Подход к определению достижимости программных дефектов, обнаруженных методом статического анализа, при помощи динамического символьного исполнения[1]

А.Ю. Герасимов <agerasimov@ispras.ru>
Л.В. Круглов <kruglov@ispras.ru>
М.К. Ермаков <mermakov@ispras.ru>
С.П. Вартанов <svartanov@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. Среди методов анализа программ на наличие дефектов выделяют методы статического и динамического анализа. В данной статье мы предлагаем комбинированный подход, заключающийся в применении динамического символьного исполнения для определения достижимости дефектов, найденных при помощи статического анализа. Предлагаемый подход является развитием ранее предложенного подхода определения достижимости определенной инструкции в программе методами динамического символьного исполнения, примененном последовательно для нескольких точек в программе, включающих точки инициализации дефекта, условные переходы в трассе дефекта и точку реализации дефекта. С начала производится статический анализ исполняемого кода программы с целью выделения путей исполнения, которые приводят к точке инициализации дефекта. Далее производится вычисление входных данных, приводящих к точке инициализации дефекта методом динамического символьного исполнения и прохождения базовых блоков, лежащих на трассе дефекта, включая точку реализации дефекта. Выбор наиболее перспективного пути для исполнения программы производится при помощи метрики минимального расстояния от пути исполнения на предыдущей итерации до следующей точки на трассе дефекта. Метрика вычисляется на основе путей в графе вызовов в программе, расширенного графом потока управления функций на путях исполнения, приводящих к реализации дефектов. Предлагаемый подход был проверен на нескольких программах с открытым исходным кодом из комплекта утилит командной строки операционной системы Debian Linux. В результате экспериментальной проверки было подтверждена возможность применения данного подхода к классификации дефектов, найденных при

[1] Исследование проводится в рамках научно-исследовательских работ Института системного программирования им. В.П. Иванникова РАН в 2014-2017 годах

помощи статического анализа программ. Также, были обнаружены некоторые ограничения, препятствующие внедрению данного подхода в промышленные инструменты анализа. Одним из направлений дальнейших исследований может быть выбрано исследование подходов к снятию этих ограничений.

Ключевые слова: статический анализ программ; динамический анализ программ.

DOI: 10.15514/ISPRAS-2017-29(5)-7

Для цитирования: Герасимов А.Ю., Круглов Л.В., Ермаков М.К., Вартанов С.П. Подход определения достижимости программных дефектов, обнаруженных методом статического анализа программ, при помощи динамического анализа. Труды ИСП РАН, том 29, вып. 5, 2017 г., стр. 111-134. DOI: 10.15514/ISPRAS-2017-29(5)-7

1. Введение

Сегодня постоянно растёт сложность разрабатываемого программного обеспечения, в связи с чем задача автоматического обнаружения и воспроизведения программных дефектов становится весьма актуальной. По одной из возможных классификаций методов анализа программ все подходы можно разделить на две большие группы: методы статического анализа и методы динамического анализа.

Статический анализ производится без запуска программы на исполнение и как правило по некоторой семантической модели программы, построенной по исходному или исполняемому коду. Существует несколько приложений статического анализа программ, таких как сбор метрик [1] и построение выводов о качестве программного обеспечения на их основе, генерация тестовых сценариев [2]. Так или иначе методы статического анализа программ предназначены для обеспечения качества программных продуктов, в связи с чем методы обнаружения дефектов и уязвимостей безопасности в программах получили наиболее широкое распространение в индустрии программной инженерии [3, 4, 5]. Стоит отметить, что методы статического анализа недостаточно точны в обнаружении ошибок [6, 7], и связано это в первую очередь с обеспечением требований масштабируемости и производительности анализа [8, 9].

Динамический анализ производится либо в процессе исполнения программы (online-анализ) [10, 11, 12], либо после завершения работы программы (offline или post-mortem анализ) [13]. Среди методов динамического анализа программ стоит выделить динамическое символьное исполнение [14], также известное как конкретно-символьное (concolic – concrete and symbolic) [15], которое совмещает реальное исполнение программы с символьным исполнением [16]. Динамическое символьное исполнение позволяет применять техники исследования путей программы [17] и, добавляя предикаты безопасности к ограничениям пути (path constraint), проверять потенциально опасные операции на наличие реальных ошибок в программах. Динамическое символьное исполнения применяется для таких задач как генерация тестовых покрытий,

анализ утечек критических данных в программе, автоматическая генерация фильтров для входных данных с целью предотвращения эксплуатации уязвимостей в программах, обнаружение ошибок и уязвимостей в программах [18]. Наиболее серьезным ограничением для применения методов чистого динамического символьного исполнения для анализа программ является проблема экспоненциального взрыва количества путей для анализа (path explosion). Каждый условный переход в программе, зависящий от входных данных, потенциально увеличивает количество путей для анализа в два раза, что в конечном итоге приводит к резкому росту количества путей для анализа и невозможности исчерпывающего (exhaustive) анализа реальных программ.

В связи с указанными выше известными ограничениями статического анализа и динамического символьного исполнения, примененными в последнее время развиваются комбинированные подходы, позволяющие компенсировать недостатки каждого из подходов в отдельности. В работе [19], посвященной описанию инструмента Check'n'Crash, рассматривается подход совмещения статического анализа кода программ на языке Java с целью обнаружения критических ошибок времени исполнения и целенаправленной генерации тестовых сценариев для подтверждения найденных ошибок в процессе запуска сгенерированных тестов. В работе [20] описывается подход к улучшению производительности генерации тестов для ошибок, найденных в программах при помощи статического анализа, путём выделения срезов (slice), относящихся только к потоку данных, влияющих на найденную ошибку. В работе [21] описывается инструмент CONBOL, котором метод статического анализа используется для обнаружения ошибок в функциях программы и генерации тестовых сценариев для демонстрации найденных ошибок. В работе [22] рассматривается инструмент HVDS, который совмещает обнаружение подозрительных мест в программе методом статического анализа и генерацию входных данных для прохождения по пути, приводящем к найденным подозрительным местам. В работе [23], посвященной инструменту DSD-Crasher, который развивает идеи Check'n'Crash, описывается подход к обнаружению критических ошибок времени исполнения, приводящих к аварийному завершению программы путем совмещения статического анализа программ на языке Java, который получает ограничения на значения входных данных на основе извлечения инвариантов поведения программы из имеющихся тестов программы и автоматической генерации тестовых сценариев для проверки истинности найденных программных ошибок. В работе [24] предлагается совмещения статического анализа программы для генерации регулярных выражений, описывающих ограничения на входные данные программы, генетического алгоритма для генерации входных данных программы для обнаружения критических ошибок времени исполнения программы. В работе [25] предлагается метод совмещающий статический анализ на основе абстрактной интерпретации программы и генерацию тестовых наборов для подтверждения дефектов в программе. В работе [26] предлагается

