0.1 Введение.

Статический анализ кода - анализ программного обеспечения без непосредственного его выполнения. Динамический анализ кода - анализ программного обеспечения, проводимый при помощи его выполнения на реальном или виртуальном процессоре. С помощью статического анализа не всегда возможно получить полную и точную информацию о типах и возможных значениях использованных в программе переменных. В этом случае можно уточнить полученную на этапе статического анализа информацию с использованием методов динамического анализа.

0.2 Постановка задачи.

Будем рассматривать однопроцессорную систему с n-битной адресацией и множеством регистров $Regs$, каждый элемент которого имеет длину n.

Пусть A - некоторая ассемблерная программа.

Пусть $P = Asm(A)$ — бинарный исполнимый модуль, полученный путем компиляции программы A.

Пусть $T = Trace(P, I)$ — трасса выполнения P на некотором наборе входных данных I на однопроцессорной системе. Под трассой в общем случае будем понимать последовательность $T = \{s_0, s_1, \ldots, s_m\}$ состояний процессора. При этом каждое состояние s_i включает совокупность значений регистров процессора, и текущую инструкцию, ожидающую выполнения: $s_i = \{regVals_i : Regs \to \{0 \ldots\ 2^n - 1\}, instr_i\}$. Тогда трасса выполнения P — это трасса, последовательность $\{instr_i\}$ которой включает в себя все инструкции, выполненные в процессе исполнения P, и только их, причем ровно в той последовательности, в которой они были выполнены на процессоре.

Мы также предполагаем, что при корректном выполнении P управление не выходит за пределы загруженного в память кода, и этот код остается неизменным, то есть P не использует самомодифицирующийся и генерируемый в процессе работы код. Это дает нам основание утверждать, что между ассемблерным листингом A программы P и трассой T можно установить соответствие, то есть для каждой инструкции $instr_i$ трассы T можно указать ее положение в ассемблерном листинге A.

Пусть кроме того предварительно был проведен статический анализ A, в результате работы которого было получено множество отождествления регистров для A — $V = \{V_1, \ldots, V_k\}$. Каждое множество V_i состоит из пар $\{reg, pos\}$, где pos задает положение в ассемблерном листинге, а $reg \in Regs$. По сути V_i представляет собой ассемблерный эквивалент переменной языка C. В дальнейшем будем называть элементы множества V переменными.

Пусть также рассматриваемая нами однопроцессорная система использует страничную организацию виртуальной памяти, с отдельной таблицей страниц для каждого процесса.

При условии выполнения высказанных выше предположений, рассмотрим следующую задачу:

Задача. *Имея трассу T выполнения программы P, а также множество переменных V, полученное на этапе статического анализа, для каждой переменной $V_i \in V$ оценить вероятность того, что она является указателем.*

0.3 Решение.

Так как рассматриваемая нами однопроцессорная система использует страничную организацию виртуальной памяти, с отдельной таблицей страниц для каждого процесса, то

каждый процесс работает в своем изолированном виртуальном адресном пространстве, и, имея доступ к таблице страниц этого процесса, можно определить диапазон доступных ему виртуальных адресов. В самом простом случае можно считать, что таблица страниц процесса остается неизменной во время его выполнения, то есть не меняется и диапазон доступных ему виртуальных адресов. Это означает, что для трассы T выполнения этого процесса можно построить множество $ValidAddr \subset \{0 \dots 2^n - 1\}$ корректных адресов виртуального адресного пространства. Тогда, используя $ValidAddr$, для любого целого числа $x \in \{0 \dots 2^n - 1\}$ легко определить, может ли оно являться указателем, или нет. С использованием этого множества можно решить обозначенную выше задачу следующим образом:

На первом этапе для каждой переменной V_i строится множество всех значений, которая она принимала на трассе T — $Val_i = Val(V_i, T)$. Это можно сделать за один проход по трассе T.

