Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа

[1] А.Ю. Герасимов <agerasimov@ispras.ru>
[2] Л.В. Круглов <kruglov@ispras.ru>
[1] Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1

Аннотация. Динамическое символьное исполнение – это хорошо известная техника, применяемая для решения различных задач анализа программ: генерация входных данных для увеличения тестового покрытия программы, для эксплуатации уязвимостей и т.д. В то же время значительное падение производительности при динамическом символьном исполнении также является хорошо известной, в общем случае NP-сложной проблемой в связи с экспоненциальным взрывом количества анализируемых путей и решением задачи SAT/SMT при разрешении предиката пути. Применение метода грубой силы при попытке анализа всех достижимых путей в программе, как правило, не имеет смысла в условиях жесткого ограничения времени решения поставленных задач. Поэтому применяются различные методы и эвристики для увеличения производительности анализа и сокращения анализируемого пространства. Мы представляем подход совмещения статического анализа исполняемого кода, основанного на использовании библиотеки binutils, и метода динамического символьного исполнения, основанного на инструменте итеративного динамического анализа Avalanche, для направленной генерации входных данных программы с целью достижения заранее определенной функции в программе. На первом шаге предлагаемого подхода строится усеченный граф вызовов программы, который содержит только те функции, вызов которых в конечном счете приводит к вызову заранее определенной функции. Далее мы дополняем граф вызовов графом потока управления внутри функций, включенных в усеченный граф вызовов. С использованием усеченного графа потока управления программы, который содержит только вызовы и условные переходы, приводящие в конечном итоге к вызову заранее определенной функции, вычисляется метрика наиболее перспективного пути для проведения дальнейшего анализа. Предложенный подход позволил значительно (до двенадцати раз для некоторых реальных программ) сократить время достижения заранее определенной функции по сравнению с методом грубой силы.

Ключевые слова: статический анализ; динамический анализ; генерация входных данных.

DOI: 10.15514/ISPRAS-2016-28(5)-10

Для цитирования: А.Ю. Герасимов, Л.В. Круглов. Вычисление входных данных для достижения определенной функции в программе методом итеративного динамического анализа. Труды ИСП РАН, том 28, вып. 5, 2016, стр. 159-174. DOI: 10.15514/ISPRAS-2016-28(5)-10

1. Введение

В связи с активным развитием области разработки программного обеспечения, применяемого в таких критических областях жизни и деятельности человека, как автоматическое управление транспортными средствами и управление опасными производствами, медицине и военной технике, то есть в областях, где вмешательство оператора вычислительного устройства для исправления ошибочной ситуации минимально или сведено к нулю, — на первый план выходит задача обеспечения высокого качества программ с точки зрения отсутствия критических ошибок времени исполнения и уязвимостей, позволяющих злоумышленнику получить контроль над исполнением программы. Для обеспечения качества программного обеспечения могут применяться различные подходы, такие как тестирование [1] и верификация моделей [2, 3] — подходы, которые направлены на выяснение соответствия разработанной программы спецификации требований или модели, составленной на стадии проектирования. С другой стороны, данные подходы не всегда способны обеспечить проверку отсутствия ошибок или нежелательного поведения программы, которое не может быть заложено в функциональные требования к программе или в модель. Для решения данной задачи применяются методы анализа программ, направленные на автоматическое выявление нежелательного поведения или критических ошибок. При этом сами методы анализа программ либо недостаточно точны и выдают большое количество ложных предупреждений об ошибках (статический анализ), либо обладают высокой вычислительной сложностью, что не позволяет проводить полный анализ программы за приемлемое время (динамический анализ).

Нередко в процессе анализа программ может быть использовано несколько инструментов, проводящих разные виды анализа программ, а также может быть вовлечен аналитик, который использует инструменты анализа программ для обнаружения ошибок и уязвимостей в программе. Например, аналитику может быть интересно проанализировать программу на наличие возможности эксплуатации уязвимости, связанной с использованием форматной строки [4] или разработчику требуется подтвердить наличие дефекта в программе, найденного инструментами статического анализа исходного кода программы. В первом случае аналитик указывает функцию, достижимость которой

необходимо проверить и предоставить информацию о том, как входные данные программы преобразуются в аргументы функции, во втором случае информация о функции, в которой потенциально реализуется дефект или уязвимость, передается инструментом статического анализа программы инструменту динамического анализа программы в виде трассы с предусловиями реализации ошибочной ситуации. В обоих случаях возникает задача вычисления входных данных, при получении которых в программе происходит вызов определенной функции, а также выявления зависимости между входными данными программы и аргументами интересующей функции. В данной статье описывается подход, направленный на совмещение методов статического и динамического анализа программ с целью повышения производительности динамического анализа программ при решении задачи вычисления входных данных для достижения указанной функции программы.

