроопераций

4. CF/OF/SF/ZF – флаговые регистры, делящие единый флаговый регистр. Их маски (0b1, 0b10, …)

**У каждого регистра есть:**

1. Маска диапазона значений (EAX = 0b1111)

2. Маска покрытия (EAX = 0b11111111)

Отсюда можно вычислить маску дополнения нулями: 1 & ~2

Таким образом, лучшим вариантом является идентификация не по регистру, а по группе однородных регистров, но с масками.

**Структура регистра в операнде такова:**

1. ID области однородных регистров = ID регистра с макс. маской

2. ID явного child регистра (AX of RAX) или 0, если регистр неявный (XMM0\_Da of XMM0)

3. Маска диапазона значений

4. Маска покрытия

Эта структура является общей для любой архитектуры.

**Важно**: размеры указывать в битах, а не в байтах из-за флаговых регистров.

Таким образом, мы делаем трансляцию ZYDIS структур в наши структуры pCode и работаем только с ними. Каждому оффсету соответствует команда, которая содержит входные и выходной варноды, которые могут содержать регистры, причем структуры регистров можно выделять в куче. Дальнейшая работа с pCode подразумевает только его чтение. У нас получится список списков микрокоманд.

## Пример 1

Удаление старшей части регистра как отдельная микрооперация несмотря на наличие специальных масок



## Пример 2

Мы видим чтение из памяти, поэтому здесь лучше отдельно запрограммить память и регистр. В итоге у нас получается чтение 4-х байтовых переменных.



## Пример 3

Берем сдвигом из целого регистра его часть. Однако тут есть проблема.



При записи в память структуры с 2-мя полями типа int в асме происходит склейка в 64 бита:

[ram] = (100 << 32) | 250

Это должно развернуться в:

a.field\_0x0 = 250

b.field\_0x4 = 100

Зная тип выражения справа по выражению слева с помощью автовычисления типов, мы можем попробовать разбивку выражения на составляющие. Не забываем про присваивания символам:

[sym] = (100 << 32) | 250

То есть любое присваивание – сигнал проверить на упаковку полей структур таким образом и создание нескольких присваиваний вместо одного.

При чтении мы получаем значение одного поля, а значит имеем дело с одним сдвигом и операцией AND:

[ram] = ([sym] >> 32) + ([sym] & 0xFFFFFFFF)

Нужно помнить, что структура может быть в одном регистре (не только XMM, но и RX, EX, X).

На рисунке выше может быть след. ситуация с содержимым регистров:

XMM1\_Db = ((([rbx] + 6) << 86) | (([eax] + 4) << 64) | (([ecx] + 2) << 32) || (0 + 1)) >> 32 & 0xFFFFFFFF

Должно быть XMM1\_Db = [ecx] + 2. Чтобы такое получить, необходимо использовать **битовые поля** структур. Делаем модуль оптимизации, который будет проходится по выражению и ставить соответствие каждому битовому адресу выражение, а после сдвига и умножения получать это выражение.

**Важно**: сделать класс mask, где можно хранить маску в битах больше 64, используя при этом класс из C++ STD. Эту маску можно юзать только в выражениях после декомпиляции.

**Важно 2**: для оптимизации сделать выше показанный микрокод короче, используя только COPY.

### Алгоритм:

???

## Пример 4

Операция умножения выглядит как несколько операций над обычными регистрами



Сложение также



**Важно**: использование микрокоманд позволяет адресовать любое выражение (например выражения памяти) как: **offset** инструкции и **id** микрокоманды

## Пример итоговой трансляции 1

\*(uint64\_t\*)[rsp:0x108] = 2000

\*(uint64\_t\*)[rsp:0x130] = 5

\*(uint64\_t\*)[rsp:0x100] = ([ecx] + 1) | ([edx] << 32)

# Сравнение чисел

Возможны ситуации знакового переполнения, когда из большого отрицательного числа вычитается большое положительное число => необходимо использовать специальные функции, определяющие переполнение вычитания или перенос бита:

SBORROW([var\_5\_32], [var\_3\_32]) **==** (([var\_5\_32] + ([var\_3\_32] \* 0xffffffff{-1})) < 0x0{0})

Здесь сравниваются 2 числа, причем одно вычитается из другого и сравнение происходит с нулём. Результат вычитания может быть числом положительным, поэтому результат сравнения нужно будет инвертировать, для чего используется SBORROW, который определяет переполнение.

Можно рассмотреть 2 варианта:

char a = -125;

char b = 20;

char c = a - b;

if (c < 0) {

a = 0;

}

Здесь будет **переполнение**, но без SBORROW – как итог, условие не выполнится

char a = -125;

char b = 20;

if (a < b) {

a = 0;

}

Здесь тоже будет **переполнение**, но с SBORROW и условие выполнится

## Оптимизация условий

1. Сначала оптимизируем все выражения, не трогая структуру условий

2. Дальше оптимизируем SBORROW условия

3. Оптимизируем отрицательные слагаемые в условиях, перемещая их вправо

4. Делаем инверсию условий, где это нужно

5. Делаем единый порядок в составных условиях

6. Оптимизируем составные условия: < 2 or == 2 -> <= 2

# Транслятор в C/C++. Этапы создания

1. pcode

2. Сделать поддержку векторов, float/double значений.

