Dynamic Symbolic Execution

Symbolic Execution

What is symbolic Execution

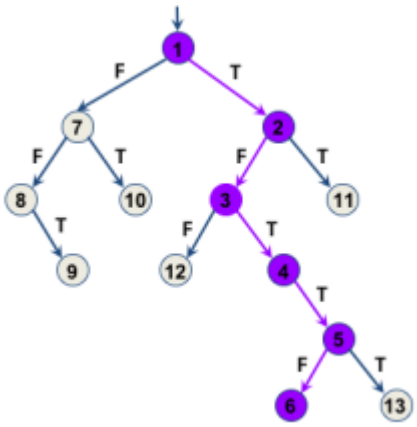
> ⓘ **Symbolic execution = symbolic + execution**

Символьные выполнение простыми словами - это нахождение пути **выполнения** с помощью **символов**

Путь выполнения

Программа - это набор инструкций, которые описывают пути выполнения код для решения какой-то определенной задачи. Если разбить программу на шаги, мы получим какой-то путь выполнения. Программа может содержать конструкции ветвления (IF ELSE), поэтому мы не сможем, выполнив программу, получить единую линию выполнения. Наши пути выполнения образуют некое древо.
Поэтому программу можно рассматривать как бинарное дерево, которое имеет бесконечную длину.
В этом дереве :

- Каждый узел представляет выполнение условного оператора
- Каждое ребро представляет выполнение последовательности необусловленных операторов
- Каждый путь в дереве представляет эквивалентный класс входных данных



Символы

Когда речь идёт о символьном выполнении, символы в этом контексте означают, что нужно рассматривать входные данные как абстрактные. то есть входные данные не имеют никакого отношения к конкретным значениям.
Например, x - символ в уравнении, при каком-то конкретном значении x получаем какое-то значение выражения.

$x^2 + 2x + 3$:

Можно сказать, что
x зависит от уравнения, ограничивающего его.
Символ же в свою очередь зависит от пути выполнения. ограничивающего его.

Что решает символьное выполнение ?

До появления символьного выполнения было популярно случайное тестирование, то есть в качестве входных данных выбирались какие-то случайные значения, затем программа запускалась с этими конкретными значениями, и изучался результат и поведение программы. Это подход может быть быстрее и проще, если вариаций значений входных переменных мало, например boolean. А как решать задачи, если домен переменной может достичь значения в 1 миллион ? Миллиард ?!

```
void test_me(int x) {
    if (x == 94389) {
        ERROR;
    }
}
```

Probability of ERROR:
$1/2^{32} \approx 0.000000023\%$

В таких случаях подход символьного выполнения приходит к нам на помощь. Поскольку как мы уже упоминали ранее, этот подход смотрит на данные не как на какие-то конкретные значение, а как символы, и анализирует путь выполнения на базе математической теории.
Символьное выполнение имеет следующие преимущества:

- Полное покрытие путей выполнения
- Обнаружение ошибок, которые могут быть пропущены при рандомном тестировании
- Анализ безопасности
- Улучшение качества кода

Как можно применять символьное выполнение

В настоящее время имеется определенный ряд бибилитек, которые поддерживают символьное выполение, например :

- angr
- miasm
- s2e
- manticore
- klee
  В нашей работе мы выберем angr, так как эта библиотека имеет следующие преимущества
- Написана на Python, поэтому легче использовать, чем другие, которые написаны на C++
- Хорошо документирована
- Имеет версии GUI, которые легко использовать, не надо создавать никакие значения
- Доступен ряд дополнительных функций : - Control-flow analysis, Disassembly, - Decompilation, ...