подход к совмещению статического анализа программ для операционной системы Android с целью обнаружения последовательности событий операционной системы, приводящей к исполнению определенных частей кода программ, и динамического анализа программы направляемого этой информацией с целью генерации последовательности событий, приводящих к исполнению программы по критическим с точки зрения безопасности путям в программе (с динамической загрузкой кода или вызовами к методам операционной системы). В работе [27] рассматривается инструмент, совмещающий динамическое символьное исполнение программ на языке C#, которое в процессе построения тестового покрытия программы одновременно проверяет запрещенные состояния, заранее найденные в процессе статического анализа программы, обнаруживающего нарушение контрактов вызова функций.

В данной статье мы рассматриваем подход совмещения статического анализа исходного кода программ с целью обнаружения потенциальных дефектов, статического анализа бинарного кода программ с целью построения возможных путей до места потенциального дефекта и вычисления входных данных для достижения потенциальных дефектов с учётом прохождения трассы дефекта при помощи динамического символьного исполнения. Статья организованна следующим образом: в разделе 2 даётся краткий обзор понятий и подходов к анализу программ, в разделе 3 описывается предлагаемый нами подход к построению входных данных для подтверждения достижимости дефектов, в разделе 4 приводится описание и результаты экспериментальной проверки предложенного подхода, в разделе 5 делаются выводы и предлагаются дальнейшие направления исследований в области анализа программ на наличие дефектов.

2. Обзор базовых понятий

2.1 Дефекты с трассой исполнения

Среди дефектов, обнаруживаемых статическими анализа торами стоит выделить группу дефектов, которые называются дефектами с трассой исполнения или дефектами типа «инициализация-реализация». На Рис. 1 приведен пример программы с дефектом такого типа.

```
void * alloc_buffer(size_t size){
    return malloc(sizeof(int) * size);
}
void mlk(size_t number, int * input){
    int *buffer = alloc_buffer(number);
    memcpy(buffer, input, number);
    if(number < 2) {
        return; // Memory leak
```

```
        }
        ...
        // Processing of data
        ...
        free(buffer);
    }
```

Рис. 1. Пример фрагмента программы с дефектом типа «инициализация-реализация»

Fig. 1. Example of a program fragment with an initialization-implementation type defect

Ошибочная ситуация инициализируется в функции `alloc_buffer` путём выделения памяти. Далее в зависимости от вычисления условия `if(number < 2)`, которое в данном случае попадает как событие в трассу дефекта, происходит переход на оператор возврата из функции, на котором происходит реализация дефекта утечка памяти в связи с потерей значения локального указателя `buffer`.

Таким образом дефекты типа «инициализация-реализация» определяются следующими характеристиками:

- *точка инициализации* дефекта, где происходит инициализации ошибочной ситуации;
- *точка реализации* дефекта, где происходит ошибочная ситуация;
- *трасса* от точки инициализации до точки реализации ошибочной ситуации, если какие-либо специфические события зарегистрированы статическим анализатором между точками инициализации и реализации дефекта.

Разыменование нулевого указателя, разыменование неинициализированного указателя, использование непроверенных входных данных в уязвимой функции, утечка памяти, утечка ресурса, выход за границы буфера в памяти являются дефектами типа «инициализация-реализация» и обнаруживаются при помощи анализа, чувствительного к пути исполнения в программе.

2.2 Путь исполнения

Путь исполнения в программе представляет собой последовательность инструкций от точки входа до точки останова программы, исполненных процессором. Если поведение программы детерминировано, то путь исполнения может быть описан последовательностью условных переходов, при этом для каждого пути исполнения может существовать от нуля до нескольких различных наборов входных данных. При чём каждый из них уникально идентифицирует единственный путь исполнения. Для генерации входных данных в процессе итеративного динамического символьного исполнения

программы требуется собрать трассу исполнения, в виде инструкций, исполненных процессором, чтобы получить достаточно информации для определения нового пути исполнения программы.

На рис. 2 представлен пример программы, в которой дефект разыменования нулевого указателя реализуется, если в качестве первого аргумента командной строки будет передана строчка `'bad'`.

```
int main(int argc, char* argv[]){
    char* buffer = NULL;
    if(argv[1][0] == 'b')
        if(argv[1][1] == 'a')
            if(argv[1][2] == 'd')
                printf("%s\n", buffer);
    return 0;
}
```

Рис. 2. Пример пути исполнения, приводящего к дефекту в программе

Fig. 2. An example of the execution path leading to a defect in the program

Для подтверждения достижимости дефекта в программе методом итеративного динамического символьного исполнения необходимо сгенерировать такой набор входных данных, который приведет к исполнению программы по пути, содержащему точку инициализации дефекта, точки событий трассы дефекта и точку реализации дефекта.

2.3 Итеративное динамическое символьное исполнение

В процессе итеративного динамического символьного исполнения программа запускается множество раз и на каждой итерации запуска анализируется новый путь в программе. Это достигается применением метода чередования путей (path alternation), основанном на инвертировании направления перехода в одном из условных переходов, зависящем от входных данных. Путь в программе определяется системой формул, описывающей ограничения над областями памяти и регистрами, содержащими помеченные (tainted) данные, то есть данные полученные из внешних источников, таких как аргументы командной строки, стандартные потоки ввода вывода, файлы и других. Будем называть такую систему формул набором ограничений пути (path constraint).