3. Структуры данных, описывающие элементы C++. Здесь реализовать УОАП. Написать Formatter, то есть класс, отображающий код: в текстовом виде, в виде элементов GUI.

4. Сделать стековую и глобальную область, в которых будут символы уровня SDA.

# Декомпилятор. Новая архитектура

Алгоритм:

1. Есть блок, в котором запрос на получение значения из определенного регистра, чтобы было выражение, а не символ регистра. Допустим регистр EBX.

2. Просматриваются верхние блоки (текущий - нет, ибо он уже был просмотрен ранее). Если нет развилки, то:

2A. Собираем части регистра (EBX = RBX + BX + BL).

2B. Иначе, входим в состояние, при котором мы можем собрать части регистров в неполных блоках (их pressure != max), увеличивая маску **needReadMask** на **canReadMask**. Эти собранные части составляют неопределенный символ. Размер этого символа = needReadMask. *В неполных блоках идет присваивание этому символу*. После выхода из неполного блока, смотрим, чтобы маска **hasReadMask** покрывала needReadMask => тогда мы выйдем из состояния построения неопределенного символа как части запрошенного регистра. Если осталась ещё маска, то продолжаем как обычно.

Каждому неполному блоку присваивается номер, равный номеру запроса.

TODO:

- вынести класс разрешения внешних символов отдельно в GrahModification (как это с MemoryOptimization было сделано)

- короткие методы

- больше пояснительных переменных и комментариев для них

# Создание типов SDA в первичном дереве

Все оптимизации должны проводиться над первичным деревом, а из него потом получать только C/C++ код без дополнительных доработок. Поэтому надо создать элементы SDA и типизировать безликий код первичного дерева:

1. Создание элемента УОАП, который заменит все чтения из памяти специальной структурой УОАП

2. Создать единый символ-переменную, которая имеет имя и тип, и которая ссылается на предыдущий символ

3. Замена всех вызовов функций конкретной функцией из SDA

4. Не надо делать касты, ибо они раз не нужны были в обычном первичном дереве, то и в новом тоже не нужны, их заменяют обычные маски

**Алгоритм**:

1. Берем и заменяем все символы на единый символ-переменную с названием и типом, причем данные берем из входных данных стека, глобальной области, про которые написано выше. Здесь же делаем разбиение сложных строк на 2 и более, используя новые переменные, определенные юзером. Важно понимать, что символ является базой всех выражений и от него будет вычислять тип вверх по рекурсии.

2. Идя вверх по рекурсии от созданных символов-переменных вычисляем тип более высоких выражений и заменяем их именованными структурами УОАП с ссылкой на типы SDA.

# Создание циклов

Есть циклы while и do-while. Есть продолжение цикла **continue** (если он в конце, то он скрыт) и есть выход из цикла на следующий за циклом блок **break**.

Основная задача – сделать тело цикла как можно меньше, и чтобы в самом цикле было как можно меньше goto операторов (все они должны быть или break, или continue). Часто эта задача заключается в выборе цикла while или do-while. Иногда, чтобы получить цикл do-while или цикл со сложным условием, необходимо перед этим преобразовать декомпилированный граф.

# Создание графа функций по PE файлу

Дана точка входа в программу. Начиная от нее, необходимо решить 2 проблемы:

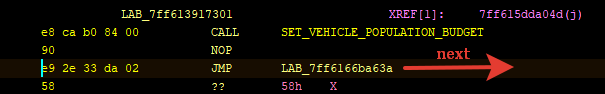
1. Покрыть всю программу (включая виртуальные функции, прерывистые функции, …)
2. Точно найти границы функции

Примеры:

1. GET\_NUM\_VEHICLE\_MODS – вызов функции через JMP. Используется, чтобы заменить две команды call, ret одной – jmp.
2. Invoke\_FIND\_ANIM\_EVENT\_PHASE – сначала идет JMP, ведущий к самой функции. На эту единственную команду JMP, есть ссылка (представленная gVar-ом) из кода инициализации нативных функций. Вывод: необходимо внутри всех функций найти всевозможные ссылки в разные места, откуда начинать анализ границ некой новой функции. Чтобы понять, что по ссылке код, а не данные, надо использовать инфу из блока секций PE формата.



1. В FUN\_7ff615dda038 последовательный вызов одной и той же функции с разными аргументами. Используются прыжки JMP, а также прерывный код.



Итого:

1. Появятся фиктивные JMP, которые перекидывают через PCode::InstructionId::NONE.
2. Пройденные инструкции можно хранить в set.
3. Обобщить asm graph building на всю программу, причем сделать алгоритм работающим в режиме потока команд. Оперируем блоками, в конце которых либо jmp, либо ret. Связи устанавливаем на ходу. Если ссылка на предыдущий созданный блок, то делить этот блок на 2. Построив граф блоков для всей программы, переходим на след. этап – определение границ функции. Для этого начинаем анализ с блоков, где пролог функции, заканчиваем, где эпилог (команда ret). Такие блоки лучше помечать на первом этапе, когда использовался decoder.
4. Короче, просто делаем asm graph building работающим в режиме потока команд. Сначала вызывается decoder, потом вызываем asm graph building, который внутри себя строить граф. Начинаем с entry функции. Игнорируя пока все внутренние call, добиваемся построения графа для этой функции. Конечных блоков (где ret) может быть несколько. Этот граф является претендентом на окончательный граф для entry функции. Далее парсим любыми средствами(легковесными) внутри этого графа все ссылки, которые могут привести нас к новым функциям в программе: в первую очередь это все call, виртуальные таблицы и т.д. Все эти ссылки добавляем в **список непосещенных адресо**в.