Angr

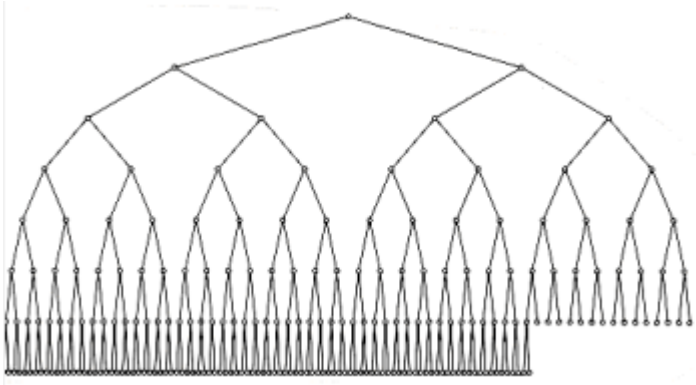
Путь выполнения

Путь выполнения представляет собой возможное выполнение программы, которое начинается и заканчивается в определённых точках
Мы можем задать начальное или финальное место

```
# start at entry point
project.factory.entry_state()
# giving start address
startAddress = 0x123456
initState = project.factory.blank_state(addr=startAddress)
# giving end address
endAddress = 0x654321
simulation.explore(find=endAddress)
```

State Explosion

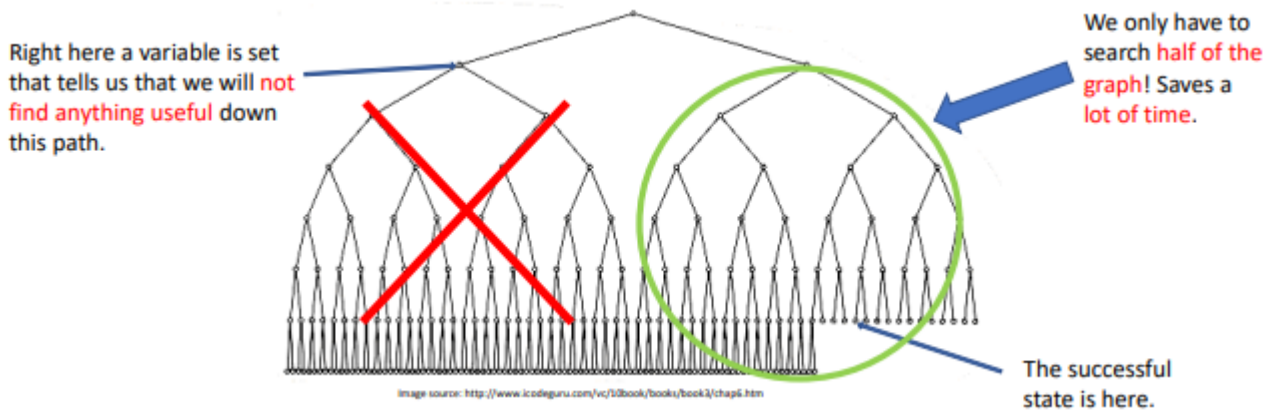
Если же мы ничего не задаём, тогда программа найдёт все возможные пути сама, и конечно это будет не эффективно для больших программ, в которых мы точно знаем, какое условие работает корректно и его можно пропустить.
Полный поиск



Angr предоставляет нам возможность выбрать пути выполнения, которые нас интересуют. Это реализует алгоритм Find and Avoid. Суть алгоритма: создаётся две предикаты и они проверяются в каждом состоянии программы.

- Is_found_path: проверяет, что в данном состоянии финальная искомая точка. Если это верно, тогда обход завершается.
- Is_avoid_path: проверяет, что для данного состояния можно пропустить проверку. (то есть нас не интересует данный путь)

```
simulation.explore(find=is_found_path, avoid=is_avoid_path)
# we also can giving address as find and avoid
simulation.explore(find=0x123456, avoid=0x123458)
```



Это ускоряет процесс обхода графа, также код становится понятнее.