Для построения системы формул ограничений пути требуется отслеживать поток операций над помеченными данными в программе в процессе её исполнения. Для этого требуется отслеживать все операции чтения из источников внешних данных программы и операции перезаписи ранее прочитанных помеченных значений до тех пор, пока они не будут перезаписаны непомеченными (внутренними) данными программы. Поскольку каждый путь в программе определяется множеством условных переходов, условия которых зависят от внешних данных программы, то они также должны быть записаны в формулу ограничений пути. Новый путь в программе может быть получен

инвертированием условий в исходной формуле ограничений пути. Предположим в трассе есть N условных переходов, зависящих от внешних данных программы. Соответственно N новых наборов входных данных для N путей в программе может быть сгенерировано по результатам исполнения по данному пути. Трасса номер i будет соответствовать *i-тому* условному переходу, все условия до условия i должны остаться неизменными, а условия после i должны быть исключены из формулы ограничений пути. Проверка системы формул ограничений на выполнимость позволит, в случае выполнимости, подобрать значения нового набора входных данных, соответствующего новому пути исполнения в программе.

На первом шаге итеративного символьного исполнения программа запускается с некоторым начальным значением набора входных данных (зерном) и строится начальная трасса исполнения программы. После того, как условия условных переходов, зависящих от входных данных, будут инвертированы и будут сгенерированы новые наборы входных данных, каждый из которых будет оценен при помощи некоторой эвристики, показывающей перспективность анализа программы по соответствующему набору пути исполнения. Каждому набору входных данных присваивается оценка на основе эвристики. Набор входных данных с лучшей оценкой выбирается для запуска программы на следующей итерации. Все остальные наборы сохраняются для использования на последующих итерациях. Когда условие в определенном условном переходе инвертировано, все условия условных переходов до него должны быть проигнорированы в процессе инвертирования на следующей итерации анализа при запуске на соответствующем наборе входных данных. Для этого инструменту анализа вместе с набором входных данных для анализируемой программы передается номер условного перехода, начиная с которого необходимо проводить инвертирование условного выражения для получения следующих наборов входных данных. Среди эвристик можно выделить следующие: случайный выбор, интересен наиболее глубокий путь, интересен путь с максимальным приростом проанализированных базовых блоков и другие.

3. Описание подхода

3.1 Обзор

В качестве входных данных для предлагаемого подхода предлагается использовать предупреждение о потенциальном дефекте в программе, сгенерированное инструментом статического анализа или аналитиком, а также исполняемый код программы, собранный из того же исходного кода, который был обработан статическим анализатором. Предупреждение может содержать детальную информацию о дефекте, такую как точку инициализации, трассу и точку реализации дефекта. В вырожденном случае оно должно содержать точку

реализации дефекта. Сама задача подтверждения достижимости дефекта может быть разделена на несколько подзадач.

Во-первых, происходит преобразование трассы дефекта в исходном коде в трассу дефекта в исполняемом коде. Такое преобразование требуется в связи с тем, что в процессе динамического символьного исполнения производится анализ исполняемого кода программы и использование информации об исходном коде на данном уровне не представляется целесообразным. В то же время наличие оптимизаций кода может серьезно изменить структуру исполняемого кода по сравнению с исходным кодом, в связи с чем в описываемом подходе предполагается отсутствие оптимизаций исполняемого кода.

Во-вторых, для определения достижимости дефекта типа «инициализация-реализация» требуется проанализировать все пути исполнения в программе, которые проходят через точку инициализации дефекта при чем максимально быстрым способом. Для решения этой задачи предлагается использовать итеративное динамическое символьное исполнение для генерации наборов входных данных и ввести новую эвристику определения наиболее перспективного пути для анализа, которая будет рассчитывать минимальное расстояние от пути исполнения программы на текущей итерации до точки инициализации дефекта с использованием графа потока управления анализируемой программы [28]. Данный граф строится перед началом динамического символьного исполнения методом статического анализа исполняемого кода.

После того, как точка инициализации дефекта достигнута итеративно используется направленное динамическое символьное исполнение для прохождения по трассе дефекта от точки инициализации дефекта, через точки событий в трассе дефекта до точки инициализации дефекта. Если удалось построить набор входных данных такой, что исполнение программы проходит через всю трассу дефекта, включая точки инициализации и реализации ошибочной ситуации, то считается, что достижимость дефекта подтверждена. В обратном случае итеративное динамическое символьное исполнение должно быть продолжено до тех пор, пока для всех известных путей, приводящих исполнение в точку инициализации дефекта, не будет проведен анализ достижимости дефекта.

3.2 Эвристика выбора кратчайшего пути

Для быстрого достижения точки инициализации дефекта предлагается использовать следующую эвристику: из всех возможных путей исполнения программы, проходящих через заданные точки, на каждом шаге итеративного динамического символьного исполнения выбирается такой путь, в котором количество условных переходов наименьшее. Кратчайший путь до точки инициализации дефекта вычисляется на графе потока управления программы,

полученного методом статического анализа исполняемого кода программы до начала итеративного динамического символьного исполнения программы.

Чтобы собрать всю требующуюся информацию, которая будет использоваться при расчете эвристики кратчайшего пути, достаточно построить сокращенный граф вызовов программы, включающий только те функции, через которые проходит путь исполнения до точки инициализации дефекта, через точки событий трассы дефекта и точку реализации дефекта. Этот граф мы строим методом обратной трассировки от точки инициализации дефекта. Сначала определяются все точки вызова функции, содержащей точку инициализации дефекта. Затем определяются точки вызова функций, вызывающих функцию, содержащую точку инициализации дефекта, и так до тех пор, пока не будет найдена функция, не вызывающаяся из других функций программы (например, главная функция программы, являющаяся точкой входа в программу). Если в процессе построения графа вызовов встречается рекурсия, то соответствующие функции пропускаются, как уже включенные в граф вызовов.

После построения неполного графа вызовов программы для каждой функции строится неполный граф потока управления функции, содержащий только те базовые блоки и условные переходы, которые лежат на путях, приводящих к точке инициализации дефекта. После построения такого сокращённого графа потока управления программы для каждого направления условного перехода в этом графе проставляется расстояние (количество условных переходов) до точки инициализации дефекта. Далее будем называть такой сокращенный и взвешенный граф потока управления программы графом достижения точки инициализации дефекта.

На рис. 3 изображен граф достижения точки инициализации дефекта в функции `target`. Расстояние от точки вызова функции target из функции transit до точки инициализации дефекта равно 0, поскольку мы исключаем ребра безусловных переходов на графе потока управления, расстояние от точки входа в функцию transit до точки инициализации дефекта равно 2, расстояние от точки входа в функцию main до точки инициализации дефекта равно 5 и 6 соответственно двум путям в функции main.