Далее статья построена следующим образом. Во втором разделе производится обзор методов анализа программ и ограничений, которые им присущи, а также даётся краткий обзор применения результатов статического анализа для упрощения проведения динамического анализа программ. В третьем разделе подробно рассматривается подход к применению результатов статического анализа для повышения производительности динамического анализа программ, в четвертом разделе приводятся результаты экспериментов, подтверждающих практическую применимость предложенного подхода, в заключении рассматриваются возможные приложения для применения описанного подхода, а также направления дальнейших исследований.

2. Методы анализа программ и их ограничения

Методы анализа программ условно можно разбить на две группы: статические, то есть такие, при которых программа анализируется без запуска на выполнение, и динамические, при которых программа анализируется в процессе (online-анализ) или по результатам (offline-анализ) выполнения.

Статический анализ программ может выполняться как по исходному коду, так и по исполняемому коду программы. Методы статического анализа программ строят модель программы, являющейся абстракцией исходного или бинарного кода программы, и производят анализ с использованием данной модели. Как правило, методы статического анализа обладают возможностью масштабирования путём независимого анализа отдельных функциональных блоков в программе и составлению автоматических аннотаций для каждого блока по набору критериев, описывающих его поведение, но при этом характеризуются возможностью выдачи ложно-положительных предупреждений об ошибочной ситуации в связи с тем, что модель может недостаточно полно описывать поведение программы, а проверка ошибочной ситуации может проводиться не на полном пути выполнения от точки входа в программу до реализации ошибочной ситуации, а только на некотором

подпути в программе от точки инициализации ошибочной ситуации до точки реализации ошибки.

В свою очередь, методы динамического анализа программ обладают высокой точностью обнаружения ошибочных ситуаций и, как правило, возможностью их воспроизведения в связи с тем, что ошибка регистрируется в момент, когда программа уже выполнила недопустимую операцию, сохранена трасса выполнения от точки входа в программу и имеются точные входные данные, которые позволяют воспроизвести ошибочное поведение. С другой стороны, методы динамического анализа крайне плохо масштабируются. Это связанно с тем, что для проведения анализа программы по альтернативному пути выполнения необходимо тем или иным способом вычислить входные данные, например методами фаззинга [5] или методами символьного выполнения [6, 7, 8] с решением формулы ограничений пути, и произвести полный или частичный запуск программы на выполнение с использованием вычисленных входных данных.

Как отмечалось в работах [9, 6], при применении методов итеративного динамического анализа, построенных на принципе символьного выполнения программы, в процессе анализа программы достаточно быстро возникает проблема экспоненциального роста количества путей, которые необходимо проанализировать. Для каждого нового пути в программе необходимо построить и вычислить одну или несколько формул ограничений пути при помощи решателя, например [10, 11], что в свою очередь согласно [6, 12] в зависимости от реализации анализируемой программы может составить до 99% времени анализа, поскольку для решения этой задачи не известно полиномиальных алгоритмов [13, 14], то на каждом запуске программы необходимо решить в общем случае NP-сложную задачу [15] для решения формулы ограничений пути или определения несовместности ограничений пути. Ранее рассматривались подходы к увеличению производительности итеративного динамического анализа путем применения параллельных и распределенных вычислений [16] и кэширования результатов вычисления булевых формул [17], но в общем случае кардинально увеличить производительность итеративного динамического анализа программ при сохранении полноты анализа на данный момент не представляется возможным в связи с экспоненциальным ростом количества путей, которые необходимо проанализировать в программе.

В связи с выше сказанным, проведение полного анализа всех путей в программе за приемлемое время не представляется возможным и крайне важным становится решение задачи выбора для анализа наиболее перспективных путей в программе. Как правило, выбор пути для дальнейшего анализа программы происходит на основе некоторой эвристической оценки перспективности пути, в связи с этим возникает задача построения качественной метрики выбора следующего пути для анализа в зависимости от целей анализа. В настоящей статье предлагается подход к определению

лучшего пути до указанной тем или иным способом функции программы на очередном шаге итеративного динамического анализа на основе результатов предварительного статического анализа исполняемого кода программы.