Далее, выбираем из списка непосещенных адресов первый адрес, и обрабатываем его. Таким образом, есть главный цикл, который пытается обработать каждый раз этот список.

Для каждого непосещенного адреса строится отдельный граф. Однако, если есть ссылка из нового графа в старый граф, то нужно указывать ссылку на блок старого графа (если надо, то разделить его), а не создавать новый блок. На следующем этапе выделения границ функции, пытаемся разрешить ситуации с пересечениями этих графов, выделяя общие ветки блоков в отдельные функции (ветка заканчивается командой ret).

Таким образом, добьемся того, что будет **список отдельных графов** для каждой функции.

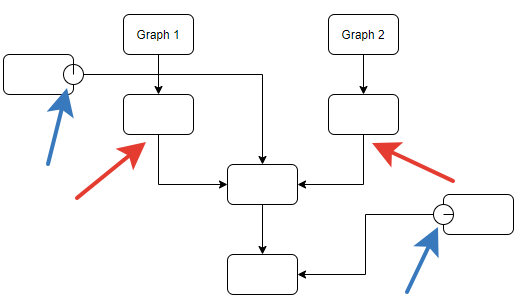
## Выделение адресов, где может располагаться код

Выделять не только функции, но и глобальные переменные! Реализовать метод containCode.

1. Вызовы call
2. VTable



## Выделение подграфов в отдельные графы



В синих стрелках – вызов функции, соответственно надо отделить подграф от графов Graph 1 и Graph 2 в красных стрелках.

# Распознавание структур

## Распознавание возвращаемого значения

Непростая задача, требующая применения разных эвристик. Логика эвристики должна быть вынесена отдельно от логики декомпиляции.

Существует 3 варианта: void, rax, xmm0. Необходимо выбрать верный.

1) Сначала декомпилируем функции с типом результат void.

2) Получив весь граф образа, декомпилировав все функции. Также можно сохранить контексты декомпиляции и анализировать их.

3) Пройтись по всем функциям, используя эвристику. Если после вызова функции запрашивался регистр rax или xmm0, то добавить к этим вариантам баллы. Если такое случалось несколько раз для одной функции, то нелинейно увеличивать баллы за этот вариант. Нелинейность увеличения исходит из правила: один раз – случайность, второй – закономерность.

### Углубление каста вниз

((p1 + 5) \* 10) + (int64\_t)(p2 \* 2 + 15), where p1 - int64\_t, p2 – uint32\_t

**Важно**: мы не должны использовать родителя с кастом на втором проходе для определения типа выражения, поскольку сам каст может удалится на текущем проходе. Необходимо это делать тогда, когда устаканятся все типы. После чего осуществить новый проход, где кастнуть этим типом два операнда, а потом рекурсивно углубляться, если каст происходит неявный. Реализовать новый метод, но с проходом сверху вниз. Если есть искусственное привидение типа из асма, то дальше не двигаться, так и должно быть!

### Правильный каст указателей со смещением

\*(uint32\_t\*)(p1 + 4) ⬄ \*(uint32\_t\*)((uint64\_t)p1 + 0x10), where p1 – uint32\_t\*

## Поиск полей структур

Изначально есть такой код:

memVar1 = \*(uint32\_t\*)((uint64\_t)p1 + 0x10)

return memVar1 \* 0.5f;

**Далее**:

1) После первого прохода создается структура struct\_1, создается поле f\_0x10 с типом uint32\_t, memVar1 приобретает тип float. Итого:

2) После второго прохода код должен быть такой:

memVar1 = \*(float\*)((uint64\_t)p1 + 0x10)

return memVar1 \* 0.5f;

3) После третьего прохода поле f\_0x10 приобретает тип float.

**Другое**:

funcVar1 = func\_1()

return funcVar1 + 0.5f;

**Вывод**: с каждым проходом тип уходит вверх (благодаря авто-переменным) в глубину.

## Вести статистику

При присваивании типов остается в итоге тот тип, у которого большой приоритет. Однако, лучшим вариантов является оставление того типа, который чаще был присвоен в разных местах программы (например, глобальной переменной).

# Проблемы / TODO

1) Поиск лишней части регистра: lea eax, [rdx-0x0B] **[Решено оптимизацией]**

2) Запрос с маской 0Xffffffff00000000 не должен затрагивать регистры eax и ниже.

3) Условие должно иметь одинаковые по размеру операнды:

((([stack\_18\_32] & 0xffff{65535}) & 0xffff{65535}) != 0xffffffffffffffff{-1}) **[Решено]**

4) Взятие символа по адресу [reg\_rsp\_64] + 0x18{24}. Нужно ввести операцию &: *&var1*.

