*ПРОГРАММИРОВАНИЕ*, 2010, № 4, с. 1–16

УДК 681.3.06

# ПРИМЕНЕНИЕ ДИНАМИЧЕСКОГО АНАЛИЗА ДЛЯ ГЕНЕРАЦИИ ВХОДНЫХ ДАННЫХ, ДЕМОНСТРИРУЮЩИХ КРИТИЧЕСКИЕ ОШИБКИ И УЯЗВИМОСТИ В ПРОГРАММАХ

© 2010 г.
**И. К. Исаев, Д. В. Сидоров**
*Институт системного программирования РАН*
*109004 Москва, ул. Солженицына, 25*
*E-mail: iisaev@ispras.ru, sidorov@ispras.ru*
Поступила в редакцию

В статье описывается Avalanche – инструмент обнаружения программных дефектов при помощи динамического анализа. Avalanche использует возможности динамической инструментации программы, предоставляемые Valgrind [1], для сбора и анализа трассы выполнения программы. Результатом такого анализа становится либо набор входных данных, на которых в программе возникает ошибка, либо набор новых тестовых данных, позволяющий обойти ранее не выполнявшиеся и, соответственно, еще не проверенные фрагменты программы. Таким образом, имея единственный набор тестовых данных, Avalanche реализует итеративный динамический анализ, при котором программа многократно выполняется на различных, автоматически генерированных тестовых данных, при этом каждый новый запуск увеличивает покрытие кода программы такими тестами. Описаны принципы работы Avalanche, а также приведены результаты применения инструмента к нескольким проектам с открытым исходным кодом. Было обнаружено свыше 15 программных дефектов, многие из которых подтверждены и уже исправлены разработчиками.

## 1. ВВЕДЕНИЕ

Долгое время считалось, что динамический анализ программ является слишком тяжеловесным подходом к обнаружению программных дефектов, и полученные результаты не оправдывают затраченных усилий и ресурсов. Однако, две важные тенденции развития современной индустрии производства программного обеспечения (ПО) позволяют по-новому взглянуть на эту проблему. С одной стороны, при постоянном увеличении объема и сложности ПО любые автоматические средства обнаружения ошибок и контроля качества могут оказаться полезными и востребованными. С другой – непрерывный рост производительности современных вычислительных систем позволяет эффективно решать все более сложные вычислительные задачи. Последние исследовательские работы в области динамического анализа – такие, как SAGE [2], EXE [3] и Flayer [4] – демонстрируют, что несмотря на присущую этому подходу сложность, его использование оправдано, по крайней мере для некоторого класса программ. Помимо исследовательских работ, стоит отметить один из самых успешных открытых проектов в этой области – Valgrind [1]. Популярность этого инструмента на платформе Linux доказывает, что использование динамического анализа оправдано, при условии, что инструмент способен точно обнаруживать реальные ошибки и может работать полностью в автоматическом режиме, т.е. без вмешательства программиста или тестировщика.

В качестве более эффективной альтернативы динамическому анализу часто рассматривают статический анализ. В случае статического анализа поиск возможных ошибок осуществляется без запуска исследуемой программы, например, по исходному коду приложения. Обычно при

этом строится абстрактная модель программы, которая собственно и является объектом анализа. При этом следует подчеркнуть следующие характерные особенности статического анализа:

- возможен раздельный анализ отдельно взятых фрагментов программы (обычно отдельных функций или процедур), что дает достаточно эффективный способ борьбы с нелинейным ростом сложности анализа;
- возможны ложные срабатывания, обусловленные либо тем, что при построении абстрактной модели некоторые детали игнорируются, либо тем, что анализ модели не является исчерпывающим;
- при обнаружении дефекта возникают, во-первых, проблема проверки истинности обнаруженного дефекта и, во-вторых, проблема воспроизведения найденного дефекта при запуске программы на определенных входных данных.

В отличие от статического анализа, динамический анализ осуществляется во время работы программы. При этом:

- для запуска программы требуются некоторые входные данные;
- динамический анализ обнаруживает дефекты только на трассе, определяемой конкретными входными данными, дефекты, находящиеся в других частях программы, не будут обнаружены;
- в большинстве реализаций появление ложных срабатываний исключено, так как обнаружение ошибки происходит в момент ее возникновения в программе; таким образом, обнаруженная ошибка является не предсказанием, сделанным на основе анализа модели программы, а констатацией факта ее возникновения.

Таким образом, вопрос о выборе входных данных достаточно важен при поиске дефектов. При применении инструментов статического анализа входные данные должны быть подобраны для уже обнаруженных дефектов – это необходимо для того, чтобы проверить истинность найденного дефекта и определить, где данная ошибка должна быть исправлена. В случае же динамического анализа входные данные приходится выбирать из соображений либо наибольшего покрытия исполняемого кода, либо возможности анализа наиболее интересных мест. От успешности выбора зависит эффективность динамического анализа.

В недавнее время были предложены различные способы, позволяющие использовать динамический анализ для одновременного нахождения и ошибок, и входных данных, позволяющих их воспроизводить. Суть этих методов сводится примерно к следующей схеме: вводится понятие символических или помеченных (tainted) данных – данных, полученных программой из внешнего источника (стандартный поток ввода, файлы, переменные окружения и т. д.); тем или иным способом собирается информация обо всех использованиях помеченных данных в программе, эта информация записывается в виде булевских ограничений на значения помеченных данных (в эти ограничения может попадать информация о переходах, зависящих от помеченных данных, и информация об использовании помеченных данных в потенциально опасных местах программы). Нахождение значений помеченных данных, делающих эти ограничения выполнимыми, может означать либо возможность возникновения ошибки в программе, либо возможность обхода иных, отличных от обойденных во время первого запуска, частей программы.