2.1 Различные подходы к выбору пути для проведения динамического анализа

Предлагаемый нами подход реализован в рамках инструмента Avalanche [6] путем разработки дополнительного модуля вычисления метрики наилучшего пути. В работе, описывающей инструмент Avalanche, рассматривается подход к реализации итеративного динамического анализа с применением метрики перспективности пути по критерию максимально быстрого прироста покрытия базовых блоков программы. Если множество проанализированных базовых блоков программы на *i*-той итерации анализа — $\boldsymbol{BBSetAnalyzed_i}$, количество базовых блоков программы, которое будет проанализировано на пути $\boldsymbol{\pi_n}$ даст покрытие $\boldsymbol{BBSet_{\pi n}}$, а количество базовых блоков, которые будут проанализированы на пути $\boldsymbol{\pi_m}$, даст покрытие $\boldsymbol{BBSet_{\pi m}}$, то в первую очередь будет проанализирован путь, для которого мощность разности множества проанализированных базовых блоков на предыдущих итерациях анализа и множества базовых блоков пути будет больше или равна:

$$\pi_{i+1} = \begin{cases} \pi_n, |BBSetAnalyzed_i \setminus BBSet_{\pi n}| \geqslant |BBSetAnalyzed_i \setminus BBSet_{\pi m}| \\ \pi_m, |BBSetAnalyzed_i \setminus BBSet_{\pi n}| < |BBSetAnalyzed_i \setminus BBSet_{\pi m}| \end{cases}$$

В другой работе [18], посвященной инструменту автоматической генерации кода эксплуатации уязвимостей в программе, рассматриваются две эвристики выбора пути для поиска программных ошибок с целью их последующей эксплуатации. Эвристика **Уязвимый-путь-сначала (Buggy-Path-First)** предполагает поиск на пути выполнения операции с ошибкой на единицу при вычисления диапазона в процессе работы с буферами в памяти (off-by-one error), которая сама по себе может и не быть эксплуатируема, однако позволяет предположить, что программист недостаточно внимательно следит за границами выделенных буферов, что может привести к нахождению ошибки переполнения буфера далее на пути, которую можно будет проэксплуатировать. Эвристика **Разрежения циклов (Loop Exhaustion)** применяется в случае, если индуктивная переменная цикла зависит от входных данных. Тогда инструмент не пытается выполнить цикл на всём пространстве значений индуктивной переменной, а вычисляет максимально возможное значение переменной и анализирует выполнение цикла сначала на максимально возможном количестве итераций с учётом ограничений пути на значение индуктивной переменной в надежде, что вычисления, требующие большего количества итераций, приведут к ошибкам переполнения буфера с большей вероятностью.

3. Описание предлагаемого подхода

Итеративный динамический анализ осуществляет последовательный обход дерева путей программы, зависящих от условных переходов, и осуществляет поиск ошибок на каждом исполненном пути. Для этого в процессе исполнения программы производится отслеживание операций над входными данными и данными, являющимися результатами этих операций. Такие данные называются помеченными (tainted). Необходимо отслеживать все операции чтения из внешних источников и записи прочитанных значений в память и регистры, которые далее также необходимо считать источниками помеченных данных до тех пор, пока они не будут перезаписаны непомеченными значениями. По результатам запуска программы производится сбор трассы выполненных операций и по собранной трассе выполненных операций строится система ограничений, описывающая ограничения на значения переменных и регистров, содержащих помеченные данные. Так как каждый путь определяется набором направлений условных переходов, то в соответствующую путям систему уравнений также необходимо включать условия переходов, зависящие от помеченных данных.

Для получения на основе исходной системы уравнений новых систем, возможно соответствующих новым путям исполнения, достаточно инвертировать условия каждого условного перехода, зависящего от входных данных. При наличии в системе N условий переходов будет получено N новых систем: в i-ой системе будет инвертировано условие i-го условного перехода, условия до i-го будут оставлены без изменений, а условия после i-го отброшены. Решив полученные N систем с помощью решателя формул, можно построить входные данные в случае успешной разрешимости формулы, которые соответствуют новым путям в исследуемой программе.