5) Сделать отдельную либу с примерами

6) Switch, циклы

7) Параметры

8) JMP вызовы функций

9) Знаки чисел, movsx, movzx

10) float, double, векторы

11) Непонятные присваивания стековым переменным

12) ((([var\_2\_32] & 0xffffffff{-1}) & 0xffffffff00000000{0}) | (([funcVar\_49999989\_64] & 0xffffffff{-1}) & 0xffffffff{-1}))

13) Больше пояснений к коду, инфа вся в контекстах!

14) Придумать порядок оптимизации. Сначала для всех линий всех блоков только один вид оптимизации, потом второй и т.д. Для быстрой оптимизации возвращать кол-во оптимизированных штуковин, чтобы повторно не делать, если штуковин 0.

Проблема: ((5 + x) + y) + 5 => (10 + x) + y

**Главное**: оптимизируя выражение X, мы должны его заменить готовым выражением Y. Части выражения X мы не должны заменять, ибо они могут быть в других выражениях вне контекста оптимизации, что приведет к непонятным ошибкам.

**Алгоритм вкратце:**

1) Сначала все листья перемещаем в правое положение в каждой операции, но если у нас два листа, то вправо перемещаем число, если оно есть.

2) Дальше все листья сдвигаем вглубь, если знак операции имеет переместительный закон. При этом опять, как и на шаге 1) приоритет отдаем числам и сдвигаем их глубже, чем другие листья.

3) Применяем алгоритм суммирования последовательно идущих операций.

**Важно**: оптимизация должна быть парная, то есть затрагивать 2 вершины.

15) PCode в гидре. Сделать что-то подобное



Сделать программирование микроопераций объектами (как и GUI разметка). По сути, тут RISC архитектура. <https://ghidra.re/courses/languages/html/pcodedescription.html>

16) Определить 5-й, 6-й, … параметр легко, зная, на каком значении RSP вызывается функция.

Часто это 0x28, 0x30 значения: mov QWORD PTR [rsp+0x28], rax. Но не стоит забывать и про случаи присвоения параметра в другом блоке.

17) Маски. Если введем XMM регистры, то один бит = 1 байт, что плохо для вычисления маски со сдвигами с константами.

18) ([reg\_edx\_32] + 0xfffffffffffffff5{-11}) <= 0xcfffffff3{-13} - перенести константы вправо (только + и делать перед подсчетом константных выражений)

19) Сделать копию первичного дерева, где все элементы уникальны. Делать после оптимизации.

20) Сделать сравнения float значений!

21) Сделать контейнеры нодов, сделать правильный обход нодов первичного дерева. Сделать клонирование после оптимизации.

22) Добавить первоначально в контекст все входные регистры нужного размера, чтобы размер как бы вычислялся юзером, а не автоматически при первом встречном регистре. Но можно не добавлять, тогда размер определен будет автоматически.

23) Не искать дальше регистр, если в него записывается значение в 2-х случаях (в ветке main и else) в условии if-else. Переменная будет не инициализирована.

24) Использовать как можно меньше переменных, чтобы не давать имена. Для этого можно юзать неинициализированные переменные, при этом давая юзеру выбор, использовать ли эту или новую переменную. См. GET\_ANGLE\_BETWEEN\_2D\_VECTORS

25) Ленивое вычисление хеша выражений с кешом, причем вычислять можно один раз для всей ветки.

26) Трансляция в C/C++ дерево должна быть наиболее простой (потому что можно сделать в pascal потом и т.д. и код трансляции не должен много повторяться). Поэтому сделать слой промежуточной оптимизации первичного дерева, который можно применять для отображения юзеру (удобное чтение, но медленно) или не применять этот слой (для анализа и быстро). Идея – код трансляции должен быть как можно меньше и проще.

27) ([funcVar\_10\_32] &.1 0xff{255}) &.1 ([funcVar\_10\_32] &.1 0xff{255}) – исправить getHash (для операций)

28) Отождествление регистра и памяти, чтобы не плодить лишние символы неоднозначных регистров. См. SET\_ENTITY\_ANIM\_SPEED *(строка DAT\_7ff614a71d58 = 0x7e68c088;)*

Можно обобщить: если все символы неоднозначных регистров имеют одно и то же значение, то удалить их все и заменить их этим выражением (которое выглядит проще всего). Можно в этом символе хранить список возможных выражений, который будет копироваться из контекста выполнения.

~~29) Сделать еще один тип подсчета блоков наряду с линейным в LinearView, чтобы при трансляции в C/C++ не писать ненужные goto.~~

30) Вычислять маску для символов (в частности, для символов неоднозначных регистров) с точностью до бита (можно просто сложить маски всех выражений, на которые ссылается этот символ). При этом создать класс БитоваяМаска, где в качестве битовой маски будет юзаться 2 XMM регистра-переменных для скорости. См. FUN\_7ff612fa55c4 (возврат 0xFFFFFFFF)

31) Исправить баг с построением одного регистра. См. GET\_HASH\_KEY*(регистр RCX)*

32) Оптимизация условия в единый блок И не сработала. См. GET\_HASH\_KEY

33) [var\_a\_32] +.4 ([var\_a\_32] \*.4 0x8{8})

34) Присваивание параметрам (регистрам)

35) Найти зоны неиспользованных переменных и юзать их для сокращения длинных выражений

36) Удалять пустые блоки первичного дерева, но лучше не удалять, а скрывать их, ибо чтобы потом удобно было делать дебаггер

37) Объединение условий должно происходить даже при вызове функции. Само объединение, где проверяется наличие содержимого в блоке, можно производить в самом конце. Можно как в гидре, сделать присваивание внутри условия, потом через запятую само условие.