В процессе вычисления эвристики минимального расстояния мы принимаем во внимание только те базовые блоки, которые лежат на пути, ведущем к точке инициализации дефекта, потому что все остальные базовые блоки не доминируют над точкой инициализации дефекта и могут быть проигнорированы.

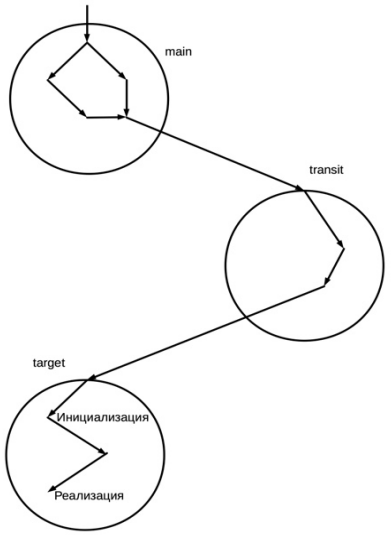

Рис. 3. Граф достижения точки инициализации дефекта
Fig. 3. The graph of reaching a point of defect initialization

Предположим, что на пути исполнения программы в процессе некоторой итерации есть три условных перехода *A*, *B* и *C*, из которых возможно достижение точки инициализации дефекта. Тогда будет выбран тот условный переход, альтернативная ветвь которого будет иметь наименьшую оценку среди других:

```
E = min(distance(C, T), distance (B,T), distance(A, T))
```

где, `T` – точка инициализации дефекта, а `distance(X,T)` функция расчёта расстояния от узла `X` до точки инициализации дефекта в графе достижения точки инициализации дефекта. Результатом вычисления эвристики для некоторого пути исполнения программы является расстояние от ближайшего к точке инициализации дефекта условного перехода на пути исполнения программы. Расстояние становится равным нулю, когда точка инициализации дефекта достигнута.

3.3 Трансформация пути дефекта

После того как точка инициализации дефекта достигнута, производится направленный анализ, который проводит исполнение программы по трассе дефекта. Для этого требуется преобразовать трассу дефекта, выраженную в терминах исходного кода в трассу дефекта в исполняемом коде, чтобы её можно было использовать в процессе итеративного динамического символьного исполнения. В рамках нашей реализации трансформация производилась для исходной трассы в программе на языке Си и исполняемого кода,

сгенерированного компиляторами GCC [29]. Реализация была протестирована для версий GCC 4.6.3 и 4.9.2 для платформы x86-64. Процесс трансформации для условного оператора описан ниже. Циклы, тернарный оператор, оператор-переключатель и другие в данной работе не рассматриваются и включены в список направлений дальнейшего исследования.

Исходя из того, что исполняемый код программы не подвергался оптимизации, последовательность условных переходов в исполняемом коде будет соответствовать последовательности конъюнктов в условных выражениях условных операторов в исходном коде.

Условный переход в исполняемом коде это либо переход на базовый блок, соответствующий `then`-ветке или базовый блок, соответствующий `else`-ветке в исходном коде программы. Возможные варианты в исполняемом коде для условного оператора показаны на рис. 4.

```
if(A)
{
    // then-ветка
}
else
{
    // else-ветка
}
```

Вариант 1: переход	Вариант 2: переход
`; на then-ветку`	`; на else-ветку`
`4004fc: cmp $0x0, -0x3(%rbp)`	`4004fc: cmp $0x0, -0x3(%rbp)`
`400500: jne 400515`	`400500: je 400515`
…	…
`400515: ; then-ветка`	`400515: ; else-ветка`

Рис 4. Пример вариантов генерации кода для условного оператора
Fig. 4. Example of code generation options for a conditional statement

Правила генерации кода для сложных условий с конъюнкциями, дизъюнкциями и отрицаниями приведены на рис. 5.

A && B	• переход на then-ветку: вариант 2 для A и вариант 1 для B; • переход на else-ветку: вариант 2 для A и вариант 2 для B
A || B	• переход на then-ветку: вариант 1 для A и вариант 1 для B;

	• переход на else-ветку: вариант 1 для A и вариант 2 для B
!A	• переход на then-ветку: вариант 2 для A; • переход на else-ветку: вариант 1 для A.

Рис 5. Варианты генерации неоптимизированного кода для условного оператора в компиляторе GCC версии 4.6.3 и 4.9.2

Переход на `then`-ветку или `else`-ветку зависит от операций внутри `then`- или `else`- блоков и формы условного выражения. Например, было обнаружено, что:

- переход на then-блок генерируется если в соответствующем блоке исходный код содержит операцию безусловного перехода: `goto`, `break` или `continue`;
- в то же время, переход на `else`-ветку генерируется если в условном операторе нет `else`-ветки и содержится внутри `then`-ветки вышестоящего условного оператора (при этом не обязательно непосредственного вышестоящего);
- переход на `else`-ветку происходит во всех остальных случаях.

После того как трансформация пути из исходного кода в исполняемый код произведена может оказаться, что в исполняемом коде несколько путей соответствуют одному пути в исходном коде. Например, на рис. 6 приведена дизъюнкция `A || B` которая должна быть вычислена в true, которой в исполняемом коде будет соответствует два пути исполнения:

- путь, соответствующий условию `A` – истинно;
- путь, соответствующий условию `A` – ложно, `B` – истинно.

```
if(A || B) { ... } else { ... }
4004f6: cmp $0x0, -0x4(%rbp)
4004fa: jne 400502                    ; A
4004fc: cmp $0x0, -0x3(%rbp)
400500: je 400515                     ; B
400502: ; then-ветка
…
400515: ; else-ветка
```

Рис. 5. Пример ситуации, где один путь в исходном коде соответствует двум путям в исполняемом коде
Fig. 5. A case where one path in the source code corresponds to two paths in executable code

Поэтому, трансформация одного пути в терминах исходного кода может привести к нескольким путям исполнения в исполняемом коде и могут существовать несколько наборов входных данных, удовлетворяющих одному сообщению о дефекте.