В этой работе описывается инструмент Avalanche, реализующий подобный подход на основе открытого фреймворка для динамической инструментации Valgrind [1] и инструмента проверки выполнимости ограничений (solver) STP [5]. При этом перед началом разработки были сформулированы следующие основные требования к создаваемому инструменту:

- как можно более полный анализ приложения;
- эффективное обнаружение дефектов (без ложных срабатываний);
- генерация входных данных, запуск анализируемого приложения на которых воспроизводит дефект.

Также на начальном этапе разработки было решено ограничить область применимости инструмента определенным классом программ – входные данные программа получает из одного файла. Любые другие данные, полученные из других источников (другие файлы, сетевые сокеты, переменные окружения), не принимаются во внимание. Таким образом, далее в статье предполагается, что анализируемая программа работает с одним входным файлом.

Статья имеет следующую структуру: в разделе 2 кратко описана общая схема инструмента, разделы 3–5 посвящены более детальному рассмотрению отдельных компонентов инструмента (в разделе 3 описан процесс отслеживания потока помеченных данных в программе, моделирование промежуточного представления Valgrind при помощи STP-деклараций, генерация ограничений для обхода новых фрагментов программы, опасные операции промежуточного представления Valgrind и генерация ограничений для их проверки, в разделе 4 – стратегия обхода дерева условных переходов программы), в разделе 6 рассматриваются результаты работы Avalanche на 10 проектах с открытым исходным кодом, приводится подробная статистика, перечисляются обнаруженные дефекты.

## 2. ОБЩАЯ СХЕМА РАБОТЫ

Инструмент Avalanche состоит из 4 основных компонент: двух модулей расширения (плагинов) Valgrind – Tracegrind и Covgrind, инструмента проверки выполнимости ограничений STP и управляющего модуля. Tracegrind динамически отслеживает поток помеченных данных в анализируемой программе и собирает условия для обхода ее непройденных частей и для срабатывания опасных операций. Эти условия при помощи управляющего модуля передаются STP для исследования их выполнимости. Если какое-то из условий выполнимо, то STP определяет те значения всех входящих в условия переменных (в том числе и значения байтов входного файла), которые обращают условие в истину.

- В случае выполнимости условий для срабатывания опасных операций программа запускается управляющим модулем повторно (на этот раз без какой-либо инструментации) с соответствующим входным файлом для подтверждения найденной ошибки.
- Выполнимые условия для обхода непройденных частей программы определяют набор возможных входных файлов для новых запусков программы. Таким образом, после каждого запуска программы инструментом STP автоматически генерируется множество входных файлов для последующих запусков анализа. Далее встает задача выбора из этого множества наиболее “интересных” входных данных – т.е. в первую очередь должны обрабатываться входные данные, на которых наиболее вероятно возникновение ошибки. Для решения этой задачи используется эвристическая метрика – количество ранее не обойденных базовых блоков в программе (базовый блок здесь понимается в том смысле, как его определяет фреймворк Valgrind [1]). Для измерения значения эвристики используется компонент Covgrind, в функции которого входит также фиксация возможных ошибок выполнения. Covgrind – гораздо более легковесный модуль, нежели Tracegrind, поэтому возможно сравнительно быстрое измерение значения эвристики для всех полученных ранее входных файлов и выбор входного файла с наибольшим ее значением. Далее будет более детально рассмотрено функционирование каждого из перечисленных компонентов.

## 3. TRACEGRIND: ОТСЛЕЖИВАНИЕ ПОТОКА ПОМЕЧЕННЫХ ДАННЫХ

Итак, основными задачами, которые решает этот компонент, являются:

- отслеживание потока помеченных данных в программе и
- составление условий, характеризующих:
    1. возможности выполнения (или невыполнения) какого-либо перехода;
    2. возможности выполнения какой-либо потенциально опасной операции.

Какие операции считать потенциально опасными – зависит от того, какие типы дефектов