38) Разный порядок итерации в множествах (set). Это приводит к неоднозначности и может нарушить целостность проведения юнит тестов.

39) В циклах предпочтение отдается только одному типу джампа (дальнему или ближнему), однако могут быть исключения. Надо это учесть в случае непонятных багов.

40) Параллельные присваивания идут ПЕРЕД выражением условия, нарушается порядок (как и с памятью) и, как следствие, ошибки декомпиляции. См. FUN\_7ff612dc2b0c (условие в цикле)

41) [var\_4\_64] &.8 [reg\_rcx:8] должно быть просто [reg\_rcx:8] или [var\_4\_64], если они равны. **Решение**: хеш и маска для локальных переменных должны считаться по-особому, учитывая местоположение в графе.

42) Убрать наследование от INodeAgregator из некоторых классов и структур. На этапе декомпиляции (до оптимизации) никаких использований repalceWith у нодов!!!

43) var1 \* 1 -> var1

~~44) флаги размером один бит почему то интерпретируется как 1 байт, дополняя не нужными битами: [var\_1\_8] = (([var\_1\_8] &.1 0xf7{247}) |.1 (([mem\_5\_64] == 0x0{0}) \*.1 0x8{8})). Также в циклах некорректно работает hasAllRegistersGatheredOnWay.~~

~~Тут же надо заменить 2 условия одним, ибо они одинаковы, если посмотреть, какие символы к чему присваиваются. Само условие: [var\_3\_64] == 0x0{0}~~

~~45) Маска представляется как значение 64 бита и смещение!~~

~~46) Убрать replaceNode, ведь есть getNodePtrsList~~

47) Сделать нодой операцию присваивания [dstExpr] = [srcExpr], так как нужно для объединения условий: if(param1 == 0x0 && (param1->field0x10 = param2) != 0)

48) Сделать поиск параметров функции по стеку, используя тот же обход, что и для поиска регистров. Можно отдельно сделать итератор. Учесть, что функция может возвращать значение через стек или глобальную область.

49) Пересмотреть структуру кода декомпилятора, использовать паттерны, сделать интерфейсы, сопроводить многие места комментариями. Каждый модуль должен четко выполнять поставленную задачу, быть ясным для понимания.

~~50) Топовые вершины, у которых есть тип: jmp, return, func param~~

51) Очистка старых выражений после клонирования

52) Избрать такую структуру в дереве, которая будет неизменчива при изменении типов

53) Сделать для каждого терма в LinearExpr инструкцию. То есть сделать структуру Term.

54) Сделать тесты, которые будут как можно более автоматическими. Важно также проверить в тестах вычисленные типы, сравнивая результат вычисления итога при помощи скомпилированного кода из нашего декомпилятора.

55) Интерфейс для NumberLeaf. Можно исправить легко проблему каста константного слагаемого в LinearExpr

56) 2 шаг при вычислении типов сделать потом так: использовать карту “выражение->тип”, по ней на следующих шагах брать тип, который для operational node будет не обобщенным типом размером n байт, а вполне конкретным, и по нему делать каст сыновей.

57) Краткий TODO на сентябрь:

- режимы декомпиляции для автоопределения размеров входных регистров, …

- поиск стековых аргументов

- удаление лишних строк с чтением, поиск взаимозависимостей (делать после SdaBuilding)

- процедура копирования одного объекта в другой одной строкой

- вычисление всех функций в программе по прыжкам, совместно использовать декомпилятор для поиска виртуальных таблиц

- вычисление всех размеров параметров и возвращаемого значения с высокой точностью

- вычисление классов в коде, типизация всех полей определенного класса, глобальных и локальных переменных

- знаковые операции div и rem

- операции сдвига: тип для правого выражения

- убрать isSigned везде у типов, кроме системных

- не допускать таких кастов *var = (Pos\*)&entity->pos.x*. Можно просто удалить поле x

- сделать SUBPIECE для извлечения поля. Признак – наличие умножающей маски, кратной FF

- условия – те же бинарные выражения, где нужен каст двух операндов

- вычислять хеши сразу рекурсивно, причем листья должны вычисляться сложно, все остальное – просто умножение на 31 и прибавка. Учитывая дальнейшее распараллеливание блока оптимизации, отсутствие кеша хеша на производительность повлиять сильно не должно

58) Для вычисления константных выражений можно сделать виртуальную машину PCode. Это нужно, если попадется jmp [rax], но rax можно статически однозначно определить. Статическое вычисление надо делать до построения графа блоков.



Попробовать применить для оптимизации дерева.