3.4 Воспроизведение пути инициализация-реализация

Когда точка инициализации потенциального дефекта достигнута, происходит применение направленного динамического символьного исполнения для исполнения пути, на котором лежат события трассы потенциального дефекта. Это позволяет сократить время достижения точки реализации потенциального дефекта. В процессе направленного динамического символьного исполнения по трассе дефекта собираются условия прохождения пути для проверки выполнимости результирующего пути и генерации входных данных и воспроизведения генерации входных данных, которые позволят провести исполнение по трассе дефекта и показать его достижимость.

Стоит отметить, что модель отслеживания потока входных данных, описанная в пункте 2.3, эффективна, но не полна: существуют неявно помеченные данные, которые модель игнорирует, учитывая только явные зависимости по входным данным. Для генерации входных данных необходимо учитывать поток явно помеченных данных, но условие на пути, которое необходимо инвертировать может зависеть как от помеченных, так и от непомеченных данных. В связи с этим инвертирование условий, зависящих от помеченных и от непомеченных данных необходимо производить раздельно. Инвертирование помеченных условных переходов приводит к генерации нового набора входных данных и исполнению нового пути. Однако, инвертирование непомеченных условий бесполезно, так как в модели нет связи между непомеченными условными переходами и входными данными, в связи с чем инвертирование непомеченного условия не приведет к генерации нового набора входных данных. Если трасса дефекта содержит событие, связанное с условным переходом, не зависящим от помеченных данных, результат направленного анализа может оказаться некорректным. При этом возможно три ситуации:

- условный переход не зависит от входных данных, но должен быть инвертирован;
- условный переход неявно зависит от входных данных, но условие не должно быть инвертировано;
- условный переход неявно зависит от входных данных, но условие не должно быть инвертировано.

В первом случае можно заключить, что данный путь от инициализации до реализации дефекта несовместен. Второй случай соответствует ситуации, когда условие уловного перехода в трассе дефекта соответствует актуальному

значению и путь оказывается совместным и, таким образом, достижимость дефекта подтверждается. В третьем случае возможны следующие варианты:

- инвертированное условие не совместно с предусловием пути и результат направленного динамического символьного исполнения некорректен;
- реализация дефекта на самом деле не зависит от инвертированного условия, которое добавлено в трассу инструментом статического анализа по ошибке, и результат динамического символьного исполнения корректен.

Поэтому требуется ручная проверка дефекта и сгенерированных входных данных, если непомеченный условный переход был инвертирован в процессе динамического символьного исполнения.

Результатом выполненного направленного динамического символьного исполнения будет система формул, описывающая ограничения пути в терминах данных, которыми оперирует программа. Если условие условного перехода инвертируется, тогда новая формула должна быть добавлена в систему

```
tainted_condition_var != old_condition
```

где `tainted_condition_var` – символьная переменная, содержащая условие условного перехода, а `old_condition` – значение условия до инвертирования. Если новая система формул неразрешима, то путь исполнения, соответствующий инвертированному условному переходу невыполним и требуется анализ следующего пути из списка путей, достигающих точки инициализации дефекта.

4. Оценка предложенного подхода

Предложенный подход к определению достижимости программных дефектов, найденных методами статического анализа исходного кода, был реализован на основе инструмента динамического символьного исполнения программ *Avalanche* [30], разработанного в ИСП РАН. В инструмент *Avalanche* были внесены следующие изменения:

- добавлен алгоритм расчёта эвристики оценки пути на основе минимального расстояния до интересующей точки в программе;
- изменен модуль распространения помеченных данных с целью реализации алгоритма направленного динамического символьного исполнения;
- реализован модуль извлечения сокращенного графа вызовов и сокращенного графа потока управления функций, который представляет собой разборщик дизассемблированного представления программы, полученного при помощи инструмента `objdump` [31];
- реализован алгоритм трансформации трассы дефекта на основе

обработчика дерева абстрактного синтаксиса, которое предоставляется инструментом статического анализа исходного кода программы.

Реализация была протестирована на наборе программ с открытым исходным кодом из комплекта поставки Debian Linux:

- **a2ps** – утилита конвертации файлов в различных форматах в PostScript документ
- **bbe** – утилита для редактирования двоичных файлов
- **bc** – интерпретатор алгебраических выражений с неограниченной точностью
- **byacc** – генератор LALR(1) парсеров
- **bzip2** – утилита для сжатия файлов
- **cabextract** – инструмент распаковки Microsoft cabinet файлов (.cab)
- **ccrypt** – утилита кодирования и раскодирования файлов и потоков
- **chrpath** – утилита редактирования путей к библиотекам в ELF-файлах
- **cpio** – утилита сжатия и извлечения архивов в различных форматах
- **discount** – программа трансформации текстовых файлов в формат Markdown

Табл. 1 содержит данные по предупреждениям о дефектах на проектах, найденных референсным инструментом статического анализа исходного кода программ на языке Си. Предупреждения были размечены на истинные (T), ложные (F) и подтвержденные (C) методом направленного динамического символьного исполнения. Разметка предупреждений о дефектах в программе на истинные и ложные произведена вручную. Всего было классифицировано 348 предупреждений о дефектах, 113 из них признаны ложными, 235 – истинными. Референсный инструмент статического анализа исходного кода программ был настроен на поиск дефектов типа «инициализация-реализация» следующих типов:

- разыменование нулевого указателя (NPD);
- выход за границы буфера в памяти (BOF);
- утечка ресурса (LEAK);
- использование входных данных в уязвимых функциях без проверки (TAINT);
- использование неинициализированных значений (UNINIT).