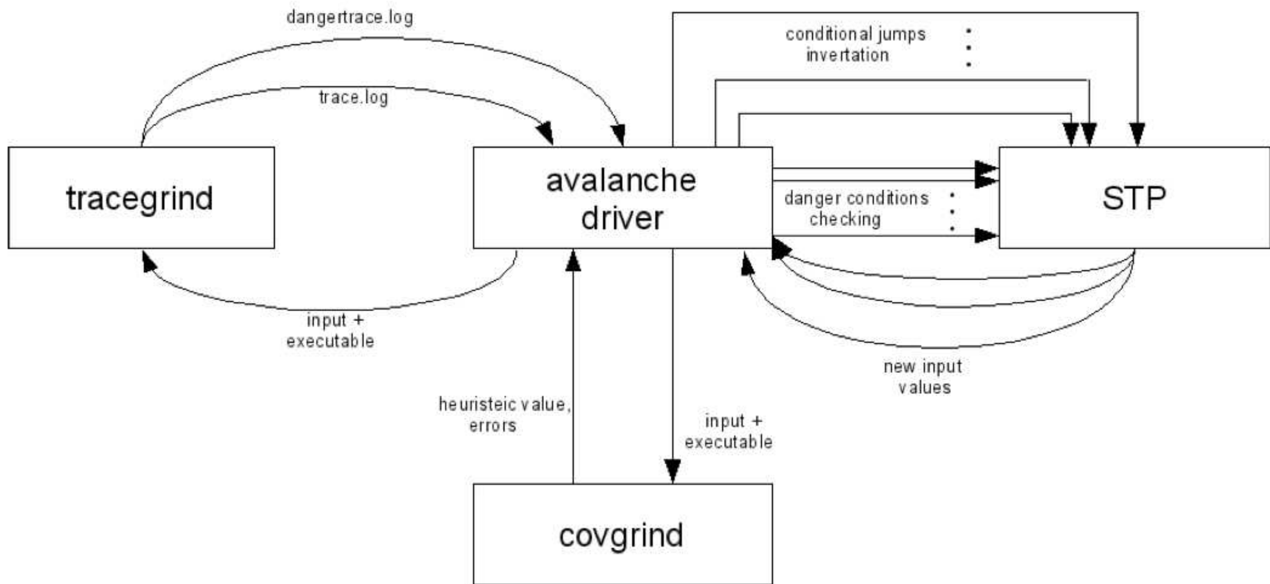


**Рис. 1.** Avalanche. Общая схема.

необходимо обнаруживать. Например, при поиске ошибок типа "разыменование нулевого указателя" любая операция чтения или записи в память может считаться потенциально опасной. В настоящий момент Avalanche отслеживает следующие операции:

- все деления (потенциальное деление на нуль);
- все операции чтения/записи в память (потенциальное разыменование нулевого указателя).

В обоих случаях использование нулевого значения в качестве операнда этих операций приведет к завершению программы (в первом случае – с исключением плавающей точки, во втором случае – с ошибкой сегментации). Поэтому условия, создаваемые Tracegrind'ом, должны отражать факт равенства делителя или адреса обращения к памяти нулю.

Рассмотрим способы решения этих задач.

### *3.1. Поток помеченных данных*

Для отслеживания появления помеченных данных в программе Tracegrind используются возможности, предоставляемые Valgrind для перехвата системных вызовов в инструментируемой программе. В частности, перехватываются следующие вызовы, ответственные за работу с файлами:

- open,
- close,
- lseek,
- read,
- map.

Имя файла, из которого программа может читать данные, должно быть передано Avalanche в качестве стартовой опции (эта опция затем передается управляющим модулем при каждом запуске Tracegrind). Tracegrind будет отслеживать только взаимодействие программы с указанным файлом, и любые данные, прочитанные из этого файла, будут считаться помеченными. Данные, прочитанные программой из других файлов, помеченными считаться не будут. На данный момент Tracegrind поддерживает только один входной файл в качестве источника помеченных данных, в дальнейшем, надеемся, это ограничение будет снято.

Поток помеченных данных отслеживается при помощи явной проверки операндов каждой инструкции внутреннего представления Valgrind.

Если какой-либо из операндов помечается. Изначально помечаются все ячейки памяти, в которые осуществляется запись при выполнении системных вызовов read и map. Затем, если какое-либо из этих значений считывается во временную переменную, то эта переменная также помечается. Использование помеченных переменных в качестве операндов различных операций (арифметических, битовых и т.д.) приведет к тому, что переменная-результат также становится помеченной. Аналогичным образом обрабатывается сохранение в память, чтение и запись в регистры. Если же помеченное значение перезаписывается (например, в регистре сохраняется константа или значение непомеченной переменной), то соответствующий объект (ячейка памяти или регистр; во временную переменную можно присвоить значение лишь один раз) больше не считается несущим помеченное значение.

Такой подход к отслеживанию потока помеченных данных в программе прост и достаточно эффективен. Однако в программе возможно появление неявных помеченных значений, т.е. таких значений, которые зависят от содержимого входного файла, но для которых нет явных присваиваний, позволяющих отследить их зависимость от прочитанных из входного файла значений. Пример:

```
int len, fd;
...
read(fd, &len, sizeof(int));
int i = 0;
for (int j = 0; j < len; j++) {
  i++;
}
```

В этом примере значение $i$ однозначно определяется содержимым файла, но отследить факт такой зависимости при помощи описанного подхода невозможно; значение $i$ будет (ошибочно) считаться непомеченным. Другим примером неявно помеченных данных может служить размер входного файла.

### *3.2. Создание STP-деклараций*

Для составления условий с целью их последующей проверки при помощи STP каждая из инструкций промежуточного представления Valgrind, оперирующая помеченными данными, переводится во входную декларацию для инструмента STP. Рассмотрим принципы такого перевода. Перевод осуществляется "на лету" во время выполнения программы при помощи ее динамического инструментирования модулем Tracegrind. Результатом работы инструментированной программы являются две трассы. Каждая трасса представляет собой текстовый файл, содержащий последовательность деклараций для STP. Содержимое трасс во многом совпадает, разница состоит в том, что в первую трассу попадают условия для проверки возможности обхода ранее не пройденных частей программы (эту трассу будем в дальнейшем называть трассой переходов и обозначать trace.log), а во вторую – условия для проверки возможности возникновения ошибок при выполнении "опасных" операций деления и обращения к памяти (будем называть ее трассой с опасными операциями и обозначать dangertrace.log). Трассы разбиваются управляющим модулем на отдельные запросы к STP. Рассмотрим составление деклараций более подробно (всюду далее, если явно не указано обратное, подразумевается, что декларации попадают в обе трассы).