#### Важно

У нас есть дамп памяти с данными и кодом (или реальная память). Мы этот дамп передаем анализатору функций, задача которого вычислить все функции с их областями. Все функции равноправны, они являются агрегаторами кода. Каждая функция хранит список PCode инструкций, причем промежуток (если есть прыжки) заполняется одной инструкцией NONE с размером, равным размеру промежутка. Для функций также хранить и список областей, чтобы в случае изменения области обновить хранимый PCode. Сам PCode можно хранить в файле, сериализовав каждую инструкцию. Для операндов некоторых инструкций (в частности, всех прыжков) хранить итоговое вычисленное значение, причем помечать, адрес это или нет. Адреса должны быть смещением относительно начала функции.

Декомпилятору передается список инструкций (с NONE инструкциями) и результат работы виртуальной PCode машины, которой сначала передается этот список инструкций для статического вычисления адресов (см. рис. выше) и других констант.

59) Виртуальное выполнение концевых функций для определения точного диапазона значений параметров.

60) Символизация: stack0x30 и stack0x32 (размер 4 байта) испытывают пересечение, что делает эти переменные зависимыми физически, хотя они независимы семантически

61) \*{uint32\_t\*}([rsp]) => LinearExpr (offset = 0)

62) В условиях размер(маска) левой и правой части должен быть один и тот же.

63) Каст выражения к float должен быть как (float&), ибо обычный (float) – это другая операция. То же касается и каста из float в uint64\_t, int32\_t в int64\_t.

64) Давать уникальные имена переменным. Можно использовать карту: var hash => var id

65) ОПТИМИЗАЦИЯ: во время всех этапов декомпиляции ноды не удалять, а только создавать, используя кастомный аллокатор.

66) Улучшить удаление строк, где неопределенные регистры. Не удалять те строки, где идет присваивание символам memVar, localVar, funcVar

67) Создание 128-битной арифметики и 16 байтовых масок (нужно для div/idiv). Проблема в том, что придется использовать 128-битный числовой тип для всего, а также неудобно представлять эти числа графически. 128-битность возникает в результате INT\_SEXT, которая на данный момент специально ограничена 64-мя битами, чтобы не было проблем со знаковым делением.

68) Создание полноценной зеркальной ноды перед этапом оптимизации так, чтобы вычисления не повторялись.

69) Хорошо подумать, нужно ли разделение выражений на SDA и не SDA. Нужно ли проводить этап оптимизации для SDA вершин? Оставить как есть и сделать 2 фабрики (sda/not sda) выражений? В гидре, например, много разных реализующих классов и много интерфейсов. Хотя в C++ чем меньше классов, тем лучше. В общем, максимально упростить выражения, ибо они получились слишком громоздкими. Для ускорения создания нодов необходима фабрика, в которой можно будет указать кастомный аллокатор.

70) Сделать генератор кода для ImGUI, реализовав все это через dynamic\_cast для каждого типа выражения.

71) Сделать другой алгоритм линеаризации графа потока управления. Каждый раз итеративно выбирается та система блока, которая удовлетворяет шаблону и наилучшим образом производит минимизацию метрики кол-ва операторов goto.

## Интересный трабл №1

[var\_2\_32] & 0xffffffff{-1} – может появится, когда запрос на RDX, а найден был только EDX.

**Решение**: правильная оптимизация + правильный выбор выходного регистра функции

## Интересный трабл №2

Запись в регистр EAX/ECX/… перезаписывает старшую половину регистра RAX/RCX/…, а вернее просто стирает её. При других вариациях (EAX-AX или EAX-AH) такого нет, биты не теряются.

Т.е. в контексте выполнения не могут быть одновременно регистры RAX/RCX/… и EAX/ECX/…!

Смысл команды ***mov eax, eax*** в затирании старшей части регистра rax.

Об этом написано здесь:

<https://stackoverflow.com/questions/25455447/x86-64-registers-rax-eax-ax-al-overwriting-full-register-contents>

## Интересный трабл №3

Чтение из памяти должно происходить присвоением символу результата чтения. Иначе могут быть проблемы:

mov rax, qword ptr [rsp – 0x20]

mov dword ptr [rsp – 0x20], 0x0

mov qword ptr [rsp – 0x10], rax

Есть:

\*(uint32\_t)\*([rsp] – 0x20) = 0x0

\*(uint64\_t)\*([rsp] – 0x10) = \*(uint64\_t)\*([rsp] – 0x20)

Должно быть:

var1 = \*(uint32\_t)\*([rsp] – 0x20)

\*(uint32\_t)\*([rsp] – 0x20) = 0x0

\*(uint64\_t)\*([rsp] – 0x10) = var1

**Решение**:

Использовать символы: при чтении всегда создаем символ. При оптимизации их убираем, если надо. Общий шаблон такой:

[символ] = чтение по адресу n байт (участок памяти)

<изменение этого участка памяти>

<использование символа в выражениях>

Убираем символ тогда, когда он используется один раз (кол-во родителей выражений = 1) и между созданием символа и использованием нет записи в этот участок памяти – всё это делать на этапе оптимизации:

1. Встречаем символ с кол-вом родителей выражений = 1, в который производится запись участка памяти, заносим его в *список*, а также выражение адреса этого участка памяти и кол-во байт чтения.

