А. В. Филоненко, И. К. Исаев, Д. В. Сидоров, Ю. М. Туманов

ПОИСК УЯЗВИМОСТЕЙ ПО БИНАРНОМУ КОДУ С ПОМОЩЬЮ ПРОВЕРКИ ВЫПОЛНИМОСТИ ОГРАНИЧЕНИЙ

Качественное программное обеспечение является важным условием надежности любой информационной системы, но часто приходится иметь дело с программным обеспечением, оформленным в виде готового к работе бинарного кода, без исходных текстов и отладочной информации. Это порождает необходимость в средствах верификации, позволяющих не только выявить факт наличия уязвимости, определить ее тип и причины возникновения, но и предоставить тест, демонстрирующий ее проявление, с которым в дальнейшем можно обратиться к разработчику.

Статические, динамические и формальные методы верификации не лишены недостатков, и на данный момент активно развиваются синтетические, главной идеей которых является сочетание основных подходов, чтобы компенсировать их слабые стороны. Так, в отдельные области выделились динамические методы, использующие элементы формальных, в частности тестирование на основе моделей (model-based testing, model driven testing) [1], использующее для построения тестов как формализацию некоторых свойств программного обеспечения, так и статический анализ кода. Примером средства тестирования на основе моделей является система Avalanche [2], называемая ниже «анализатор», автоматически создающая тесты, для которых осуществляется поиск уязвимостей по бинарному коду, и при обнаружении создающая входной файл, демонстрирующий ее проявление. Эта система позволяет осуществлять целенаправленный поиск уязвимостей по созданным критериям, что понижает временную сложность и повышает его эффективность. Avalanche работает под управлением ОС Linux и использует возможности динамического преобразования исполняемого кода, предоставляемые открытым фреймворком Valgrind [3, 4], а также возможности открытого решающего модуля STP, позволяющего проверять выполнимость расширенных булевых формул [5].

Представление программы

Для того чтобы было удобно рассматривать принцип работы анализатора, программу можно представлять в виде ориентированного дерева с помеченными вершинами, как показано на рис. 1.

Корень дерева — это вершина первого типа, которая является начальной только для одного ребра и считается точкой входа в программу. К вершинам второго типа относятся условные переходы, зависящие от внешних данных, составляющие множество J мощностью n, каждый элемент которого принимает значение 1, если условный переход совершен, и 0 — если нет. Вершины второго типа всегда являются начальными для пары ребер. В третий тип входят вершины, из которых не исходит ни одного ребра, соответствующие вариантам завершения программы. Если в программе существует возможность прийти в одну вершину двумя или более различными способами, то выполняется операция расклеивания, при которой вершина и идущие из нее пути будут продублированы для каждого входящего в нее ребра. Рёбра — это целостная часть кода без условных переходов, зависящих от внешних данных. Если тестируемое приложение не содержит бесконечных циклов, то дерево является конечным. Под трассой подразумевается цепь, соединяющая корень и лист, что на практике является собранной трассой для программы, запущенной на определенных входных данных. Множество всех входных данных обозначается через X и имеет мощность x. Множество входных данных, приводящих к проявлению уязвимостей, для которых у анализатора заданы критерии, — V размерности v, $V \subseteq X$. Таким образом, если V пусто, то программа не содержит уязвимостей тех типов, для которых составлены модели, и успешно пройдет верификацию.