Изначально все множества Val_i инициализируются пустыми: $\forall i \; Val_i^0 = \emptyset$. Далее, в силу возможности установления соответствия между A и T, для любого положения в трассе j существует соответствующее ему положение в ассемблерном листинге j_A. Используя множество V, по j_A для каждого регистра легко определить, какой переменной он принадлежит. Таким образом на каждом шаге трассы T нам известны значения всех регистров процессора, и переменные, соответствующие этим регистрам. Множества $Vals_i$ обновляются по следующему правилу:

$$Val_i^{j+1} = \begin{cases} Val_i^j \cup \{regVals_j(reg)\}, & \text{если } \exists reg : \{reg, j_A\} \in V_i, \\ Val_i^j, & \text{иначе.} \end{cases}$$

Затем для каждой переменной V_i по построенному множеству значений Val_i определяется вероятность того, что она является указателем. Для этого введем множество корректных значений указателей: $ValidPtr = ValidAddr \cup \{0\}$. Оценка выполняется на основе следующих эвристик:

1. В большинстве случаев целочисленные переменные в программе принимают небольшие значения. С другой стороны, существует нижняя грань возможных значений корректных виртуальных адресов, равная размеру одной страницы виртуальной памяти.

2. Переменная, являющаяся указателем, может принимать значения, не принадлежащие множеству $ValidPtr$. Например, рассмотрим следующий код на C:

   ```
   int sum(int *begin, int *end) {
       int result = 0;
       for(int* ptr = begin; ptr != end; result += *ptr++);
       return result;
   }
   ```

   Обращение к памяти по указателю end никогда не будет произведено, и он вполне может указывать за пределы виртуального адресного пространства процесса. Однако значение end будет входить в множество значений переменной ptr.

3. Вообще говоря, все значения целочисленной переменной вполне могут лежать в множестве $ValidPtr$, как например в случае $Val_i = \{0\}$.

Тогда в простейшем случае можно оценить вероятность того, что переменная V_i является указателем, следующим образом:

$$P(pointer(V_i)) = \frac{|Val_i \cap ValidPtr|}{|Val_i|}$$

Следует отметить, что полученное с использованием этой формулы число скорее является мерой нашей уверенности в том, что V_i является указателем.

Число $P(integer(V_i)) = 1 - P(pointer(V_i))$ определяет вероятность того, что V_i является целым числом. При $P(pointer(V_i)) \approx P(integer(V_i))$ нельзя дать точный ответ, является ли V_i указателем, или целым числом. Также если $\|Val_i\|$ мало, то полученной оценке нельзя доверять, так как она по сути является статистической — ее точность существенно зависит от количества собранной о V_i информации.

Приведенная выше формула также не учитывает того факта, что целочисленные переменные в программе обычно принимают небольшие значения, однако и она дает хорошую оценку в большинстве случаев. Приведем несколько простых способов улучшения этой оценки:

1. Использование информации, собранной с нескольких трасс выполнения P. В этом случае если имеются трассы $T_1, T_2, \ldots, T_m$, и для каждой из них построены множества значений переменных $Vals_i^{T_j}$, то можно положить

   $$Vals_i = \bigcup_{j=1}^{m} Vals_i^{T_j}.$$

   Этот прием увеличит количество элементов в множествах $Vals_i$, тем самым повысив точность оценки.

2. Использование для построения множеств $Vals_i$ не всех пар $\{reg, pos\}$ из V_i, а только тех, у которых соответствующая позиции pos инструкция ассемблерного листинга A использует регистр reg для косвенной адресации. При таком подходе приведенные при построении формулы соображения немного меняются. Так, в предположении, что тип переменной остается неизменным во всех точках программы, случай $\exists x \in Val_i : x \notin ValidAddr$ означает, что x не может быть указателем, и наоборот, $\exists x \in Val_i : x \in ValidAddr$ означает, что x - указатель.