2. Во всех местах, где запись в память, проверяем пересечение памяти текущего записываемого участка с участком в *списке*.

## Интересный трабл №4

cmp cx, 0xFFFF -> ecx & 0xFFFF + 0xFFFF \* 0xFFFF

**Проблема**: выход из диапазона значений

**Решение**: обрезать

Нужно проводить параллель между командой ассемблера и высокоуровневым представлением. В данном случае мы берем часть регистра ecx, поэтому обрезаем не нужное (не рассматриваем старшую часть), далее мы вычитаем 0xFFFF, однако делаем сложение с умножением на -1. В результате получается большой операнд. В итоге мы записываем результат вычитания, однако рассматриваем только младшую часть. При этом есть несколько вариантов интерпретации высокоуровневого представления и решений проблемы:

1. В терминах C/C++. 0xFFFF \* 0xFFFF – это умножение двух констант типа int, значит результат будет типа int. Ошибка, так как результат сложения должен быть short. Значит мы должны умножить результат сложения на 0xFFFF.

2. Можно перенести умножение с регистра ecx на все выражение целиком: (ecx + 0xFFFF \* 0xFFFF) & 0xFFFF. Такой вариант легче, потому что минимум умножений и здесь есть логика, которая заключается в том, что запись в целевой регистр идет с обрезанием, то есть соответствие между низким и высоким уровнями в плане бит сохраняется. Причем важно отметить, что операнды обрезать нет смысла, так их старшие биты никак не влияют в итоге на младшие биты. Однако этот вариант не годен, так нет способа оптимизировать это выражение, потому что здесь идет сложение с переносом бит в старшую часть.

3. Так большинство основных операций в высокоуровневом представлении связано с операцией ассемблера, можно пометить эти операции как связанные с инструкцией pCode и сделать неизменяемую маску. Причем убрать везде явное умножение на обрезающую маску, где это сделано, чтобы просто убрать старшую часть. Таким образом мы можем иметь такое выражение *ecx + 0xFFFF \* 0xFFFFFFFFFFFFFFFF* и если учитывать скрытую маску 0xFFFF, то все будет верно. Во время оптимизации можно убрать лишние части выражения: ([reg\_rbx\_64] & 0xffffffff00000000{0} | [var\_2\_32]) – это регистр ecx, тогда убираем первый операнд побитового сложения.

**Важно 1**: при оптимизации надо обрезать *0xFFFF \* 0xFFFFFFFFFFFFFFFF*, однако это касается только константных выражений. *В случае не константы в качестве первого операнда, лучше умножать на - 1 соответствующего размера*.

**Важно 2**: при трансляции в C/C++ будем идти снизу вверх, вычислять последовательно тип за типом, и если размер вычисленного типа не совпал с маской (особенно связанного выражения), то делать каст.

**Важно 3**: при получении регистра из частей сложно ему указать неявную маску, поэтому юзаем явное умножение.

**Важно 4**: маска регистров – это одно (она может быть не с начала, например 0xFF00), а маска выражения – это по сути размер: 1, 2, 4, 8, 16, 32, … Но может быть и: 0b10

**Проблема 1**: рассмотреть случай сдвига влево (…) << 32 – нужно ли увеличивать маску?

**Проблема 2**: нужно ли для обрезания использовать побитовое умножение, ведь у нас только 64 бита максимум? То есть возможны ли случаи обрезания например регистра размера 2x до размера x, где x > 64 бит. Скажем, ymm0 на xmm0\_QQa и xmm0\_QQb? Такого вроде быть не может, ибо операции возможны только с 64 битами максимум. Однако в гидре в операции mul есть умножение 128 битных операндов, однако результат такого умножения дробится до 64 бит. На каких-нибудь машинах конечно возможно наличие >= 128 битных операций, хотя анализ проводится с использование винды на 64 битной системе. Так что в перспективе потом перейти на обрезание через функцию SUBPIECE.

## Интересный трабл №5

Если оптимизируется операция, то сначала оптимизируется левый операнд, потом правый. Из-за наличия взаимозависимости при оптимизации левого операнда может удалится/заменится правый.

**Решение**: 1) использовать двойной указатель с проверкой на nullptr 2) клонировать всё дерево перед оптимизацией

Строки как пояснения к коду

Строки – это единственное человеческое описание происходящему во время выполнения кода. В них заложен определенный смысл, они появляются в определенном месте программы с какой-то целью.

Можно классифицировать строки:

1. Логирование/debug – бывает нечасто и не везде, но несет очень много информации о происходящем. Часто может принимать параметры.
2. UI текст: сюжетные диалоги/подсказки/статистика – мало несет информации о ядре игры, разве только в месте программы, связанной с сюжетом.

Однако, можно хорошо отреверсить игровой интерфейс.

1. Пути до ресурсов игры – различные архивы, названия моделек, текстур и т.д. Поможет узнать логику загрузки из файлов-ресурсов в ОЗУ.