На каждом шаге итеративного динамического анализа исследуется один путь исполнения, строится описывающая его система уравнений и вычисляются новые наборы входных данных, соответствующие новым путям исполнения. Выбор единственного пути для анализа на следующей итерации осуществляется на основе вычисления метрик (эвристических оценок) и происходит при запуске программы на новых входных данных, вычисленных с учётом метрики наилучшего пути. Пути, отброшенные на текущем шаге анализа, сохраняются для анализа на последующих шагах, когда метрика перспективности отброшенного пути превысит метрику перспективности оставшихся путей на очередном шаге анализа. Это позволяет проанализировать программу на всех достижимых путях с помощью итеративного динамического анализа, но просматривать в первую очередь наиболее перспективные пути, получая различные наборы входных данных, соответствующие различным путям, на которых происходит вызов заданной функции.

В настоящей работе мы предлагаем проводить предварительный статический анализ исполняемого кода программы до запуска итеративного динамического

анализа. В результате проведения статического анализа программы строится граф вызовов и граф условных переходов, которые позволяют вычислить множество путей достижения указанной функции, наиболее интересной с точки зрения целей анализа. Итеративный динамический анализ в процессе построения формулы ограничений пути использует информацию, извлеченную из графа вызовов и графа переходов, полученных в результате статического анализа, и формирует набор утверждений для вычисления входных данных программы таким образом, чтобы на каждой следующей итерации анализа максимально быстро приближаться к указанной функции вплоть до достижения точки её вызова на очередном шаге анализа.

3.1 Построение путей до функции

Под графом вызовов будем понимать ориентированный циклический граф, узлы которого представляют собой различные функции исследуемой программы, а ребра — вызовы функций. Минимальное расстояние между узлами в графе вызовов определим как минимальное количество ребер на пути между данными узлами.

Для вычисления предлагаемой метрики достаточно иметь неполный граф вызовов — граф, содержащий только те узлы, из которых достижима заданная функция. Такой граф может быть эффективно построен обратным проходом, начиная с указанной функции. Сначала необходимо найти все точки вызова выбранной функции, соответствующие некоторым вызывающим функциям. Затем для каждой вызывающей функции необходимо найти все точки её вызова и соответствующие им вызывающие функции. И так до тех пор, пока не будет достигнута некоторая конечная функция или набор функций, которые явно не вызываются ни из каких других функций программы. Это может быть как точка входа в программу, так и функция, вызываемая через указатель (виртуальный вызов), или недостижимая функция. Далее под графом путей до выбранной функции будем понимать построенный таким образом неполный граф вызовов программы.

Расширим граф вызовов до графа переходов: дополним каждый узел в графе вызовов (функцию) информацией о путях исполнения внутри функции. Для этого требуется построить неполный граф потока управления каждой функции, состоящий из базовых блоков, лежащих на путях от входа в функцию до точек вызова других функций из неполного графа вызовов, ведущих к указанной функции. Определим расстояние в графе переходов аналогично — как количество ребер между узлами в расширенном графе переходов.

Вычисление расстояния от некоторой вершины графа переходов до выбранной функции удобно проводить для графа, в котором все ребра заменены на обратные и имеют единичный вес, с помощью алгоритма поиска кратчайшего расстояния от указанной вершины до всех остальных, например с помощью алгоритма Дейкстры.

Описанные графы вызовов и переходов строятся по исполняемому коду программы до начала итеративного динамического анализа, так как в этом случае будут учтены оптимизации компилятора, и он будет точно соответствовать путям исполнения исследуемой программы. При реализации предложенного подхода построение графов производится на основе анализа ассемблерного кода, полученного с помощью утилиты objdump пакета инструментов для работы с кодом объектных файлов binutils [19].

3.2 Метрика выбора наиболее перспективного пути

В качестве метрики для эффективного достижения указанной функции в анализируемой программе предлагается использовать расстояния в графе вызовов и условных переходов от точки на пути исполнения до точки входа в интересующую функцию. Минимальное расстояние будет соответствовать лучшему пути для анализа на следующей итерации.

Проиллюстрируем работу алгоритма выбора следующего пути для анализа на примере (Рис. 1). Допустим, что в процессе выполнения очередной итерации анализа (путь выполнения отмечен белыми узлами графа переходов) обнаружено несколько возможных путей для дальнейшего анализа, которые можно получить путём инвертирования условий в условных переходах 1, 2, 3 и 4.