#### *3.2.1. Моделирование сущностей*

Входной файл, набор регистров и адресное пространство (память) программы представляются при помощи массивов STP. Элементы всех трех массивов – восьмибитовые вектора. Массивы, моделирующие адресное пространство и входной файл, индексируются тридцатьюдвухбитными номерами, массив регистров – восьмибитными. Так выглядят декларации для массива-памяти и массива-регистров:

```
memory_0 : ARRAY BITVECTOR(32) OF BITVECTOR(8);
registers_0: ARRAY BITVECTOR(8) OF BITVECTOR(8);
```

Аналогично объявляется и массив, моделирующий входной файл, но в его названии записывается имя файла (input.txt):

```
file_input_dot_txt : ARRAY BITVECTOR(32)
OF BITVECTOR(8);
```

```
memory_1 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_0
          WITH [0hex04057000] := file_input_dot_txt[0hex00000000];
```

**Рис. 2.**

```
memory_717 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_716
             WITH [0hexbec91560] := t_40139a0_0_22[7:0];
memory_718 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_717
             WITH [0hexbec91561] := t_40139a0_0_22[15:8];
memory_719 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_718
             WITH [0hexbec91562] := t_40139a0_0_22[23:16];
memory_720 : ARRAY BITVECTOR(32) OF BITVECTOR(8) = memory_719
             WITH [0hexbec91563] := t_40139a0_0_22[31:24];
```

**Рис. 3.**

Временные переменные моделируются битовыми векторами соответствующей размерности (1, 8, 16, 32 или 64). В именовании переменной (все декларируемые объекты в соответствии с синтаксисом STP должны иметь уникальные имена) используется адрес базового блока, которому принадлежит переменная, номер переменной в блоке и номер текущего посещения блока:

```
t_409f9d8_59_2 : BITVECTOR(8);
```

Данная декларация соответствует 59-ой однобайтовой переменной из базового блока с начальным адресом `409f9d8`, причем декларация была создана во время третьего выполнения соответствующего базового блока (это соответствует последней двойке в имени переменной; посещения нумеруются с нуля).

### *3.2.2. Декларации для системных вызовов*

При перехвате системных вызовов `read` и `map` создается набор деклараций, отражающих факты равенства значений, прочитанных из файла, содержимому ячеек памяти, по которым осуществлялась запись (см. рис. 2).

Данная декларация моделирует чтение первого байта из файла input.txt в ячейку памяти с адресом `0x04057000`. В общем случае, каждый из вызовов `read` и `map` приводит к появлению цепочки таких деклараций – по числу прочитанных байт.

### *3.2.3. Декларации для инструкций, оперирующих помеченными данными*

Все инструкции также моделируются достаточно очевидным образом. Так, например, для инструкции чтения из памяти `t17 = Ldle:I8(t15)` (в переменную `t17` загружается один байт по адресу, содержащемуся в переменной `t15`, при этом ячейка по этому адресу содержит помеченное значение) будет создана декларация

```
ASSERT(t_40d1ff8_17_0=memory_712[0hex04057000]);
```

Если переменная `t15` помечена, то в трассу с опасными операциями также добавляется декларация, отражающая условие равенства `t15` нулю, а также синтаксическое указание STP для проверки выполнимости формулы `QUERY(FALSE);`:

```
ASSERT(t_40d1ff8_15_0=0hex00000000);
QUERY(FALSE);
```

Чуть большую сложность представляют инструкции, читающие или записывающие сразу несколько байт. Поскольку элементы в моделирующих массивах имеют размер в один байт, приходится, соответственно, или разбивать значение длинной переменной по кускам (при помощи операции выборки), или собирать длинное значение из отдельных байт при помощи операций конкатенации и битовой дизъюнкции. Допустим, в инструкции `STle(t22) = t0` переменная `t0` – помеченная, имеет размер 4 байта. Тогда будут созданы декларации, приведенные на рис. 3.

```
ASSERT(t_40d3cab_2_0=((0hex000000 @ registers_4[0hex00]) |
                      (0hex0000 @ registers_4[0hex01] @ 0hex00) |
                      (0hex00 @ registers_4[0hex02] @ 0hex0000) |
                      (registers_4[0hex03] @ 0hex000000)));
```

**Рис. 4.**

```
ASSERT(t_40d3cab_10_0=IF t_40d3cab_11_0=0hex00000000
                      THEN 0bin1 ELSE 0bin0 ENDIF);
```

**Рис. 5.**

```
ASSERT(t_40d3cab_0_0=BVPLUS(32,t_40d3cab_2_0,0hex00000001));
```

**Рис. 6.**

Здесь по младшим адресам записываются младшие байты из переменной `t22`, получаемые при помощи операции выборки по соответствующим битам. В случае чтения происходит обратное – длинное значение склеивается из фрагментов. Так, при обработке инструкции `t2 = GET:I32(0)` (регистр 0 содержит помеченное значение) будет создана декларация, показанная на рис. 4.

Регистру 0 в представлении Valgrind соответствует регистр `eax`, при этом регистр `al` также обозначается нулем (как младшая часть `eax`), регистр `ah` – единицей и т.д. Каждый из четырех байт получается при помощи обращения к массиву, затем прочитанные векторы конкатенируются с нулевыми векторами соответствующей длины и объединяются при помощи битовой дизъюнкции.

Арифметические и битовые операции (а также операции приведения между различными типами) из промежуточного представления Valgrind моделируются при помощи соответствующих операций STP. Для операций сравнения также используется операция `IF THEN ELSE`. Например, операция `t10 = CmpEQ32(t11,0x0:I32)` породит декларацию, показанную на рис. 5.