Табл. 1 Результаты экспериментальной проверки подхода. F – ложные предупреждения, T – истинные предупреждения, C – подтвержденные предупреждения
Table. 1 Results of experimental verification of the approach. F - false warnings, T - true warnings, C - confirmed warnings

Project	NPD			BOF			LEAK			TAINT			UNINIT			TOTAL		
	F	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**	**F**	**T**	**C**
a2ps	65	33	2*	5	2		2	22	11		7	2				72	64	15
bbe		2			10			1									13	0
bc							1	6	4		1	1				1	7	5
byacc				10				1						2		10	3	0
bzip					1			14			9			1			25	0
cabextract					2			3			1	1					6	1
ccrypt							7	15	2		1					7	16	2
cgrpath								9			10	3					19	3
cpio	19	22					2	18	2		4		1	1		22	45	2
discount		37					1				1					1	38	0
Total	84	94	2*	15	15	0	13	89	19	0	34	7	1	4	0	113	235	28

Предложенный подход был применен к каждому из предупреждений о дефекте от инструмента статического анализа исходного кода программ. Инструмент направленного динамического символьного исполнения запускался с 30-ти минутным ограничением по времени. Всего для 28 предупреждений о дефектах была подтверждена достижимость. Ни для одного из дефектов типа использование неинициализированного значения и выход за границы буфера не была подтверждена достижимость. Стоит отметить, что два подтвержденных дефекта типа разыменование нулевого указателя оказались подтверждены ложно. Связано это с имеющимися ограничениями в реализации предложенного подхода: отсутствием добавления в систему формул ограничений пути предиката безопасности. В обоих случаях ложного подтверждения дефекта требуется добавление ограничения вида:

```
variable == NULL
```

которое соответствовало бы строчке исходного кода:

```
variable = someFunction(arguments);
```

где функция `someFunction` ни на одном из выполнимых путей не возвращает `NULL`. Это означает, что достижимость точки инициализации дефекта, прохождение по трассе дефекта и достижимость точки реализации дефекта ещё не означает подтверждение наличия дефекта в программе. Пока не реализована функциональность добавления предиката безопасности для проверки

реализации дефекта, данное ограничение подхода будет ложно подтверждать достижимость нереализуемых дефектов.

Для остальных предупреждений о дефектах, которые классифицированы как истинные, достижимость была подтверждена. Таким образом 26 дефектов были подтверждены верно. Для 9 предупреждений о дефектах потребовалась ручная проверка в связи с наличием в трассе события, связанного с непомеченным условием в условном переходе. Все они были классифицированы как ложные.

5. Заключение

В данной статье описан подход для подтверждения при помощи динамического символьного исполнения достижимости дефектов типа инициализация-реализация, найденных при помощи статического анализа исходного кода. Экспериментальная проверка реализации подхода показала применимость данного подхода для подтверждения достижимости дефектов на наборе реальных программ с открытым исходным кодом.

По результатам реализации подхода и экспериментальной проверки можно выделить несколько направлений для дальнейших исследований с целью улучшения применимости предложенного подхода на практике:

- исследование алгоритмов минимально достаточной помеченности потока данных программы, с учётом неявных зависимостей по данным и по управлению;
- исследование и реализация подхода определения предиката безопасности для различных дефектов с целью подтверждения дефекта, а не только его достижимости, методами итеративного динамического символьного исполнения;
- поддержка других типов дефектов в программах.

Список литературы

[1]. Vogelsang A., Fehnker A., Huuck R., Reif W. Software Metrics in Static Program Analysis. ICFEM'10 Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, Shanghai, Shina, November 17-19, 2010, pp. 485-500

[2]. Kim Y., Kim Y., Kim T., Lee G., Jang Y., Kim M. Automated unit testing of large industrial embedded software using concolic testing. ACE'13 Proceedings of the 28yh IEEE/ACM International Conference on Automated Software Engineering, Silicaon Valley, CA, USA, November 11-15, 2013, pp. 519-528

[3]. Xie Y., Chou A., Engler D. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. ESEC/FSE-11 Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, Finland, September 01-05, 2003, pp. 327-336

[4]. Bessey A., Block K., Chelf B, Chow A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D. A few billion lines of code later: using static analysis to find bugs in the real world. Communications od ACM, vol. 53, issue 2, February 2010, pp. 66-75

[5]. Иванников В.П., Белеванцев А.А., Бородин А.Е., Игнатьев В.Н., Журихин Д.М., Аветисян А.И., Леонов М.И. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды ИСП РАН, том 26, вып. 1, 2014, стр. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7

[6]. Engler D., Chelf B., Chou A., Hallen S. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, vol. 4, article No.1, San-Diego, CA, USA, October 22-25, 2000

[7]. Johnson B., Song Y., Murphy-Hill E., Bowdidge R., Why don't software developers use static analysis tools to find bugs? ICSE'13 Proceedings of the 2013 International conference on Software Engineering. San Francisco, CA, USA, May 18-26, 2013

[8]. Christakis M., Müller P., Wüstholz An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzers. VMCAI'15 International Workshop on Verification, Model Checking and Abstract Interpretation, Springer, 2015, pp. 336-354

[9]. Livshits B., Sridharan M., Smaragdakis Y., Lhoták O., Amaral J.N., Chang B.-Y. E., Guyer S.Z., Khedker U.P., Møhler A., Vardoulakis D. In defence of soundness: a manifesto. Communications of ACM, vol. 58, no. 2, February 2015

[10]. Cadar C., Dunbar D., Endger D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems. OSDI'08 Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, San Diego, CA, USA, December 08-10, 2008, pp. 209-224

[11]. Averginos T., Cha S.K., Revert A., Schwartz E.J., Woo M., Brumley D. Automatic Exploit Generation. Communications of the ACM, volume 57, issue 2, February 2014, pp. 74-84

[12]. Chipunov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) – Special Issue APLOS 2011, article no. 2, volume 30, issue 1, February 2012

[13]. Manevich R., Sridharan M., Adams S., Das M., Yang Z. PSE: explaining program failures via post-mortem static analysis. SIGSOFT'04/FSE-12 Proceedings of the 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering, Newport Beach, NY, USA, 2004, pp. 63-72

[14]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: a New Approach to Computer Security via Binary Analysis. ICISS'08 Proceedings of the 4th international Conference on Information Systems Security, Hydarabad, India, December 16-20, 2008, pp. 1-25

[15]. Sen K., Marinov D., Agha G. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, September 05-09, 2005, pp. 263-272

[16]. King J.C. Symbolic Execution and Program Testing. Communications of the ACM, volume 19, issue 7, 1976, pp. 385-394

[17]. Cadar C., Ganesh V., Pawlowski P., Dill D.L., Engler D.R. EXE: automatically generating inputs of death. CCS'06 Proceedings of the 13th ACM Conference on Computer and

Communications Security, Alexandria, Virginia, USA, October 30 - November 03, 2006, pp. 322-335

[18]. Schwartz E.J., Averginos T., Brumley D. All You Ever Wanted to Know About Dynamic Tait Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). SP'10 Proceedings of the 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 16-19, 2010, pp. 317-331