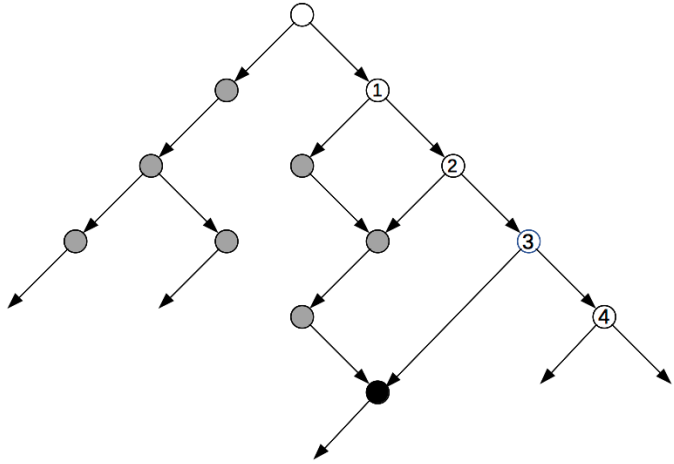

Рис. 1. Вычисление метрики выбора наилучшего пути для анализа

Fig. 1. Metric calculation for best path selection during analysis

Для каждого из возможных путей выполнения проводится вычисление входных данных и запускается легковесный проход, который отслеживает только посещения базовых блоков, присутствующих на пути выполнения. Для каждого базового блока по графу потока управления полученному на этапе

статического анализа исполняемого файла вычисляется расстояние до указанной функции. Тот путь, на котором вычисленное расстояние будет наименьшим, выбирается как наиболее перспективный для анализа на следующей итерации. Успешное достижение указанной функции фиксируется, когда значение метрики расстояния становится равным нулю. В представленном примере:

- при инвертировании условия в узле 1 до указанной функции необходимо будет пройти 3 условных перехода или вызова (путь 1). Значение метрики равно 3;
- при инвертировании условия в узле 2 до указанной функции необходимо будет пройти 2 условных перехода или вызова (путь 2). Значение метрики равно 2;
- при инвертировании условия в узле 3 до указанной функции останется 0 условных переходов или вызовов (путь 3). Значение метрики равно 0;
- при инвертировании условия в узле 4 мы никогда не достигнем указанной функции. Анализ для данного пути проводится в последнюю очередь, так как в процессе статического анализа путь, ведущий к указанной функции, не найден, но может существовать путь, зависящий от состояния времени выполнения программы, например при наличии перехода по вычисляемому адресу.

Соответственно, для последующего анализа будет выбран путь 3, как путь с наименьшей метрикой приближения к указанной функции.

4. Результаты экспериментов

Полученное решение было протестировано на ряде проектов с открытым исходным кодом, в которые специально был привнесен дефект разыменования нулевого указателя:

- `qtdump. libquicktime` – библиотека чтения и записи файлов в форматах QuickTime/AVI/MP4; утилита `qtdump` отображает разобранное содержимое файла;
- `cjpeg. libjpeg7` – библиотека для обработки файлов в формате JPEG; утилита `cjpeg` конвертирует файлы в формат JPEG;
- `swfdump. swftools` – пакет программ для работы с SWF-файлами; утилита `swfdump` отображает подробную информацию о файле и дизассемблированный код;
- `gif2rgb. giflib` – библиотека для чтения и записи файлов в формате GIF; утилита `gif2rgb` конвертирует GIF-изображения в 24-битные RGB изображения;

- `tiffdump`. `libtiff` – библиотека для обработки файлов в формате TIFF; утилита `tiffdump` выводит подробную информацию о TIFF-файле;
- `mpeg3cat`. `Libmpeg3` — библиотека для редактирования MPEG-файлов. `mpeg3cat` — утилита для конкатенации и разделения MPEG-потоков;
- `xmllint`. `libxml2` — библиотека для работы с файлами в формате XML. Утилита `xmllint` позволяет производить разбор XML-файлов.

В таблице 1 представлены результаты измерения времени достижения одного и того же дефекта и количестве потребовавшихся для этого итераций. Сравниваются исходная метрика покрытия базовых блоков и реализованная метрика минимального расстояния до указанной функции, использующая граф вызовов и использующая граф переходов с упрощенным графом потока управления для каждого узла.