Операция сложения `t11 = Add32(t2,0x1:I32)` – породит декларацию, приведенную на рис. 6.

Операция знакового приведения типа моделируется при помощи аналогичной операции STP. Операции `t16 = 8Sto32(t17)` будет соответствовать декларация

```
ASSERT(t_40d1ff8_16_0=BVSOX(t_40d1ff8_17_0, 32));
```

Если в операции деления второй операнд является помеченным, то в трассу с опасными операциями добавляется декларация для отражения условия равенства делителя нулю и указание `QUERY(FALSE);` Например, по операции `t18 = DivModU64to32(0x3B9AC9F4:I64,t16)` в трассе с опасными операциями будет записана декларация:

```
ASSERT(t_4053000_16_0=0hex00000000);
QUERY(FALSE);
```

На данном этапе разработки инструментируются и приводят к созданию STP деклараций только целочисленные операции. Арифметические операции с плавающей точкой, по-видимому, поддерживаться не будут, в том числе по причине сложности их моделирования имеющимися в STP операциями.

### *3.2.4. Условные переходы*

В случае, если выполнение дошло до инструкции условного перехода, условие которого зависит от помеченной переменной, то, в зависимости от выполнения или невыполнения этого условия, создается декларация, отражающая факт истинности или ложности этой переменной. Кроме того, в трассу переходов также добавляется указание `QUERY(FALSE);` Инструкция непроизошедшего условного перехода `if (t3) goto 0x40D3CC1:I32` приведет к созданию следующих деклараций:

```
ASSERT(t_40d3cab_3_0=0bin0);
QUERY(FALSE);
```

```
ASSERT(t1=0);    ASSERT(t1=1);    ASSERT(t1=1);          ASSERT(t1=1);
QUERY(FALSE);    ASSERT(t2=0);    ASSERT(t2=1);          ASSERT(t2=1);
                 QUERY(FALSE);    ASSERT(t3=1);    ...   ASSERT(t3=0);
                                  QUERY(FALSE);          ...
                                                         ASSERT(tn=1);
                                                         QUERY(FALSE);
```

**Рис. 7.**

## 4. УПРАВЛЯЮЩИЙ МОДУЛЬ

Управляющий модуль является связующим звеном между остальными компонентами Avalanche. В его функции входит:

- координация взаимодействия между процессами остальных компонентов;
- управление обходом дерева условных переходов программы.

После завершения работы модуля Tracegrind управляющий модуль осуществляет разбор двух трасс, полученных в результате работы Tracegrind. Схематично их содержимое выглядит примерно следующим образом:

| trace.log | dangertrace.log |
|---|---|
| `ASSERT(t1=1);` | `ASSERT(t1=1);` |
| `QUERY(FALSE);` | `ASSERT(t2=1);` |
| `ASSERT(t2=1);` | `ASSERT(t_div=0);` |
| `QUERY(FALSE);` | `QUERY(FALSE);` |
| `ASSERT(t3=0);` | `ASSERT(t3=0);` |
| `QUERY(FALSE);` | `ASSERT(t_deref=0);` |
| `...` | `QUERY(FALSE);` |
| `ASSERT(tn=0);` | `...` |
| `QUERY(FALSE);` | `ASSERT(tn=0);` |

Здесь условиями вида `ASSERT(ti=1(0))` обозначены условия, отвечающие выполнению (невыполнению) условных переходов, зависящих от помеченных данных, а условиями вида `ASSERT(t_div=0)` (`ASSERT(t_deref=0)`) – условия равенства делителя (адреса обращения к памяти) нулю. Декларации, получающиеся от перевода других инструкций, опущены в целях наглядности и экономии места.

Сначала управляющий модуль осуществляет разбор трассы dangertrace.log и строит по его содержимому следующие запросы:

```
ASSERT(t1=1);    ASSERT(t1=1);
ASSERT(t2=1);    ASSERT(t2=1);
ASSERT(t_div=0); ASSERT(t3=0);
QUERY(FALSE);    ASSERT(t_deref=0);
                 QUERY(FALSE);
```

Как видим, выполнение этих запросов отражает реализацию ошибок в опасных операциях на той же трассе программы. Если в результате работы STP оказывается, что запросы выполнимы, то по значениям переменных, обращающим запрос в истину, управляющий модуль строит новое содержимое файла и запускает программу с новым файлом, чтобы проверить факт возникновения ошибки. В случае подтверждения ошибки входной файл сохраняется управляющим модулем.

Затем управляющий модуль осуществляет разбор трассы trace.log и передает STP для проверки выполнимости следующие запросы, показанные на рис. 7.

Как видим, в $i$-ом запросе первые $i-1$ условий в условных переходах остаются прямыми (т.е. с тем же исходом, что и при выполнении программы), а $i$-ое условие перехода инвертируется (т.е. если переход не был выполнен, то в запрос попадает условие `ASSERT(ti=1)`, а если переход был выполнен, то условие `ASSERT(ti=0)`). Если запрос выполним, то управляющий модуль запоминает возможное содержимое нового входного файла, позволяющего запустить приложение по новой трассе. После проверки выполнимости всех запросов у управляющего модуля имеется набор новых содержимых входных файлов.