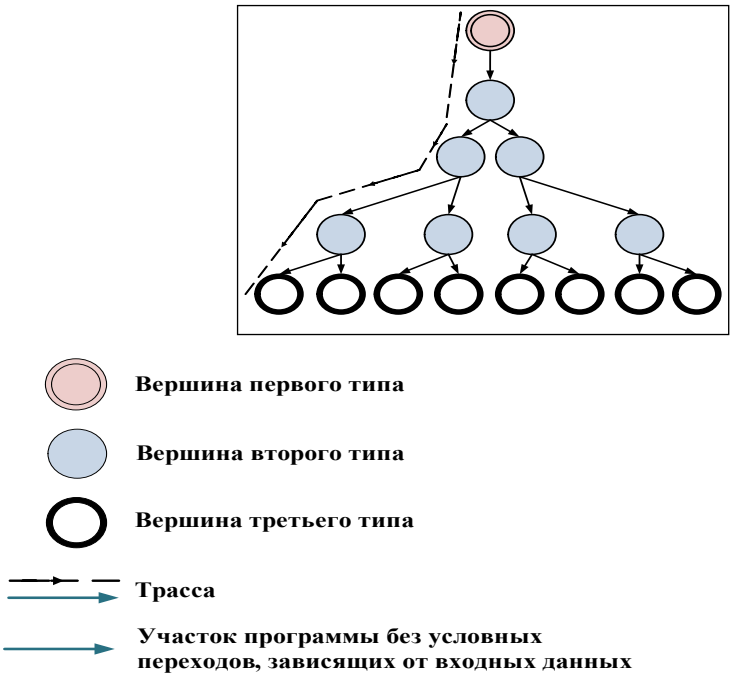

Рис. 1. Представление программы в виде дерева

Алгоритм работы анализатора

Общая схема работы заключается в последовательном запуске программы на различных входных данных и ее анализе, в случае обнаружения теста, расцененного как элемент множества V, анализатор сохранит его отдельно. На вход подается программа, запущенная на каких-либо входных данных, т. е. на произвольном элементе множества X. В результате получается одна трасса, по которой необходимо составить все остальные. Анализатор в ходе своей работы динамически выполняет ряд изоморфных преобразований исполняемого кода с помощью Valgrind, на основе которых формирует булево уравнение для каждой операции, зависящей от внешних данных, т. е. системных вызовов, чтения в и из регистров, арифметических и логических операций, условных переходов и т. п. С учетом описанного выше представления программы, чтобы вычислить возможность и значения для нового набора входных данных, уравнения собираются в системы уравнений следующим образом:

• в первую систему уравнений войдут все операции, принадлежащие ребру, где смежными вершинами являются корень дерева и первый условный переход, а значение, присвоенное этому переходу, инвертируется, т. е. если переход на трассе произошел, то переменная, хранящая это значение, станет равной нулю и наоборот;

• во вторую войдут, помимо всех операций первого ребра и неинвертированного значения первого перехода, все операции второго и инвертированное значение второго условного перехода и так до самого конца цепи.

Системы уравнений будут решены внутренними средствами STP. Для всех найденных решений в исходном входном файле будут заменены соответствующие биты, в результате чего будет сформирован новый файл. Если в программе на имеющейся трассе встретилось более одного элемента, принадлежащего J, то вариантов входных файлов будет несколько. В этом случае анализатор по умолчанию выбирает для нового запуска тот тест, который был сформирован из системы уравнений, содержащей наименьшее количество элементов J, иначе говоря, для условного перехода, наиболее близкого к корню дерева из рассматриваемых. Анализатор содержит возможность задавать первую и последнюю вершины для создания новых трасс, тем самым регулируя глубину, с которой и по которую будут инвертироваться элементы J.

Для программы, имеющей n условных переходов, зависящих от входных данных, и не содержащей бесконечных циклов, в противном случае сработает счетчик времени анализатора, и используемый тест выведется как элемент V, существует $n + 1$ трасс и $2n + 1$ ребер, которые теоретически можно обойти за конечное время. Чтобы совершить полный анализ исследуемого приложения по всем существующим для него критериям, необходимо протестировать на всех элементах множества X каждое ребро, при том что даже попытка проверки одной трассы на всем допустимом для нее подмножестве множества X, т. е. используемой на этой трассе части входных данных, является трудоемкой задачей. По этой причине возникает необходимость осуществлять целенаправленный поиск определенных типов уязвимостей, для которых возможно составить формальную модель требований.