Табл. 1. Время и количество итераций анализа до обнаружения дефекта

Table 1. Time and iterations count before reaching defect

Проект	Avalanche		с графом вызовов		с графом путей	
	Итераций	Время	Итераций	Время	Итераций	Время
qtdump	73	1:56:07	60	17:15	60	17:26
	38	19:06	33	2:47	33	2:26
cjpeg	116	6:12	98	4:48	93	4:43
	30	6:08	14	0:33	5	0:27
swfdump	5	0:10	3	0:10	3	0:10
	12	0:21	6	0:18	4	0:13
gif2rgb	10	1:13	432	7:34:13	5	0:24
tiffdump	16	0:22	3	0:12	3	0:14
mpeg3cat	24	4:32:24	421	53:01	15	8:28
xmllint	Не найдено за 10 часов		Не найдено за 10 часов		325	46:04

По результатам экспериментов можно сделать следующие выводы:

- количество итераций, потребовавшихся для достижения дефекта, в общем случае незначительно меньше для новой метрики;
- анализ проекта `gif2rgb` показал, что новая метрика, использующая только граф вызовов, в некоторых случаях значительно замедляет достижение дефекта — более чем в 420 раз для данного эксперимента. Это связано с тем, что, с одной стороны, содержащая дефект функция расположена неглубоко в стеке вызовов и, с другой стороны, путь от точки входа в функцию до дефекта содержит большое количество условных переходов. В этом случае при использовании графа вызовов большое количество путей внутри функции будет иметь одинаковую эвристическую оценку вне зависимости от близости дефекта, что не позволяет выбирать для анализа пути, приближающиеся к дефекту. Однако учёт условных переходов при использовании метрики с графом переходов позволяет получить выигрыш по времени в 3 раза для того же дефекта по сравнению с исходной метрикой пути, рассчитанной по приросту покрытия базовых блоков.

Реализованная метрика позволяет получить значительный выигрыш по времени нахождения дефекта по сравнению с исходным инструментом Avalanche в случае, если известна функция, в которой наиболее вероятно обнаружение дефекта. Наилучшие результаты были получены для проектов `qtdump` (ускорение в 6 раз), `cjpeg` (ускорение в 12 раз).

5. Заключение

Применение статического анализа исполняемого или исходного кода программ, как предварительного шага перед проведением динамического анализа программ не является новым подходом. В работах [20, 21] рассматривается применение предварительного статического анализа графа вызовов программы для обнаружения точек входа в программах для ОС Android и построения путей для последующего проведения динамического символьного выполнения программы с целью обнаружения ошибок времени исполнения. Особенностью программ для ОС Android является наличие нескольких точек входа в потоке управления программы, связанное с моделью обработки сообщений операционной системы как от элементов управления пользовательского интерфейса, так и от других системных событий, по этому для проведения динамического анализа крайне важно определить возможные точки входа в поток управления программы.

Представленный в настоящей статье подход к решению задачи повышения производительности итеративного динамического анализа может быть применен для увеличения производительности решения целого класса задач в области обеспечения качества программного обеспечения. Дополнительно, представленный подход является частью решения задачи подтверждения

ошибок и уязвимостей (дефектов) в программном обеспечении, обнаруженных методами статического анализа исходного кода.

Дельнейшие исследования в области совмещения статического и динамического анализа программ могут быть направлены на решение задачи подтверждения дефектов, найденных методами статического анализа программ, генерации кода эксплуатации подтвержденных дефектов и улучшение производительности алгоритмов генерации данных для фаззинга.

Список литературы

[1]. Myers G. J., Badgett T., Sandler C. The Art of Software Testing. Third Edition. John Wiley & Sons, Inc., Hoboken, New Jersey, 2012, 240 p.

[2]. Ю.Г. Карпов. MODEL CHECKING. Верификация параллельных и распределенных систем. СПб:БХВ-Петербург. 2010

[3]. Кларк Э.М., Грамберг О., Пелед Д. Верификация моделей программ: Model Checking, М.:МЦНМО, 2002

[4]. Kyung-Suk Lhee, S.J. Chapin. Buffer Overflow and Format String Overflow Vulnerabilities. Software-Practice & Experience — Special Issue: Security Software, Volume 33 Issue 5, 25 April 2003, pp. 423-460

[5]. Ari Takanen, Jared D. Demott, Charles Miller. Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008

[6]. И.К. Исаев, Д.В. Сидоров. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование №4, 2010 г.

