И. В. Арзамарцев, Г. И. Борзунов

ДЕТЕРМИНИРОВАННЫЙ АЛГОРИТМ АНАЛИЗА КОДА ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

В данной работе представлен алгоритм анализа программного обеспечения (ПО), позволяющий вычислить необходимые значения входных данных ПО для получения искомого состояния ПО. Этот алгоритм основан на использовании схемы алгоритма символьного исполнения [1] и применении ранее предложенных авторами алгоритмов анализа сложных циклических конструкций [2].

В предыдущих работах задача анализа кода ПО решалась посредством введения эвристик, на основании которых устанавливался приоритет путей исполнения кода. Так, в работе [4] предпочтение отдавалось путям, которые:

- исполняют ранее не покрытые блоки кода;
- исполняют операции с памятью;
- исполняют операции с символьными указателями.

Данная эвристика позволяет отдавать предпочтение тем путям, исполнение которых наиболее вероятно приведет к проявлению дефекта в ПО. Также авторами был представлен новый подход по обработке раздвоений символьного исполнения при выполнении некоторого условия в коде, который не приводил к экспоненциальному расходу оперативной памяти и позволял осуществлять приоритизацию исполнения выбранных путей. В работе [5] был представлен метод эвристического анализа циклов, в котором осуществляется составление карты всевозможных путей исполнения цикла, с присвоением каждому уникального номера и длины пути. Затем в соответствии с геометрическим распределением осуществляется выбор полученных путей. Предложенный метод, как отмечают сами авторы, имеет ограниченное применение, хотя в некоторых случаях позволяет в несколько раз сократить вычислительные затраты по сравнению с простейшим алгоритмом символьного исполнения, осуществляющим исполнение всех возможных путей, а также в случае случайного выбора исполняемого пути. Таким образом, разработка новых подходов к решению рассматриваемой задачи остается актуальной.

В данной работе предлагается детерминированный алгоритм решения задачи анализа кода ПО. Код любого ПО можно разбить на последовательно исполняющиеся блоки кода, осуществив последовательный анализ которых можно провести полный анализ программы. В данной работе предлагается следующее разделение программы на блоки:

- инструкции последовательно исполняющегося кода, не содержащие циклических конструкций;
- инструкции последовательно исполняющегося кода, содержащие циклические конструкции.

Блоки последовательно исполняющегося кода поддаются анализу посредством символьного исполнения из [1, 2], что позволяет найти для каждого такого блока при заданном наборе выходных данных соответствующий ему набор входных данных.

С использованием описанного в [2] алгоритма развертывания циклов можно предложить следующий способ оптимизации символьного исполнения при вычислении входных данных для блока кода, содержащего цикл.

Рассмотрим произвольную итерацию цикла, не имеющего вложенные циклы, которую можно представить следующим образом (см. рис. 1).

Рис. 1. Итерация цикла

Под действием подразумевается последовательность некоторых арифметических операций в рамках некоторого условия языка программирования.

Каждое действие обладает следующими свойствами:

- может быть выполнено или не выполнено в зависимости от значений переменных, участвующих в условии его выполнения;
- либо уменьшает, либо увеличивает значение некоторых переменных.

Действие также может добавлять символьную зависимость от внешней переменной, значение которой определит, будет значение увеличено либо уменьшено.

Следовательно, каждое такое действие можно представить следующим образом (см. рис. 2).

Рис. 2. Схема представления действий итерации цикла

Здесь под переменной подразумевается некоторый адрес в памяти и размер. Под типом изменения значения — ее увеличение либо уменьшение. Под ограничениями внешних переменных — диапазоны значений переменных, при которых в рассматриваемом действии произойдет искомый тип изменения значения.

Обозначим такое представление действия как схему действия.

Схема действия может быть получена посредством символьного исполнения действия и выделения:

- переменных, участвующих в условии выполнения данного действия и получении диапазонов их значений, при которых действие будет выполнено;
- всех арифметических операций, участвующих в данном действии;
- выходных переменных в полученных арифметических операциях;

- значения других аргументов арифметической операции и их диапазонов значений для уменьшения/увеличения значения выходных переменных данной операции.

Рассмотрим модельный пример: пусть имеется цикл, представленный на рис. 3:

```
1.  for (int i = 0; i < ArraySize; ++i)
2.  {
3.      if (numbersArray[i] < 0)
4.          ++positiveNumbersCount;
5.
6.      ++totalNumbers;
7.  }
```

Рис. 3. Цикл по обработке массива чисел

В соответствии с представленным выше описанием итерация данного цикла состоит из трех действий: условия на строках 3–4, единичной операции на строке 6, а также действия по увеличению счетчика и перехода на следующую итерацию цикла.

Соответственно схема для действия 1 выглядит следующим образом:

[positiveNumbersCount] {Increase [+1]} (numbersArray[i] < 0);

для действия 2:

[totalNumbers] {Increase [+1]} ();

для действия 3:

[i] {Increase [+1]} ().

Имея такие схемы для каждого действия итерации цикла, можно построить алгоритм по направленному символьному исполнению, направляя поток управления в сторону увеличения/уменьшения значения переменной в зависимости от контекста задачи.

