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.

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