На рис. 8 изображен фрагмент дерева условных переходов программы. На нем условные переходы изображены вершинами, нижняя исходящая дуга вершины соответствует прямому выполнению программы, верхняя – инвертированному (т.е. выполнению на данных, полученных в результате успешной проверки выполнимости соответствующего запроса к STP). Так, самому первому выполнению программы соответ-

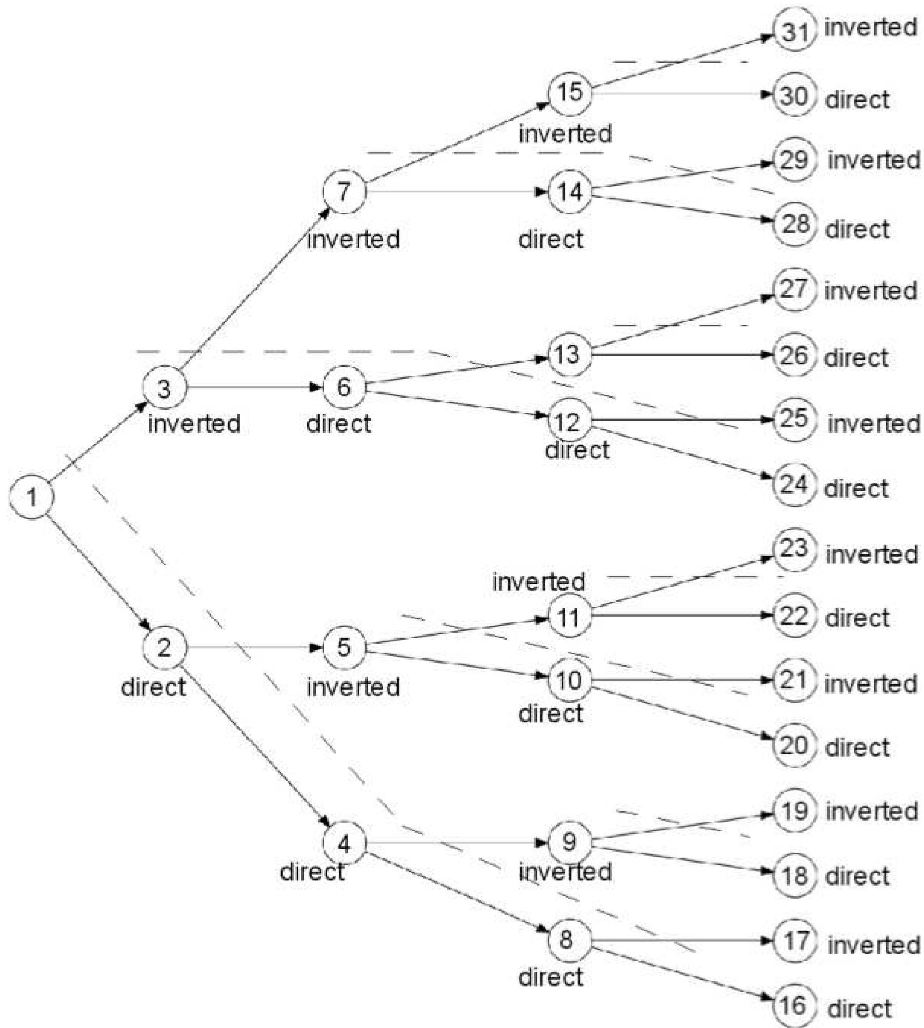


**Рис. 8.** Дерево условных переходов программы.

ствует путь 1-2-4-8-16... Допустим, после первого выполнения все запросы на инвертирование условных переходов оказались удачными, тогда на следующем шаге у управляющего модуля есть значения входных данных, чтобы пройти по веткам 1-3-6-12-24..., 1-2-5-10-20..., 1-2-4-9-18..., 1-2-4-8-17... и т.д. Поэтому первое выполнение изображено разрезающей чертой, отсекающей на дереве самую нижнюю ветку. Как уже было сказано, из имеющегося набора входных файлов выбирается тот, запуск анализируемой программы с которым приведет к покрытию наибольшего числа ранее непокрытых блоков. Для подсчета числа таких блоков управляющий модуль запускает Covgrind с анализируемым приложением и с каждым из полученных входных файлов.

Предположим, что наибольший прирост покрытия дает ветка 1-3-6-12-24... Тогда после второго исполнения анализируемой программы с Tracegrind у управляющего модуля, помимо входных файлов, полученных после первого выполнения (и еще не выбранных для исполнения), добавятся входные файлы для обхода веток 1-3-7-14-28..., 1-3-6-13-26..., 1-3-6-12-25... Из них снова будет выбран файл с наилучшим покрытием, и приложение будет снова запущено с Tracegrind, и так далее. Таким образом, осуществляется поветочный обход дерева условных переходов программы, а также накопление файлов для обхода ранее не обойденных ветвей. Легко видеть, что полный обход всего дерева потребует экспоненциального числа запусков приложения с Tracegrind ($2^{n-2}$, где $n$ – глубина дерева, или, иными словами, число условных переходов в программе, зависящих от помеченных данных).

Кроме того, каждая вершина на одной ветке также требует решения NP-полной задачи проверки выполнимости булевской формулы. Поэтому даже обработка всего одной ветки может потребовать существенного времени. Чтобы ускорить процесс обхода дерева, можно при запуске Avalanche указать в качестве одной из стартовых опций пороговое значение глубины просмотра – тогда на каждой ветке будет инвертироваться не более указанного числа условных переходов, а условные переходы, лежащие глубже указанной пороговой глубины, не будут порождать никаких запросов к STP (что соответствует отсечению нижних частей ветвей дерева).