Предположим, что в ходе символьного выполнения цикла на некотором наборе входных данных мы получили формулу, описывающую значение искомой переменной большее, чем требовалось. Тогда мы можем попытаться скорректировать ее значение посредством изменения входного значения данных таким образом, что направим символьное исполнение в действие со схемой, которое уменьшит это значение, либо не будем направлять в действие со схемой, увеличивающей значение искомой переменной. Последовательно осуществляя «перебор» таких попыток, мы сможем найти нужное значение набора входных данных либо доказать, что его не существует.

В случае наличия символьной зависимости искомой переменной от входных данных в результате выполнения некоторого действия необходимо учитывать возможные диапазоны значений полученных входных данных и исходя из границ данных диапазонов осуществлять увеличение/уменьшение зависимости результирующего значения от констант.

Отдельно следует рассмотреть условие повторения цикла и участие в нем изменяемых в теле цикла переменных. Каждый цикл может иметь несколько точек выхода в своей итерации, каждая из которых может быть представлена предикатом от одной или нескольких переменных. В общем случае возможно довольно сложное формирование условий выхода из цикла, но весьма распространенными являются следующие циклические конструкции:

- циклические конструкции со счетчиком ссылок, в которых используется специальная переменная, к которой на каждой итерации применяется определенное арифметическое действие;
- циклические конструкции с условием выхода, в которых условия выхода не зависят от предыдущих итераций.

Оба данных случая легко представимы при проведении символьного исполнения, при этом для анализа циклов первого типа планируется использовать метод рекуррентных выражений из работы [3].

Для включения в анализ циклической конструкции условий выхода из цикла необходимо:
- выделить все точки выхода из цикла, а также переменных, участвующих в условии выхода из цикла;
- определить все действия, изменяющие переменные условия выхода из цикла;
- при символьном исполнении точки выхода проверять, получено ли искомое значение выходных данных, и в этом случае с учетом п. 2 проверить, можно ли осуществить выход из цикла.

Предлагаемый метод состоит из этапов инициализации и вычислений.

Этап инициализации отвечает за первоначальный анализ итерации цикла и заключается в выполнении:
- выделения всех действий и составления их схем работы;
- выделения всех точек выхода из цикла, а также переменных, участвующих в условии выхода из цикла;
- выделения всех действий над результирующим значением искомой переменной и найденными ранее переменными;
- выделения всех переменных, которые участвуют в найденных ранее действиях в качестве аргументов арифметических операций, а также переменных, принимающих участие в разветвлении потока управления при осуществлении ранее найденных операций;
- повторения предыдущих операций, пока не останутся зависимости от констант и внешних по отношению к циклу переменных и входных данных.

На этапе вычислений на основе полученной на этапе инициализации информации предпринимается попытка найти искомые значения входных данных путем последовательного исполнения итераций цикла следующим образом:
- произвольным образом выполнить цикл, стараясь приблизиться к искомому значению, исходя из того, что если текущее значение меньше искомого, то выполняем действие, увеличивающее значение, иначе уменьшающее;
- сравнить полученное результирующее значение из предыдущей операции с искомым результатом, в случае несовпадения определить необходимость увеличения либо уменьшения результирующего значения;
- в соответствии с приведенным выше алгоритмом обхода последовательно выбирать соответствующие действия;
- при анализе точки выхода проверять, получено ли искомое значение выходных данных, и в этом случае проверить, можно ли осуществить выход из цикла посредством анализа формулы символьного выражения переменных в условии выхода;
- после каждой итерации осуществлять проверку невозможности продолжения, проверяя монотонность последующих изменений, в случае возможности продолжить алгоритм, перейти к п. 2 алгоритма.

Предлагаемый алгоритм обеспечивает определение за детерминированное число шагов искомых наборов входных данных, необходимых для получения результирующих значений выходных данных произвольного блока кода. С помощью данного алгоритма можно, начиная с последнего блока кода, последовательно определить искомые входные данные, необходимые для получения произвольного состояния последнего блока кода ПО, проверив его таким образом на наличие/отсутствие определенных свойств.

Результаты данной работы могут быть использованы для анализа программного обеспечения с целью выявления:
- ошибок в алгоритме исполнения;
- уязвимостей;
- недокументированных возможностей;
-вредоносной логики исполнения.

СПИСОК ЛИТЕРАТУРЫ:

1. *King J. C.* Symbolic Execution and Program Testing // Magazine Communications of ACM. 1976. Vol. 19. P. 385–394.
2. *Арзамарцев И. В., Юров И. А.* Метод анализа зависимостей между входными и выходными данными алгоритмов // Безопасность информационных технологий. 2013. № 2. С. 18–22.
3. *Carette J., Janicki R., Zhai Y.* Program verification by calculating relations // Conference of Applied Simulation and Modeling. 2006. Track 522-112.
4. *Sang Kil Cha, Avgerinos T., Rebert A., Brumley D.* Unleashing Mayhem on Binary Code // Proceedings of the 2012 IEEE Symposium on Security and Privacy. P. 380–394.
5. *Babić D., Martignoni L., McCamant S., Song D.* Statically-directed dynamic automated test generation // Proceedings of the 2011 International Symposium on Software Testing and Analysis. P. 12–22.