И. В. Арзамарцев, И. А. Юров

МЕТОД АНАЛИЗА ЗАВИСИМОСТЕЙ МЕЖДУ ВХОДНЫМИ И ВЫХОДНЫМИ ДАННЫМИ АЛГОРИТМОВ

В статье предлагается метод выявления зависимостей между входными и выходными данными алгоритмов. Такие зависимости являются математическими соотношениями между входным и выходным набором данных. Выявление подобных зависимостей играет важную роль в обеспечении информационной безопасности автоматизированных систем. С использованием анализа зависимостей между входными и выходными данными алгоритмов возможны разработка антивирусных программ, опирающихся как на сигнатурные, так и на проактивные методы выявления вредоносных данных, создание брандмауэров, позволяющих выявлять вредоносные данные в информационном потоке, верификаторов исходного кода программного обеспечения, позволяющих проводить процесс верификации кода на наличие ошибок и уязвимостей.

Под алгоритмом в данной статье подразумевается набор инструкций некоторого языка программирования, позволяющего осуществлять арифметические операции, операции присваивания и ветвления. В рамках данной статьи в качестве такого языка будет использоваться язык «С».

Предлагается использовать следующий метод анализа алгоритмов:

- определение выходного набора данных;
- построение потока управления анализируемого алгоритма;
- модифицированное символьное исполнение анализируемого алгоритма.

Определение выходного набора данных осуществляется либо вручную, на основе определения набора переменных, влияние на которые необходимо отследить, либо автоматически в случае представления алгоритма в виде некоторой функции. При этом будут отслеживаться все аргументы функции, переданные указателю на данные, не являющиеся константными (в соответствии с семантикой языка «С»), а также результирующее значение функции.

Результатом построения потока управления анализируемого алгоритма служат граф вызовов и граф потока управления, построение которых не является темой данной статьи и описано в книге [1].

Символьное исполнение (от англ. symbolic execution) — метод анализа потоков данных, основанный на построении алгебраических уравнений, описывающих входные и выходные параметры некоторого функционального блока исследуемого ПО. Символьное исполнение реализуется путем подстановки вместо реальных значений входных данных переменных величин и исполнения программы [2].

Применение известных вариантов символьного исполнения [3–5] не позволяет в общем случае осуществить полноценный анализ циклических конструкций в силу наличия большого либо бесконечного числа итераций при некоторых внутренних условиях.

Для повышения эффективности символьного исполнения предлагается использовать:

- функции ветвления;
- графовое представление данных;
- выявление инвариантных взаимосвязей между данными;
- склеивание потока данных в итерациях цикла.

Для сокращения времени работы алгоритма символьного исполнения за счет уменьшения количества рассматриваемых ветвей предлагается использовать следующую функцию ветвления.

Определим функцию $ib_a(x) = 1$, если $x < a$, и 0 в противном случае («ib» — сокращение от «if below»).

Через данную функцию можно также определить следующие полезные функции:
- $ie_a(x) = [1 - ib_a(x)] * [1 - ib_x(a)]$ (if equal $ie_a(x) = 1$, если x = a, и 0 в противном случае);
- $ig_a(x) = 1 - ib_a(x) - ie_a(x)$ (if grow $ig_a(x) = 1$, если x > a, и 0 в противном случае).

Тогда с помощью данных функций можно задать любое логическое условие. Рассмотрим для примера следующую функцию:

```
int simpleFunc(int x)
{
if (x > 5)
x = x + 10;
return x;
}.
```

Данная функция может быть представлена в виде $x_{new} = x_{old} + ig_5(x) * 10$.

Если условие содержит несколько логических элементов, то их можно представить с помощью умножения в случае конъюнкции и с помощью сложения в случае дизъюнкции.

Так, сложение, умножение и вычитание позволяют построить выражение для псевдобулевой функции: $x1 \mid x2 = x1 + x2 - x1 * x2$, где x1, x2 {0,1}.

Для примера, если немного усложнить предыдущую функцию:

```
int simpleFunc(int x)
{
if ((x > 5) && ((x == 1) || (x == 2)))
x = x + 10;
return x;
},
```

то получим $x_{new} = x_{old} + [ie5(x) *\{ie_1(x) + ie_2(x) - ig_1(x) * ie_2(x)\}] * 10$.

Использование данной функции является удобным при расчете значений, которые должны принять входные параметры некоторого блока кода для достижения тех или иных результатов.