Внедрение символа

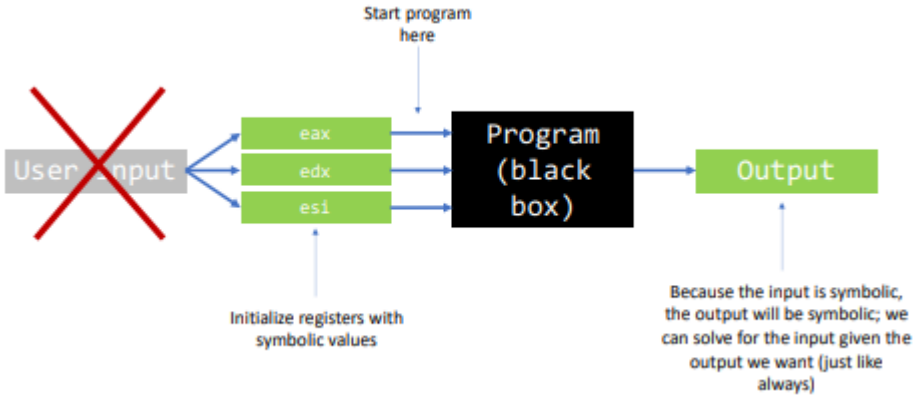
Самая интересная возможность символьного выполнения - это внедрение символа в программу. Программа рассматривается на уровень абстракции выше.
В Angr мы можем внедрять символы с помощью битовых векторов.

bitvectors

> Битовый вектор, также известный как битовый массив или битовая строка, представляет собой структуру данных, используемую в информатике для хранения последовательности битов. Каждый бит в векторе может быть равен 0 или 1, представляя два возможных состояния

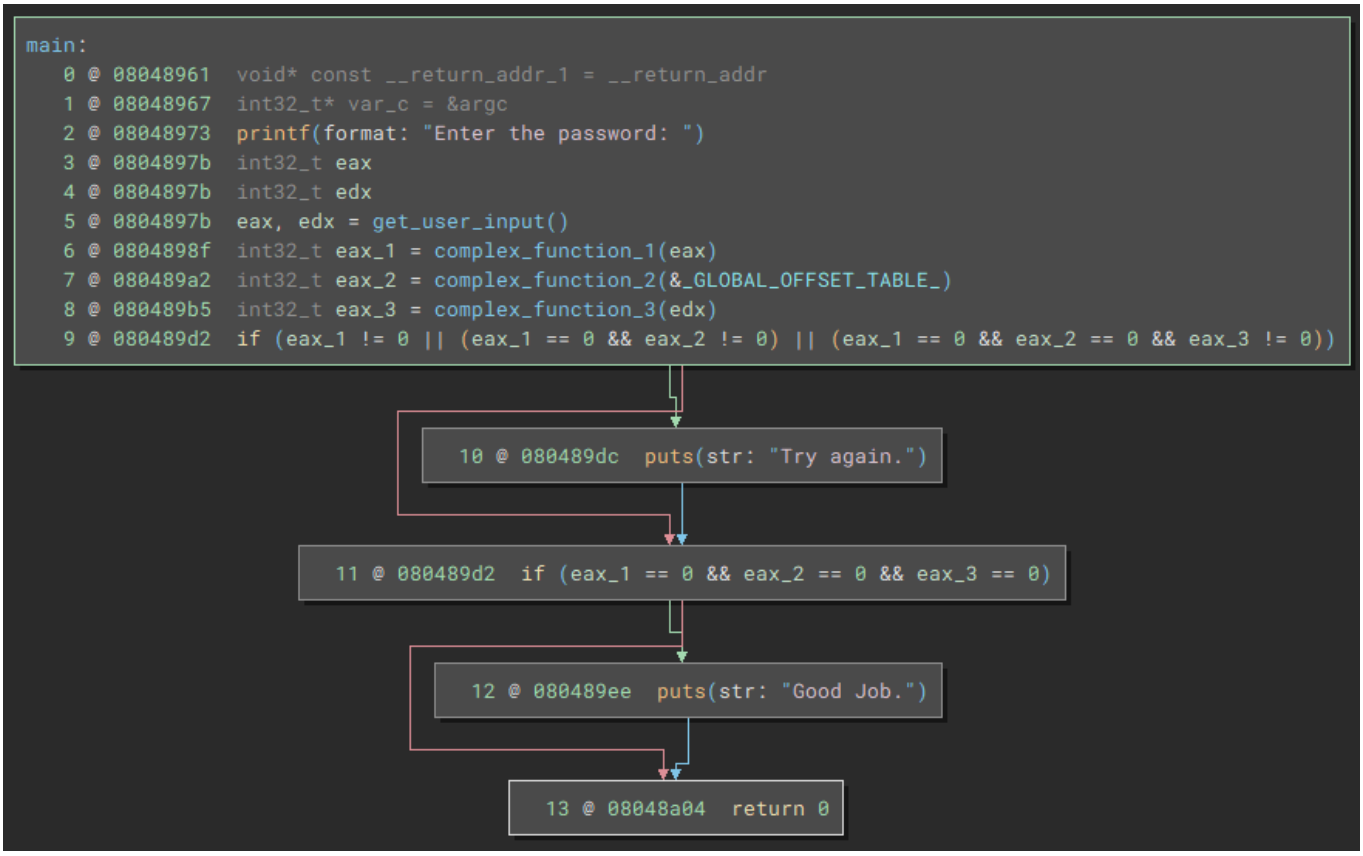
Битовый вектор может представлять собой любой тип, который может в него поместиться. Таким образом, битовый вектор может хранить любую переменную, если мы задали для нее правильный размер.
Суть внедрения символов:

1. создаётся битовый вектор, который будет представлять значение регистра (должен иметь тот же размер, что и этот регистр)
2. этот битовый вектор будет обрабатываться как символ в программе, значение состояния будет храниться в этом битовом векторе
3. когда мы достигнем нужного нам состояния, мы вычислим битовый вектор, чтобы получить конкретное значение

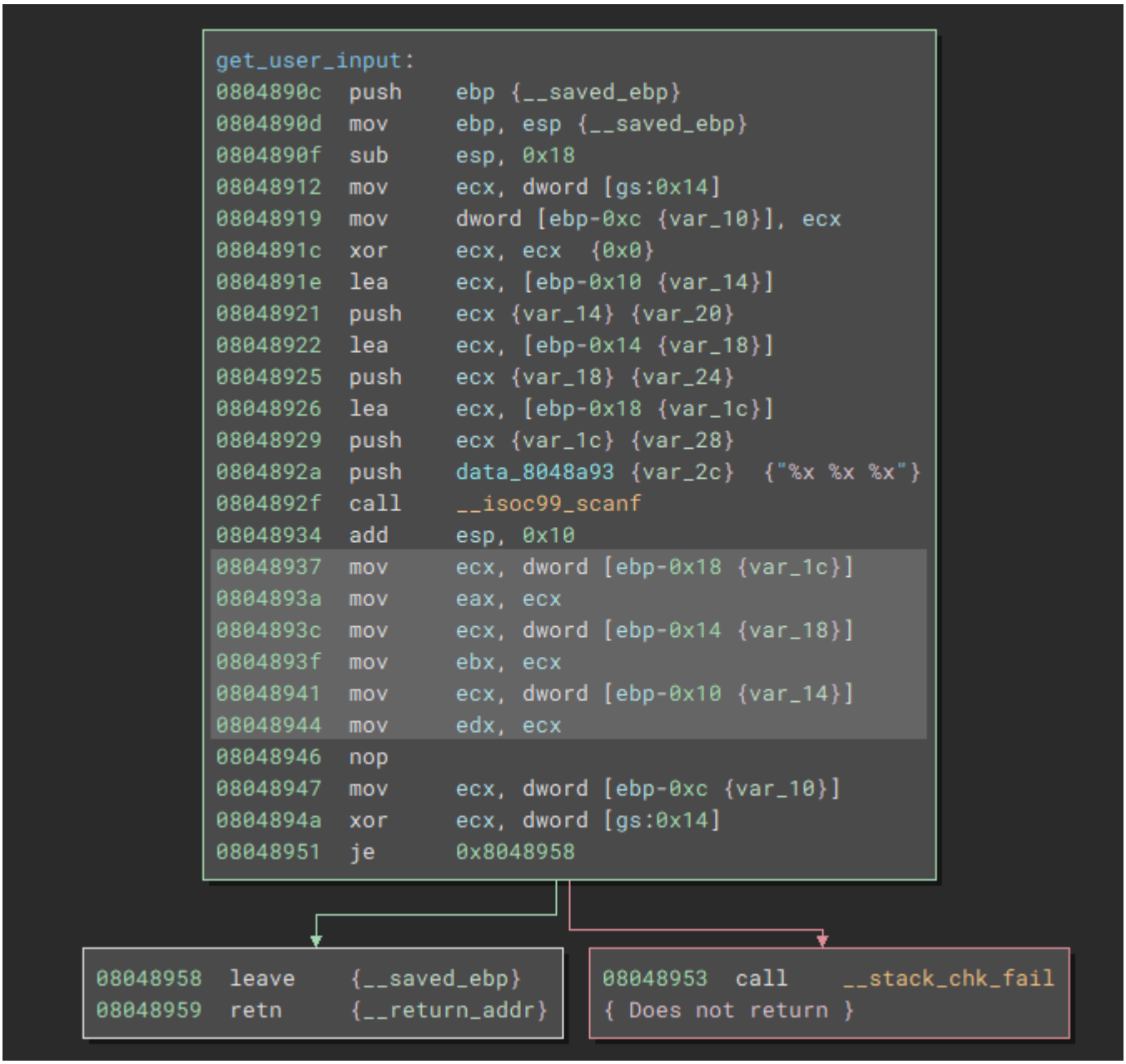


Пример :
У нас есть программа в двоичном файле, которая запрашивает у нас пароль для входа в систему. Мы можем использовать Angr и символьное выполнение, чтобы получить секретный пароль
Первый шаг, нам нужно дизассемблировать файл, чтобы получить представление о программе

На этом шаге мы используем двоичный файл ninja для разборки файла



В коде есть функция `get_user_input`, которая используется для получения входных данных от пользователя, нас интересует эта часть

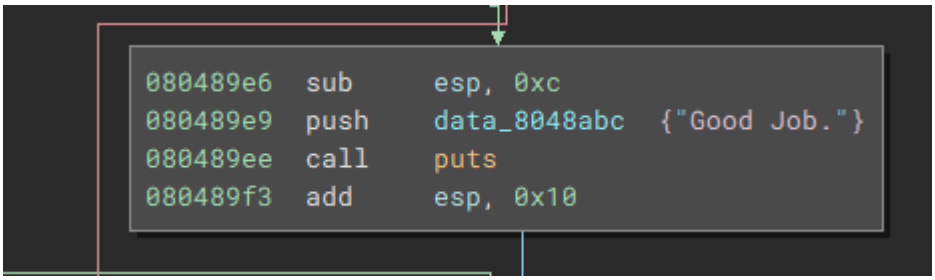


Мы видим, что значение input будет храниться в регистрах eax, ebx, edx, поэтому, чтобы получить пароль, нам нужно ввести символ в эти регистры

```
# create bitvector, each register have length of 32 bit,
# because of it store value of int32
register_length = 32
eax = claripy.BVS("eax", register_length)
ebx = claripy.BVS("ebx", register_length)
edx = claripy.BVS("edx", register_length)

# inject symbol to register
init_state.regs.eax = eax
init_state.regs.ebx = ebx
init_state.regs.edx = edx
```

Стоит заметить, что нас интересует значение этих регистров только в том состоянии, когда пароль успешно введён. Итак, когда мы получим состояние успешно введённого пароля, запишем значение регистра в этом состоянии.




```
if simulation.found:
   success_state = simulation.found[0]
   # solve symbol for value input
   passwd1 = format(success_state.solver.eval(eax), 'x')
   passwd2 = format(success_state.solver.eval(ebx), 'x')
   passwd3 = format(success_state.solver.eval(edx), 'x')
   input = passwd1 + " " + passwd2 + " " + passwd3
   print("Input is: " + input)
else:
   print("False to get input")
```

Результат выполнения программы: успешно получен верный пароль.