Таким образом, мы можем хукнуть все функции, где есть строка-аргумент, и запоминать строку в некоторую глобальную таблицу. Надо сделать занесение строк в таблицу максимально быстро. Для этого надо быстро сделать проверку на то, что строка данная уже есть в таблице.

1. Можно убрать из строки числа, оставив только буквы
2. Закодировать строку в 32-х битный хеш – это будет ключ для map
3. Сделать в map небольшую структуру, где хранить уже саму строку, список адресов вызовов функции.

Весь этот сервисный код должен работать очень быстро.

После этого можно сохранить строки в файл. Так как длины строк разные, надо пользоваться BitStream. Формат такой: [строка до \0] [кол-во адресов] {[адрес1], …}

Сделать потом удобный поиск – для этого можно использовать таблицу. Таким образом мы свяжем каждую нашу строку с определенным набором адресов, причем они будут в порядке, при котором легко отследить путь жизни этой строки: от её генерации до непосредственного отображения.

Просмотр значений структур, визуализация координат

Если посмотреть на структуру персонажа в игре, то там она очень большая. Часто из этой структуры вытекает еще куча структур, в итоге получается очень большое дерево. Реверсить такую структуру очень проблематично и долго. Однако можно заметить, что большинство полей в ней – это координаты или матрицы преобразования. Все эти координаты можно легко визуализировать.

Необходимо сделать удобное изменение значение в структурах, используя ползунки, 2д джойстики и т.д.

EXE and Libraries (DLL, Lib)

0x7FF7: GTA5.exe

0x7CD7: kernel32.dll

0x7AA7: d3d11.dll

Address space

Player\_GiveWeapon

Player\_SetSpeed

CreateFile

OpenFile

CreateD3D11Device

MSVRC.lib

rand

Есть единое адресное пространство процесса GTA5.exe. В нем расположены модули – это exe файл, а также динамические библиотеки. Статические библиотеки расположены внутри модуля GTA5.exe. Главный модуль – это исполняемый, все остальные обслуживают его и не могут работать независимо. Если посмотреть в стек, то первая функция – это main в .exe файле. Потом идем бесконечный цикл и в нем каждый кадр выполняются функции, вызовы которых образуют стек вызовов, причем вызов может быть библиотечной функции. Если функция принадлежит модулю динамической библиотеки, то команда call берет адрес функции из таблицы соотвествия.

Исполняемый модуль – агрегатор всех других модулей, которые вместе разделяют единое адресное пространство. Дочерние .exe не в счет, ибо у них другое пространство. У каждого модуля есть свой базовый адрес, относительно которого можно запомнить смещения функций и глобальных переменных.

В исходном коде программы DLL как и LIB предоставляют специальные .h файлы, в которых содержатся декларации функций. Реализации этих функций упакованы в самих модулях. Таким образом мы разделяем машинный код по разным файлам. Вызов библиотечной функции из программы – это ссылка на функцию, реализация которой в другом модуле.

Таким образом, DLL – это часть нашей программы, но просто физически она отделена от нее, но в исходниках нет. Это значит, что типы, классы и т.д.

API вызовы

В программе они виртуальные, так как из другой библиотеки вызываются. Сделать возможность в СДА добавлять библиотеки с функциями. Сигнатуры сделать типами.

Можно определить как рисуется игра.

Поиск значения в игре

В нужное место программы можно попасть по бряку на нужное место в памяти. Но чтобы найти это место в памяти, надо следить за миллионами значениями, которые постоянно изменяются. Надо сделать быстрый и гибкий алгоритм поиска таких значений в игре. Для этого я использовал векторы AVX.

Основная память:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 5 | 5 | 10 | 2 | 3 | 6 | 7 | 0 | 0 | 0 | 2 | 2 | 3 | 3 | 2 | 1 | 0 | 0 | 0 |

Буферная память:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 5 | 6 | 10 | 2 | 3 | 6 | 7 |  |  |  |  |  |  |  |  |  |  |  |  |

Должно быть как можно меньше обращения к памяти и векторных операций.

Алгоритм сравнения:

1. Берем вектор из основной памяти
2. Берем вектор из буферной памяти
3. Делаем операцию сравнения 2-х векторов, получаем маску
4. Применяем эту маску к вектору из буфера
5. Перезаписываем вектор в буферную память

Память саму делим на несколько сегментов и для каждого сегмента применяем алгоритм.

Асимптотика алгоритма устойчива относительно кол-ва найденных ячеек и зависит только от размера анализируемой памяти.

Сравнить как компилирует компилятор и, возможно, написать код на асме.

Поиск шаблона сделать тоже быстрым. Для этого использовать выравнивание. Пусть нам дан шаблон 5, 10, 3, 3 – тут 4 байта.

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 4 | 5 | 5 | 10 | 2 | 3 | 6 | 7 | 0 | 0 | 0 | 2 | 2 | 3 | 3 | 2 | 1 | 0 | 0 | 0 |

Mov rax, qword[mem]

Mov rbx, qword[mem+0x8]

Test rax, 0x10A01100110000

# GITHUB

1. Отменить все текущие изменения (все красные файлы) и вернуться к состоянию последнего коммита:

git reset --hard HEAD *(HEAD можно заменить первыми 7-ю цифрами хеша коммита)*

2.