[19]. Csallner C., Smaragdakis Y. Check'N'Crash: combining static checking and testing. ICSE'05 Proceedings of the 27th international conference on software engineering, St. Louis, MO, USA, May 15-21, 2005, pp. 422-431

[20]. Chebaro O., Kosmatov N., Giorgetti A., Julliand J. Programs slicing enhances a verification technique combining static and dynamic analysis. SAC'12 Proceedings of the 27th Annual ACM Symposium of Applied Computing, Trento, Italy, March 26-30, 2012, pp. 1284-1291

[21]. Kim T., Park J., Kulinda I., Jang Y. Concolic Testing Framework for Industrial Embedded Software. APSEC'14 Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, volume 2, Jeju, South Korea, December 01-04, 2014, pp. 7-10

[22]. Hanna A., Ling H.Z., Yang X., Debbabi M. A synergy between static and dynamic analysis or the detection of software security vulnerabilities. OTM'09 Proceedings of the Confederated International Congress, CoopIS, DOA, IS and ADBASE 2009 on the Move to Meaningful Internet Systems: part II. Vilamoura, Portugal, November 01-06, 2009, pp. 815-832

[23]. Csallner C, Smaragdakis Y. DSD-Crasher: a hybrid analysis tool for bug finding. IISTA'06 Proceedings of the 2006 International Symposium on Software Testing and Analysis. Portland, Maine, USA, July 17-20, 2006, pp. 245-254

[24]. Artho C, BIere A. Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science (ENTCS), volume 131, May, 2005, pp. 3-14

[25]. Chebaro o., Kostomarov N., Giorgetti A., Julliand J. Combining static analysis and test generation for C program debugging. TAP'10 Proceedings of the 4th International Conference on Tests and Proofs. Málaga, Spain, July 01-02, 2010, pp. 94-100

[26]. Schütte J., Fedler R., Tetze D. ConDroid: targeted dynamic analysis of Android Applications. AINA'15 Proceedings of IEEE 26th international Conference on Advanced Information Networking and Applications, Gwangui, South Korea, March 24-27, 2015

[27]. Ge X., Taneja K., Xie T., Tillmann N. DyTa: dynamic symbolic execution guided with static verification results. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, Mau 21-28, 2011, pp. 992-994

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

[29]. Stallman R. M. Using the GNU Compiler Collection: a GNU Manual for GCC Version 4.3.3. Free Software Foundation Inc., May, 2004

[30]. Исаев И.К., Сидоров Д.В. Применение динамического анализа для генерации входных данных, демонстрирующих критические ошибки и уязвимости в программах. Программирование, 2010, №4, стр. 1-16

[31]. GNU binutils [HTML] http://www.gnu.org/software/binutils, accessed 01.11.2017

An approach of reachability determination for static analysis defects with help of dynamic symbolic execution

A.Y. Gerasimov <agerasimov@ispras.ru>
L.V. Kruglov <kruglov@ispras.ru>
M.K. Ermakov <mermakov@ispras.ru>
S.P. Vartanov <svartanov@ispras.ru>
Ivannikov Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. Historically program analysis methods are divided into two groups – static program analysis methods and dynamic program analysis methods. In this paper, we present a combined approach which allows to determine reachability for defects found by static program analysis techniques through applying dynamic symbolic execution for a program. This approach is an extension of our previously proposed approach for determining the reachability of specific program instructions using dynamic symbolic execution. We focus on several points in the program which include a defect initialisation point, a defect realisation point, and additional intermediate conditional jumps related to the defect in question. Our approach can be described as follows. First of all, we perform static analysis of program executable code to gather information on execution paths which guide dynamic symbolic execution to the point of defect initialisation. Next, we perform concolic execution in order to obtain an input data set to reach the defect initialisation point as well as the defect realisation point through intermediate conditional jumps. Concolic execution is guided by minimizing the distance from a previous path to the next defect trace point when selecting execution paths. The distance metric is calculated using an extended graph of the program combining its call graph and portions of its control flow graph that include all the paths through which the defect realisation point can be reached. We have evaluated our approach using several open source command line programs from Linux Debian. The evaluation confirms that the proposed approach can be used for classification of defects found by static program analysis. However, we have found some limitations, which prevent deploying this approach to industrial program analysis tools. Mitigation of these limitations serves as one of the possible directions for future research.

Keywords: static program analysis; dynamic program analysis.

DOI: 10.15514/ISPRAS-2017-29(5)-7

For citation: Gerasimov A.Y., Kruglov L.V., Ermakov M.K., Vartanov S.P. An approach of reachability confirmation for static analysis defects with help of dynamic symbolic execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 5, 2017. pp. 111-134 (in Russian). DOI: 10.15514/ISPRAS-2017-29(5)-7

References

[1]. Vogelsang A., Fehnker A., Huuck R., Reif W. Software Metrics in Static Program Analysis. ICFEM'10 Proceedings of the 12th International Conference on Formal Engineering Methods and Software Engineering, Shanghai, Shina, November 17-19, 2010, pp. 485-500

[2]. Kim Y., Kim Y., Kim T., Lee G., Jang Y., Kim M. Automated unit testing of large industrial embedded software using concolic testing. ACE'13 Proceedings of the 28yh IEEE/ACM International Conference on Automated Software Engineering, Silicaon Valley, CA, USA, November 11-15, 2013, pp. 519-528

[3]. Xie Y., Chou A., Engler D. ARCHER: Using Symbolic, Path-Sensitive Analysis to Detect Memory Access Errors. ESEC/FSE-11 Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Helsinki, Finland, September 01-05, 2003, pp. 327-336

[4]. Bessey A., Block K., Chelf B, Chow A., Fulton B., Hallem S., Henri-Gros C., Kamsky A., McPeak S., Engler D. A few billion lines of code later: using static analysis to find bugs in the real world. Communications od ACM, vol. 53, issue 2, February 2010, pp. 66-75

[5]. Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignat'ev V.N., Zhurikhin D.M., Avetisyan A.I., Leonov M.I. Static analyzer Svace for finding of defects in program source code. Trudy ISP RAN/Proc. ISP RAS, vol. 26, issue 1, 2014, pp. 231-250. DOI: 10.15514/ISPRAS-2014-26(1)-7

[6]. Engler D., Chelf B., Chou A., Hallen S. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, vol. 4, article No.1, San-Diego, CA, USA, October 22-25, 2000