[7]. Cadar C., Dunbar D., Engler. D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), December 8-10, 2008, San Diego, CA, USA

[8]. Chipounov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) Special issue: Best papers of ASPLOS, February 2012.

[9]. В.В. Каушан, Ю.В. Маркин, В.А. Падарян, А.Ю. Тихонов. Методы поиска ошибок в бинарном коде. Препринты Института системного программирования РАН, препринт 24, 2013 г.

[10]. L. de Moura, N. Bjørner. Z3: an Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008

[11]. Ganesh V.. Decision Procedures for Bit-Vectors, Arrays and Integers. (PhD. Thesis) Computer Science Department, Stanford University, Stanford, CA, U.S., Sept 2007

[12]. Исаев И.К., Сидоров Д.В., Герасимов А.Ю., Ермаков М.К. Avalanche: Применение динамического анализа для автоматического обнаружения ошибок в программах использующих сетевые сокеты. Труды ИСП РАН, том 21, 2011 г., стр. 55-70

[13]. I. Johnson. Formal Verification with SMT Solvers: Why and How. ACL2 Theorem Proving Seminar at the University of Texas, Autin, 2009

[14]. Новикова Н.М. Основы оптимизации. Москва. 1998. 17–22 с.

[15]. S.A. Cook. The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing, New York, USA, NY,1971, pp 151-158

[16]. М.К. Ермаков, А.Ю. Герасимов. Avalanche: применение параллельного и распределенного динамического анализа программ для ускорения поиска дефектов и уязвимостей. Труды ИСП РАН, том 25, 2013 г., стр. 29-38. DOI: 10.15514/ISPRAS-2013-25-2

[17]. С.П. Вартанов, Д.В. Сидоров. Оптимизация задачи проверки выполнимости булевских ограничений при помощи кэширования промежуточных результатов. Труды ИСП РАН, том 22, 2012 г., стр. 281-292. DOI: 10.15514/ISPRAS-2012-22-16

[18]. Thanassis Agerinos, Sang Kil Cha, Brent Lim Tze Hao, David Broomley. AEG: Automatic Exploit Generation. Proceedings of the Network and Distributed Security Symposyum, Carnegie Mellon University, 2011

[19]. GNU Binutils [HTML] (http://www.gnu.org/software/binutils/)

[20]. Schütte J., Fedler R., Titze D. ConDroid: Targeted Dynamic Analysis of Android Applications. Advanced Information Networking and Applications (AINA), IEEE, Gwangui, 2105, DOI:10.1109/AINA.2015.238

[21]. Wong M., Lie D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS), Feb 2016.

Input data generation for reaching specific function in program by iterative dynamic analysis

[1] *A.Y. Gerasimov <agerasimov@ispras.ru>*
[2] *L.V. Kruglov <kruglov@ispras.ru>*
[1] *Institute for System Programming of the Russian Academy of Sciences, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Lomonosov Moscow State University, GSP-1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. Dynamic symbolic execution is a well-known technique used for different tasks of program analysis: input generation for increasing test coverage for program, inputs of death generation, exploit generation and etc. But huge time costs of program analysis during dynamic symbolic execution for any real-life program is a well-known problem caused by path explosion and necessity of path constraint solving for every path with different SAT/SMT techniques which is a NP-complete task in general case. Brute force analysis of every path in program has limited practical sense for time limited analysis; instead different techniques and heuristics are used to improve analysis performance and reduce space of analysis for specific needs of analyst or while solving specific problem under analysis. We present our approach which combines static analysis of program binary code based on binutils library with dynamic symbolic execution tool based on Avalanche – an iterative dynamic analysis tool to perform targeted input data generation for reaching specific function in the program. As the first step of our algorithm we extract reduced program call graph which contains only calls to functions which ends with the function of interest, then we amplify this call graph with control flow graph inside of functions included into reduced call graph. Using the reduced control-flow graph of program which contain only calls and conditional jumps directions which lead to the function of interest we built the metric of best next analysis direction. This approach allows us to significantly (up to twelve times for some real world programs) reduce the time of reaching function of interest comparatively to brute force program paths analysis with inversion of every conditional jump at the execution path dependent on tainted data.