Графовое представление данных оказывается полезным в случае, когда код осуществляет работу с данными, находящимися в разных областях памяти. В этом случае предлагается представлять данные в виде ориентированного графа, где в качестве «родительских» вершин выступают переменные, на основе которых был получен новый набор переменных-потомков. Набор переменных-потомков также представляется в виде вершин, в которых заканчиваются дуги, идущие из соответствующих родительских вершин.

Алгоритм графового представления данных выглядит следующим образом:
- осуществить символьное исполнение исследуемого участка кода;
- выделить операции обращения с памятью, в которых переменная, содержащая адрес памяти, зависит от другой переменной, содержащей некоторый адрес памяти;
- в случае нахождения зависимости переменная, участвующая в вычислении другого адреса, помечается как «родительская», а полученный в результате вычислений адрес — как «потомок»;
- провести дуги из родительских вершин в соответствующие им вершины-потомки.

Для примера рассмотрим небольшую функцию, осуществляющую обход односвязного списка, представленную ниже на рис. 1:

```
Node* ProcessList(Node* node, int searchedValue)
{
    for (Node* curNode = node; curNode != NULL; curNode=curNode->next)
    {
        if (curNode->value == searchedValue)
            return curNode;
    }
    return NULL;
}
```

Рис. 1. Функция поиска значения в односвязном списке

При использовании алгоритма графового представления данных получим, что корнем графа является переменная node. Обращение к новым вершинам данных осуществляется последовательно через атрибут next. Завершением обхода является нахождение искомой величины, либо по достижении NULL в качестве следующей вершины.

Полученные сведения о методах обращения к элементам памяти позволят описать использующийся в данном контексте односвязный список для его последующей обработки алгоритмом символьного исполнения.

Такое представление описывает произвольную структуру данных, однако может быть неточным, поскольку может содержать в качестве разных вершин с точки зрения реализации алгоритма один и тот же участок памяти.

Выявление инвариантных взаимосвязей между данными заключается в нахождении участков кода, направленных на изменение некоторого набора данных таким образом, что независимо от начальных значений данные на выходе будут обладать определенными свойствами.

Алгоритм работы выявления инвариантных взаимосвязей между данными может быть представлен следующим образом:

- определение взаимосвязи значений данных после выполнения анализируемой операции;
- сопоставление условия проведения операции, взаимосвязи значений данным, при которых условие выполнимо и невыполнимо;
- если результат не зависит от начальных значений, то мы нашли инвариантную зависимость между результирующими значениями данных.

Описанный подход программирования является весьма распространенным при обработке различных данных с целью придания им некоторого шаблонного вида. Для примера рассмотрим следующий блок кода, использующийся в алгоритмах сортировки:

if ($a_1 < a_2$)

swap(a_1, a_2).

Из приведенного выше алгоритма получим два возможных исхода:

- если $a_1 < a_2$, то после выполнения операции $a_1 >= a_2$;
- если $a_1 \geq a_2$, то после выполнения операции $a_1 >= a_2$.

Следовательно, независимо от начальных значений переменных a1 и a2 после выполнения данного блока кода всегда будет реализовано условие $a1 \geq a2$.

Склеивание потока данных в итерациях цикла осуществляется посредством построения специальной таблицы, каждая строка которой содержит возможные значения переменных после завершения очередной итерации.

Новые строки в данной таблице появляются в соответствии с предлагаемым алгоритмом в результате следующих действий:

- построение графа потока управления и выделение в нем вложенной итерации цикла;
- символьное исполнение итерации цикла с применением функций условий;

- выделение из полученных уравнений всевозможных результирующих значений переменных с учетом ограничений на значения переменных;
- выделение результирующих значений итерации, определяющих монотонное изменение значений переменных;
- рекурсивное продление итерации посредством подстановки полученного множества значений переменных после итерации с учетом данных, оперируемых в новой итерации.

Рассмотрим в качестве примера вложенный цикл алгоритма пузырьковой сортировки, представленный ниже на рис. 2:

```
for (int j = 0; j < n - 1; j++)
{
    if (arr[j] > arr[j+1])
    {
        int b = arr[j+1];
        arr[j + 1] = arr[j];
        arr[j] = b;
    }
}
```

Рис. 2. Программный код вложенного цикла пузырьковой сортировки

Из анализа графа потока управления на наличие циклов получим в качестве итерации цикла блок кода if из приведенного выше рисунка, а также условие выхода из цикла j >= n – 1.