## 5. COVGRIND: ПОКРЫТИЕ ВОЗМОЖНЫХ СОСТОЯНИЙ ПРОГРАММЫ ТЕСТОВЫМИ ПРИМЕРАМИ

Компонент Covgrind предельно прост – он создает список адресов выполненных базовых блоков, по которому управляющий модуль подсчитывает число ранее не встречавшихся блоков. Также перед началом работы Covgrind устанавливает значение таймера. Если во время выполнения анализируемой программы таймер срабатывает (т.е. истекает установленное время), то считается, что это означает наличие в программе бесконечного цикла. Значение таймера – одна из стартовых опций при запуске Avalanche. Если же происходит ненормальное завершение Covgrind (т.е. завершение по сигналу), то считается, что в программе произошла ошибка. Как в случае ненормального завершения, так и в случае зацикливания программы управляющий модуль отдельно сохраняет входной файл, с которым осуществлялся данный запуск.

## 6. РЕЗУЛЬТАТЫ РАБОТЫ AVALANCHE НА РЕАЛЬНЫХ ПРИЛОЖЕНИЯХ

### 6.1. *Рассматриваемые проекты*

Эффективность поиска ошибок при помощи Avalanche была исследована на большом числе проектов с открытым исходным кодом. Далее будут рассмотрены результаты работы на следующих 10 проектах:

**qtdump (libquicktime-1.1.3).** libquicktime – библиотека для записи и чтения файлов в формате quicktime/avi/mp4. Qtdump – обертка вокруг библиотечной функции quicktime_dump(file), отображающей разобранное содержимое (parsed contents) переданного файла.

**flvdumper (gnash-0.8.6).** gnash – GNU-проигрыватель флеш-файлов. Программа flvdumper распечатывает внутреннюю информацию flv-видеофайла.

**cjpeg (libjpeg7).** libjpeg – библиотека для обработки jpeg-файлов. cjpeg конвертирует файлы из формата bmp в формат jpeg.

**sndfile-mix-to-mono (sndfile-tools-1.02).** libsndfile – библиотека для записи и чтения файлов в форматах MS Windows WAV и Apple/SGI AIFF. Sndfile-tools – набор прикладных программ, использующих libsndfile. Sndfile-mix-to-mono преобразует многоканальный входной файл в моно-файл, соединяя все входные каналы в один.

**swfdump (swftools-0.9.0).** SWFTools – набор утилит для работы с Adobe Flash файлами (SWF-файлами). Swfdump распечатывает различную информацию об SWF-файлах, такую как содержащиеся рисунки/шрифты/звуки, перекрестные ссылки, дизассемблирует содержащийся код, и т.д.

**avibench (avifile).** avifile – проигрыватель и библиотека для работы с мультимедиа-файлами в формате AVI. Avibench используется для сбора статистики о производительности функций библиотеки avifile.

**xmllint (libxml2-2.7.6).** libxml2 – библиотека для разбора xml-файлов. Xmllint – парсер на основе libxml2.

**mpeg2dec (libmpeg2-0.5.1).** libmpeg2 – библиотека для декодирования mpeg-2 и mpeg-1 видеопотоков. Mpeg2dec – простая тестовая утилита для libmpeg2, декодирует mpeg-1 и mpeg-2.

**mpeg3dump (libmpeg3-1.8).** libmpeg3 – библиотека для декодирования сжатых MPEG-

**Таблица 1**. Статистика работы Avalanche при начальном запуске с некорректными данными

| | qtdump | flv-dumper | cjpeg | sndfile-mix-to mono | swf-dump | avi-bench | xmllint | mpeg2-dec | mpeg3-dump | speex-enc |
|---|---|---|---|---|---|---|---|---|---|---|
| число запусков | 126 | 97 | 112 | 10 | 355 | 37 | 60 | 147 | 34 | 5 |
| начальное покрытие | 2927 | 5853 | 2259 | 2395 | 2202 | 6037 | 3918 | 2503 | 2757 | 3507 |
| прирост покрытия | 1667 (57%) | 16 (< 1%) | 1188 (53%) | 821 (34%) | 2010 (91%) | 1357 (22%) | 2231 (57%) | 206 (8%) | 1885 (68%) | 32 (1%) |
| точность предсказания | 95% | 100% | 96% | 100% | 70% | 86% | 100% | 100% | 93% | 100% |
| время Tracegrind | 4% | 84% | 3% | < 1% | 10% | 3% | 2% | 4% | 5% | 1% |
| время STP | 61% | 4% | 96% | 99% | 54% | 80% | 93% | 50% | 73% | < 1% |
| время Covgrind | 34% | 11% | 1% | < 1% | 35% | 17% | 4% | 46% | 21% | 98% |
| опасные операции | 173 | 0 | 404 | 0 | 159 | 1576 | 81 | 0 | 585 | 0 |
| число дефектов | 5 | 0 | 1 | 1 | 2 | 1 | 0 | 0 | 2 | 0 |
| тестовые данные | 82 | 0 | 35 | 1 | 437 | 2 | 0 | 0 | 3136 | 24 |
| $t_{min}$ | 491 | – | 252 | 8 | 55 | 3041 | – | – | 5 | – |
| $t_{max}$ | 5827 | – | 252 | 8 | 684 | 3041 | – | – | 5 | – |

файлов. mpeg3dump – утилита, использующая libMPEG3, сбрасывает информацию или извлекает аудио в 24-битный PCM-файл.

**speexenc (speex-1.2rc1).** Speex – открытый формат сжатия файлов с записью речи. Speexenc осуществляет сжатие в формат speex из форматов WAV или несжатых файлов.