Keywords: dynamic program analysis; static program analysis; directed analysis; input data generation.

DOI: 10.15514/ISPRAS-2016-28(5)-10

For citation: A.Y. Gerasimov, L.V. Kruglov. Input data generation for reaching specific function in program by iterative dynamic analysis method. Trudy ISP RAN/Proc. ISP RAS, vol. 28, issue 5, 2016, pp. 159-174 (in Russian). DOI: 10.15514/ISPRAS-2016-28(5)-10

References

[1]. Myers G. J., Badgett T., Sandler C. The Art of Software Testing. Third Edition. John Wiley & Sons, Inc., Hoboken, New Jersey, 2012, 240 p.

[2]. Ju.G. Karpov. MODEL CHECKING. Verification of parralel and distributed systems. SPb:BHV-Peterburg. 2010 (in Russian)

[3]. Klark Je.M., Gramberg O., Peled D. Verification of program models: Model Checking, M.:MCNMO, 2002 (in Russian)

[4]. Kyung-Suk Lhee, S.J. Chapin. Buffer Overflow and Format String Overflow Vulnerabilities. Software-Practice & Experience — Special Issue: Security Software, Volume 33 Issue 5, 25 April 2003, pp. 423-460

[5]. Ari Takanen, Jared D. Demott, Charles Miller. Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008

[6]. I.K. Isaev, D.V. Sidorov. Application of dynamic analysis for generating input data exposing critical errors and vulnerabilities in programs. Programmirovanie №4, 2010 g. (in Russian)

[7]. Cadar C., Dunbar D., Engler. D. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs USENIX Symposium on Operating Systems Design and Implementation (OSDI 2008), December 8-10, 2008, San Diego, CA, USA

[8]. Chipounov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) Special issue: Best papers of ASPLOS, February 2012.

[9]. V.V. Kaushan, Ju.V. Markin, V.A. Padarjan, A.Ju. Tihonov. Methods of finding errors in binary code. ISP RAS preprints, Preprint 24, 2013. (in Russian)

[10]. L. de Moura, N. Bjørner. Z3: an Efficient SMT Solver. Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008

[11]. Ganesh V. Decision Procedures for Bit-Vectors, Arrays and Integers. (PhD. Thesis) Computer Science Department, Stanford University, Stanford, CA, U.S., Sept 2007

[12]. Isaev I.K., Sidorov D.V., Gerasimov A.Ju., Ermakov M.K. Avalanche: application of dynamic analysis for automatic error detection in programs using network sockets. Trudy ISP RAN/Proc. ISP RAS, vol 21, 2011. (in Russian)

[13]. I. Johnson. Formal Verification with SMT Solvers: Why and How. ACL2 Theorem Proving Seminar at the University of Texas, Autin, 2009

[14]. Novikova N.M. Optimization basics. Moskva. 1998. pp. 17–22. (in Russian)

[15]. S.A. Cook. The complexity of theorem-proving procedures. Proceedings of the third annual ACM symposium on Theory of computing, New York, USA, NY,1971, pp 151-158

[16]. M.K. Ermakov, A.Y. Gerasimov. [Avalanche: adaptation of parallel and distributed computing for dynamic analysis to improve performance of defect detection]. Trudy ISP RAN/Proc. ISP RAS, vol. 25, 2013, pp. 29-38 (in Russian).

[17]. S.P. Vartanov, D.V. Sidorov. [Optimization of Boolean satisfiability solver by caching intermediate results]. Trudy ISP RAN/Proc. ISP RAS, vol. 22, 2012, pp. 281-292 (in Russian).

[18]. Thanassis Agerinos, Sang Kil Cha, Brent Lim Tze Hao, David Broomley. AEG: Automatic Exploit Generation. Proceedings of the Network and Distributed Security Symposyum, Carnegie Mellon University, 2011

[19]. GNU Binutils [HTML] (http://www.gnu.org/software/binutils/)

[20]. Schütte J., Fedler R., Titze D. ConDroid: Targeted Dynamic Analysis of Android Applications. Advanced Information Networking and Applications (AINA), IEEE, Gwangui, 2105, DOI:10.1109/AINA.2015.238

[21]. Wong M., Lie D. IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware. In Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS), Feb 2016.