Метод целенаправленного поиска уязвимостей, зависящих от входных данных

Целью метода является создание по единственному тесту и наложение на всю трассу таких условий для каждого критерия, проверка выполнимости которых давала бы однозначный ответ о соответствии данному критерию всей трассы без необходимости ее анализа для каждого элемента множества X отдельно. Пусть K_i — i-й критерий верификации для некоторого типа уязвимости, где $i \in [0,m]$, где m — количество критериев. Чтобы создать формальную модель требований для этого критерия, которая будет применяться для поиска в коде потенциально уязвимых мест, нужно выявить зависимость ее проявления от входных данных и составить одно или несколько уравнений, отвечающих следующим требованиям:

- в левой части находятся неизвестные переменные, отражающие параметры, зависящие от входных данных, отсутствие ограничений на определенные значения которых может привести к появлению уязвимости;
- правая часть содержит константы и переменные, определяемые через входные данные;
- если уравнение имеет хотя бы одно решение, значит, уязвимость существует и можно составить такие входные данные, которые будут считаться элементом множества V.

Анализатор отдельно проверяет каждую последовательность операций, которую регистрирует как потенциально опасную по определенному критерию с помощью существующей для него модели требований. Для этого он на основе всего участка данной трассы от точки входа до конца проверяемой последовательности операций строит и решает систему уравнений, в которую входят:

- все формулы, представляющие зависимость операций от внешних данных, включая неинвертированные условные переходы;
- уравнение или уравнения, составленные для проверяемой последовательности на основе соответствующей модели, отвечающие вышеописанным требованиям.

В тех случаях, когда анализатор находит решение такой системы уравнений, он выводит результат, расцененный как элемент V, и следующий запуск программы будет осуществлен на этом тесте. Если наличие уязвимости подтвердится, то анализатор сохраняет данный тест как демонстрирующий проявление найденной уязвимости, что исключает появление ошибок первого рода, т. е. ложных срабатываний.

Рассмотрим i-й критерий на конкретной трассе. Пусть каждая неизвестная переменная в левой части зависит от подмножества $X^i_y \subseteq X$ мощности x^i_y, где $y \in [1,Y]$, Y — количество неизвестных переменных в правой части уравнений, необходимых для проверки модели. Тогда поиск одной уязвимости этого типа по данному методу заменяет L_i запусков программы для одной трассы на разных входных данных:

$$L_i = \left| \bigcup_{y=\overline{1,Y}} \left(X_y^i\right) \right| - \left| \bigcap_{y=\overline{1,Y}} \left(X_y^i\right) \right| - 1$$

Вычитание единицы соответствует входным данным, на основе которых была составлена проверяемая трасса.

С учетом того, что для каждой трассы будут сразу составлены проверки по всем критериям для каждого ребра, входящего в нее, метод заменяет L_0 запусков:

$$L_0 = \sum_{i=1}^{m} a_i\, b_i\, L_i, \ ,$$

где индекс i соответствует i-му критерию для проверки и $i \in [1,m]$, $a_i \in [0,1]$ отражает факт присутствия хотя бы одной потенциально опасной последовательности операций, зарегистрированной i-й моделью, а b_i — количество мест в трассе, где встречается потенциальная уязвимость для i-го критерия.

На данный момент Avalanche содержит формальные модели проверки для 3 типов уязвимостей, определяемых описанным методом:

- деление на нуль;
- разыменование нулевого указателя;
- переполнение буфера на стеке для платформы x86 при условии зависимости читаемой длины от входных данных.

Во всех трех случаях условия для модели задаются одной неизвестной переменной. В первом и втором случаях проверяется возможность равенства переменной в левой части, т. е. делителя в первом случае и адреса чтения или записи — во втором, нулю. В третьем случае уравнение отражает возможность передачи большей длины читаемых данных, чем размер зарезервированного буфера, в правой части уравнения вычисляется разность между адресом дна стека для вызываемой функции и местоположением адреса возврата в этом стеке.

СПИСОК ЛИТЕРАТУРЫ:

1. *Broy M., Jonsson B., Katoen J.-P., Leucker M., Pretschner A.* (eds.) Model Based Testing of Reactive Systems. LNCS 3472, Springer, 2005.

2. *Исаев И., Сидоров Д.* Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах // Программирование. 2010. № 4.

3. Официальный сайт Valgrind. URL: www.valgrind.org.

4. *Nethercote N., Valgrind S. J.* A Framework for Heavyweight Dynamic Binary Instrumentation. National ICT Australia. OpenWorks LLP. Cambridge.

5. *Ganesh V., Dill D.* A decision procedure for bit-vectors and arrays. 2007.