Далее, осуществляя символьное исполнение данной итерации цикла, получим следующие выражения:

$arr[j] = gn_{arr[j]}(arr[j + 1]) * arr[j + 1] + be_{arr[j]}(arr[j + 1]) * arr[j]$,
$arr[j + 1] = gn_{arr[j]}(arr[j + 1]) * arr[j] + be_{arr[j]}(arr[j + 1]) * arr[j + 1]$,
$j_{new} = j_{old} + 1$.

Откуда получим, что $arr[j]_{new} = \{arr[j], arr[j + 1]\}$ и $arr[j + 1]_{new} = \{arr[j + 1], arr[j]\}$ в зависимости от выполнения условия arr[j] > arr[j + 1], а также $j_{new} = j + 1$ вне зависимости от условий.

Тогда, рассматривая следующую итерацию цикла с учетом полученной ранее информации о множестве возможных значений элемента массива arr[j + 1], можно получить, что

$arr[j + 1]_{new} = \{arr[j + 2], arr[j + 1], arr[j]\}$ и $arr[j + 2]_{new} = \{arr[j + 2], arr[j + 1], arr[j]\}$, $j_{new} = j + 1$.

После завершения рекурсивного повторения этих итераций будет получен результат: $arr[i]_{new} = \{arr[k], k = 0;i\}$, j = n – 1.

Можно рекурсивно применять ограничения на результирующие значения переменных, если осуществить привязку условий к значениям переменных, введя вспомогательные функции min и max.

В данном примере получим, что, поскольку

$$arr[j]_{new} = \begin{cases} arr[j], \text{ если } arr[j] \leqslant arr[j+1] \\ arr[j+1], \text{ в противном случае} \end{cases},$$

следовательно:
$arr[j]_{new} = \min\{arr[j]; arr[j + 1]\}$,
$arr[j + 1]_{new} = \max\{arr[j + 1], arr[j]\}$.

Аналогичным образом рассматривается монотонное увеличение переменных внутри цикла для определения возможных границ значений переменной.

Предложенные в данной статье методы анализа алгоритмов способствуют улучшению существующих методов анализа программного обеспечения, таких как символьное исполнение и использование taint-меток [6]. Эти методы могут применяться как к распространенным языкам программирования, так и к разнообразным архитектурам наборов машинных команд. Указанные методы позволяют выявить общую схему потока данных и могут использоваться в доказательствах эквивалентности алгоритмов, при поиске входных данных, обеспечивающих реализацию определенного пути в теле ПО, и т. п.

Таким образом, предложенные в статье методы анализа алгоритмов позволят улучшить существующие подходы, используемые в антивирусных программах, брандмауэрах, верификаторах и т. п., что, в свою очередь, позволит повысить уровень защищенности объектов информатизации, на которых они используются.

Однако предложенные методы не позволяют провести полноценный анализ сложных программных выражений, например анализ вычисления однонаправленных функций. В дальнейшем предполагается исследование потока данных в сложных циклических конструкциях и формальное доказательство возможности или невозможности исполнения определенных ветвей программ.

СПИСОК ЛИТЕРАТУРЫ:

1. *Ахо А. В., Моника С., Джефри Д.* Компиляторы: принципы, технологии и инструментарий. 2-е изд. Пер. с англ. М.: ООО «И. Д. Вильямс», 2008.
2. *King J. C.* Symbolic Execution and Program Testing // Magazine Communications of ACM. 1976. Vol. 19. P. 385–394.
3. *Suzette P., Guowei Y., Neha R., Sarfraz K.* Directed Incremental Symbolic Execution // Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation. 2011. P. 504–515.
4. *Sarfaz K., Corina S., Willem V.* Generalized Symbolic Execution for Model Checking and Testing // Proceedings of the 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. 2003. P. 553–568.
5. *Saxena P., Poosankam P., McCamant S., Song D.* Loop-Extended Symbolic Execution on Binary Programs // Proceedings of the 18th International Symposium on Software Testing and Analysis. 2009. P. 225–236.
6. *Newsome J., Song D.* Dynamic Taint Analysis for Automatic Detection, Analysis and Signature Generation for Exploits on Commodity Software // Network and Distributed Systems Security Symposium. 2005. P. 60–70.