   Рассмотрим случай, когда тип переменной не является неизменным во всех точках программы. Примером может служить следующий код на C:

   ```
   char f(int a, int b) {
       return *(a + (char *) b);
   }
   ```

   И возможный соответствующий ему код на ассемблере для x86:

   ```
   1: push ebp
   2: mov  ebp, esp
   3: push ebx
   4: mov  eax, dword ptr [ebp+8]
   5: mov  ebx, dword ptr [ebp+0Ch]
   6: mov  al, byte ptr [eax+ebx]
   7: pop  ebx
   8: pop  ebp
   9: ret
   ```

Теперь рассмотрим возможные варианты вызова функции `f`:

```
char str[] = "ab";
printf("%c", f((int) str, 0));
printf("%c", f(1, (int) str));
```

Как видно из примера, параметры функции f имеют разные типы, и поэтому относительно переменных a и b невозможно дать точный ответ на вопрос о том, являются ли они указателями, или нет. Для построения множеств $Vals$, соответствующих переменным a и b, предложенная модификация алгоритма выделит инструкцию в строке `6` ассемблерного листинга. И если в трассе встретятся оба варианта вызова функции f, то мы получим, что для соответствующих множеств $Vals$ оба множества $Vals \cap ValidAddr$ и $Vals \setminus ValidAddr$ могут оказаться не пусты, что будет сигнализировать о невыполнении предположения о неизменности типов переменных во всех точках программы.

Также возможен случай, когда полученное подмножество пар $\{reg, pos\}$ окажется пустым. Тогда данный подход неприменим, и следует воспользоваться стандартной стратегией построения множества $Vals_i$.

0.4 Техническая реализация.

Рассмотрим реализацию описанного выше метода на примере операционной системы Windows XP. Весь процесс получения информации о переменных можно разбить на несколько этапов:

1. Сбор трассы выполнения программы P. Это можно сделать с использованием какого-либо эмулятора, добавив в него возможность снятия трасс. Так, для x86-систем можно использовать QEMU [2].

2. После снятия трассы выполнения программы P, необходимо установить соответствие между инструкциями из A и T. Для этого надо узнать, по какому виртуальному адресу был загружен в память исполнимый модуль. Далее, с учетом того, что значение регистра *EIP* известно на каждом шаге трассы, легко установить соответствие между инструкциями из A и T. Узнать, по какому виртуальному адресу исполнимый модуль был загружен в память, можно несколькими способами:

   (a) В поле `image base` PE-заголовка записан базовый адрес образа исполнимого модуля в памяти, а в поле `entry point` - смещение от начала исполняемого модуля до точки входа программы [1]. В большинстве случаев для загрузки исполнимого модуля в память ОС использует адрес, заданный в `image base`, но, вообще говоря, это не гарантируется. Поэтому можно использовать простой прием - первой инструкцией трассы выполнения P будет инструкция, на которую указывает `entry point`. Зная значение регистра *EIP* на первом шаге трассы, а также значение `entry point`, можно вычислить адрес, начиная с которого исполнимый модуль был загружен в память:

   $$\texttt{imagebase} = EIP - \texttt{entrypoint}$$

   (b) “Изнутри” процесса базовый адрес образа исполнимого модуля можно узнать, выполнив следующий код:

```
1: mov eax, dword ptr fs:[00000030h]
2: mov eax, dword ptr [eax+8]
```

По адресу `fs:[0]` располагается структура Thread Environment Block (*TEB*). Первая инструкция загружает в `eax` одно из полей этой структуры - указатель на Process Environment Block (*PEB*). В структуре *PEB* есть поле ImageBaseAddress, его и загружает в `eax` вторая инструкция.

С учетом того, что базовый адрес образа исполнимого модуля не меняется во время выполнения программы, этот подход можно использовать в случае, если есть возможность модификации A, или если возможно изменить код используемого эмулятора, добавив в него необходимую функциональность.