[7]. Johnson B., Song Y., Murphy-Hill E., Bowdidge R., Why don't software developers use static analysis tools to find bugs? ICSE'13 Proceedings of the 2013 International conference on Software Engineering. San Francisco, CA, USA, May 18-26, 2013

[8]. Christakis M., Müller P., Wüstholz An Experimental Evaluation of Deliberate Unsoundness in a Static Program Analyzers. VMCAI'15 International Workshop on Verification, Model Checking and Abstract Interpretation, Springer, 2015, pp. 336-354

[9]. Livshits B., Sridharan M., Smaragdakis Y., Lhoták O., Amaral J.N., Chang B.-Y. E., Guyer S.Z., Khedker U.P., Møhler A., Vardoulakis D. In defence of soundness: a manifesto. Communications of ACM, vol. 58, no. 2, February 2015

[10]. Cadar C., Dunbar D., Endger D. KLEE: unassisted and automatic generation of high-coverage tests for complex systems. OSDI'08 Proceedings of the 8th USENIX conference on Operating Systems Design and Implementation, San Diego, CA, USA, December 08-10, 2008, pp. 209-224

[11]. Averginos T., Cha S.K., Revert A., Schwartz E.J., Woo M., Brumley D. Automatic Exploit Generation. Communications of the ACM, volume 57, issue 2, February 2014, pp. 74-84

[12]. Chipunov V., Kuznetsov V., Candea G. The S2E Platform: Design, Implementation, and Applications. ACM Transactions on Computer Systems (TOCS) – Special Issue APLOS 2011, article no. 2, volume 30, issue 1, February 2012

[13]. Manevich R., Sridharan M., Adams S., Das M., Yang Z. PSE: explaining program failures via post-mortem static analysis. SIGSOFT'04/FSE-12 Proceedings of the 12th ACM

SIGSOFT 12th International Symposium on Foundations of Software Engineering, Newport Beach, NY, USA, 2004, pp. 63-72

[14]. Song D., Brumley D., Yin H., Caballero J., Jager I., Kang M.G., Liang Z., Newsome J., Poosankam P., Saxena P. BitBlaze: a New Approach to Computer Security via Binary Analysis. ICISS'08 Proceedings of the 4th international Conference on Information Systems Security, Hydarabad, India, December 16-20, 2008, pp. 1-25

[15]. Sen K., Marinov D., Agha G. CUTE: a concolic unit testing engine for C. ESEC/FSE-13 Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, Lisbon, Portugal, September 05-09, 2005, pp. 263-272

[16]. King J.C. Symbolic Execution and Program Testing. Communications of the ACM, volume 19, issue 7, 1976, pp. 385-394

[17]. Cadar C., Ganesh V., Pawlowski P., Dill D.L., Engler D.R. EXE: automatically generating inputs of death. CCS'06 Proceedings of the 13th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, October 30 - November 03, 2006, pp. 322-335

[18]. Schwartz E.J., Averginos T., Brumley D. All You Ever Wanted to Know About Dynamic Tait Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask). SP'10 Proceedings of the 2010 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 16-19, 2010, pp. 317-331

[19]. Csallner C., Smaragdakis Y. Check'N'Crash: combining static checking and testing. ICSE'05 Proceedings of the 27th international conference on software engineering, St. Louis, MO, USA, May 15-21, 2005, pp. 422-431

[20]. Chebaro O., Kosmatov N., Giorgetti A., Julliand J. Programs slicing enhances a verification technique combining static and dynamic analysis. SAC'12 Proceedings of the 27th Annual ACM Symposium of Applied Computing, Trento, Italy, March 26-30, 2012, pp. 1284-1291

[21]. Kim T., Park J., Kulinda I., Jang Y. Concolic Testing Framework for Industrial Embedded Software. APSEC'14 Proceedings of the 2014 21st Asia-Pacific Software Engineering Conference, volume 2, Jeju, South Korea, December 01-04, 2014, pp. 7-10

[22]. Hanna A., Ling H.Z., Yang X., Debbabi M. A synergy between static and dynamic analysis or the detection of software security vulnerabilities. OTM'09 Proceedings of the Confederated International Congress, CoopIS, DOA, IS and ADBASE 2009 on the Move to Meaningful Internet Systems: part II. Vilamoura, Portugal, November 01-06, 2009, pp. 815-832

[23]. Csallner C, Smaragdakis Y. DSD-Crasher: a hybrid analysis tool for bug finding. IISTA'06 Proceedings of the 2006 International Symposium on Software Testing and Analysis. Portland, Maine, USA, July 17-20, 2006, pp. 245-254

[24]. Artho C, BIere A. Combined Static and Dynamic Analysis. Electronic Notes in Theoretical Computer Science (ENTCS), volume 131, May, 2005, pp. 3-14

[25]. Chebaro o., Kostomarov N., Giorgetti A., Julliand J. Combining static analysis and test generation for C program debugging. TAP'10 Proceedings of the 4th International Conference on Tests and Proofs. Málaga, Spain, July 01-02, 2010, pp. 94-100

[26]. Schütte J., Fedler R., Tetze D. ConDroid: targeted dynamic analysis of Android Applications. AINA'15 Proceedings of IEEE 26th international Conference on Advanced Information Networking and Applications, Gwangui, South Korea, March 24-27, 2015

[27]. Ge X., Taneja K., Xie T., Tillmann N. DyTa: dynamic symbolic execution guided with static verification results. ICSE'11 Proceedings of the 33rd International Conference on Software Engineering, Waikiki, Honolulu, HI, USA, Mau 21-28, 2011, pp. 992-994

[28]. Gerasimov A.Y., Kruglov L.V. 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. DOI: 10.15514/ISPRAS-2016-28(5)-10

[29]. Stallman R. M. Using the GNU Compiler Collection: a GNU Manual for GCC Version 4.3.3. Free Software Foundation Inc., May, 2004

[30]. Isaev I.K., Sidorov D.V. The use of dynamic analysis for generation of input data that demonstrates critical bugs and vulnerabilities in programs. Programming and Computer Software, 2010, vol. 36, No. 4, pp. 225-236. DOI: 10.1134/S0361768810040055

[31]. GNU binutils [HTML] http://www.gnu.org/software/binutils, accessed 01.11.2017