(c) Базовый адрес образа исполнимого модуля также можно узнать "извне" процесса, используя функцию NtQueryInformationProcess из `ntdll.dll`:

```
typedef NTSTATUS (NTAPI *pfnNtQueryInformationProcess)
    (HANDLE, PROCESSINFOCLASS, PVOID, ULONG, PULONG);

void* GetImageBase(HANDLE hProcess) {
    pfnNtQueryInformationProcess NtQueryInformationProcess;
    HMODULE hNtDll;
    PROCESS_BASIC_INFORMATION pbi;
    PEB peb;

    hNtDll = LoadLibrary(_T("ntdll.dll"));

    NtQueryInformationProcess =
        (pfnNtQueryInformationProcess)
        GetProcAddress(hNtDll, "NtQueryInformationProcess");

    NtQueryInformationProcess(hProcess,
        ProcessBasicInformation, &pbi, sizeof(pbi), NULL);

    ReadProcessMemory(hProcess,
        pbi.PebBaseAddress, &peb, sizeof(peb), NULL);

    FreeLibrary(hNtDll);
    return peb.ImageBaseAddress;
}
```

Структура PROCESS_BASIC_INFORMATION и перечисление PROCESSINFOCLASS объявлены в заголовочном файле <winternl.h>. Описание же структуры PEB лучше посмотреть в заголовочных файлах ReactOS [3] - то описание, которое предлагает Microsoft в MSDN, является неполным.

Описанная выше функция GetImageBase возвращает базовый адрес образа исполнимого модуля по дескриптору процесса. Приведенный код является лишь примером — в частности, в нем не реализована обработка возможных ошибок. GetImageBase можно в частности использовать в отдельном процессе внутри эмулятора, который будет запускать программу P на выполнение с использованием функции CreateProcess в режиме CREATE_SUSPENDED, затем использовать GetImageBase для получения базы образа, после чего вызывать ResumeThread для основного потока P.

3. Также для применения описанного выше алгоритма необходимо построить множество $ValidAddr$. Для этого, во-первых, необходимо знать размер одной страницы виртуальной памяти. Это можно сделать с использованием следующей функции:

```
DWORD GetPageSize() {
    SYSTEM_INFO si;
    GetSystemInfo(&si);
    return si.dwPageSize;
}
```

Кроме того необходимо уметь получать множество адресов виртуальной памяти, доступных процессу в данный момент. Стоит отметить, что при построении алгоритма мы считали, что множество $ValidAddr$ неизменно, в то время как множество адресов виртуальной памяти, доступных процессу, постоянно меняется. Для сохранения корректности алгоритма достаточно взять $ValidAddr$ равным объединению множеств доступных адресов виртуальной памяти по всем шагам трассы T.

Пусть $ValidAddr_i$ — множество адресов виртуальной памяти, доступных процессу на i-м шаге трассы T. Ясно, что нерационально вычислять $ValidAddr_i$ на каждом шаге трассы, так как это множество может меняться только при обращении к системным вызовам работы с памятью. Для того, чтобы пересчитывать его только в местах возможного изменения, нужно или установить hook на некоторые из системных вызовов, или расширить функциональность эмулятора. Вычисление множества $ValidAddr_i$ можно производить с использованием функции VirtualQueryEx следующим образом:

```
void GetValidAddr_i(HANDLE hProcess) {
    MEMORY_BASIC_INFORMATION mbi;
    PCHAR p = 0;
    while(VirtualQueryEx(hProcess, p, &mbi, sizeof(mbi)) != 0) {
        if(mbi.State == MEM_COMMIT) {
            /* Обработка.
             * В полях mbi.BaseAddress и mbi.RegionSize содержатся
             * базовый адрес и размер очередного блока виртуальной
             * памяти.
             */
        }
        p += mbi.RegionSize;
    }
}
```

При таком подходе итоговое множество $ValidAddr$ строится просто как объединение всех полученных множеств $ValidAddr_i$.

Литература

[1] Microsoft Portable Executable and Common Object File Format Specification.
http://www.microsoft.com/whdc/system/platform/firmware/PECOFF.mspx

[2] QEMU - open source processor emulator.
http://bellard.org/qemu/

[3] React Operating System.
http://www.reactos.org/