### *6.2. Параметры запуска и статистика работы*

В первом случае программы запускались на небольшом входном файле (712 байт). Содержимое файлов не соответствовало формату данных, ожидаемому приложениями, т.е. с точки зрения исследуемых приложений входной файл был сформирован неправильно. Во втором случае входные данные, напротив, были сформированы из корректных с точки зрения исследуемых приложений файлов (т.е. соответствующих ожидаемому формату) путем обрезания по первым 712 байтам (размер входного файла ограничивается с целью сокращения получаемых ограничений и ускорения последующей проверки их выполнимости). Стартовая опция глубины просмотра во всех случаях указывалась равной 100 условиям (зависящим от помеченных данных). Стартовая опция для порогового значения таймаута (для обнаружения бесконечных циклов) задавалась равной 300 секундам. На каждом из 10 указанных проектов Avalanche работал в течение 7500 секунд, затем работа Avalanche принудительно завершалась (т.е. ни разу не был осуществлен полный обход дерева условных переходов программы).

Результаты работы приведены в таблицах 1 и 2.

*Общее число запусков* – число запусков анализируемого приложения с модулем Tracegrind (это число также равно числу частично просмотренных веток дерева условных переходов программы).

*Точность предсказания* – это выраженная в процентах доля запусков, при которых выполнение шло именно по той ветке дерева, по которой оно должно было пройти в соответствии с проверкой выполнимости формулы, отражающей условие инвертированности одного из переходов по сравнению с предыдущим запуском. Сделаем несколько предположений о том, почему

**Таблица 2**. Статистика работы Avalanche при начальном запуске с частично корректными данными

| | qtdump | flv-dumper | cjpeg | sndfile-mix-to-mono | swf-dump | avi-bench | xmllint | mpeg2-dec | mpeg3-dump | speex-enc |
|---|---|---|---|---|---|---|---|---|---|---|
| число запусков | 3 | 83 | 20 | 1 | 293 | 28 | 58 | 145 | 62 | 3 |
| начальное покрытие | 3060 | 6994 | 3098 | 2497 | 2770 | 6077 | 4617 | 5131 | 3195 | 3537 |
| прирост покрытия | 103 (4%) | 569 (8%) | 311 (10%) | 0 | 2350 (85%) | 1135 (19%) | 1953 (42%) | 25 (< 1%) | 1457 (46%) | 54 (2%) |
| точность предсказания | 100% | 89% | 95% | – | 68% | 86% | 100% | 100% | 94% | 100% |
| время Tracegrind | < 1% | 58% | < 1% | < 1% | 9% | 3% | 2% | 5% | 5% | < 1% |
| время STP | 1% | 28% | 99% | > 99% | 61% | 80% | 93% | 73% | 75% | 3% |
| время Covgrind | 99% | 10% | < 1% | < 1% | 29% | 17% | 4% | 21% | 20% | 97% |
| опасные операции | 0 | 663 | 214 | 1 | 231 | 3075 | 104 | 0 | 585 | 0 |
| число дефектов | 1 | 1 | 1 | 0 | 2 | 1 | 0 | 1 | 2 | 0 |
| тестовые данные | 24 | 2 | 37 | 0 | 933 | 2 | 0 | 127 | 3136 | 25 |
| $t_{min}$ | 314 | 156 | 3 | – | 59 | 2919 | – | 12 | 1 | – |
| $t_{max}$ | 314 | 156 | 3 | – | 248 | 2919 | – | 12 | 9 | – |

показатель точности предсказания не достигает 100%.

- Приближения при переводе инструкций промежуточного представления Valgrind в декларации STP. Правила синтаксиса деклараций STP требуют, чтобы второй операнд операций бинарного сдвига был задан в виде явного числа. Если же при выполнении программы второй операнд является помеченной переменной, то при переводе приходится вместо помеченной переменной указывать ее значение, что может привести к неточному результату после проверки выполнимости полученной формулы.

- Наличие недетерминированных ситуаций при работе программ. Например, использование адресной арифметики при работе с динамической памятью может привести к появлению разных трасс даже при запуске программы на одних и тех же входных данных.

- Наличие в программе неявно помеченных данных.

*Начальное покрытие* – количество базовых блоков при первом запуске программы.

*Прирост покрытия* – суммарное количество новых базовых блоков, обойденных за время работы Avalanche (т.е. число тех базовых блоков, которые не были пройдены при первом запуске, но были пройдены при последующих запусках). В значения чисел для измерения покрытия включены не только базовые блоки самой программы, но и базовые блоки для библиотечной части приложений. Указано также процентное соотношение прироста покрытия и начального покрытия.

*Время Tracegrind*, *время STP* и *время Covgrind* – выраженное в процентах отношение времени работы каждого из компонентов Avalanche к общему времени работы.

*Опасные операции* – количество проверенных формул для поиска ошибок при выполнении “опасных” операций за все время работы Avalanche.

*Тестовые данные* – общее число созданных в процессе работы Avalanche тестовых примеров, приводящих к ошибке в программе (завершение по сигналу или бесконечный цикл).

*Число дефектов* – число уникальных ошибок, воспроизводимых при помощи созданных Avalanche тестовых данных. Причина того, что общее число тестовых примеров превосходит число уникальных дефектов, заключается в том, что один и тот же дефект может возникнуть на разных трассах программы.

$t_{min}$ – время нахождения первого дефекта (в секундах от начала работы Avalanche).

$t_{max}$ – время нахождения последнего уникального дефекта (в секундах от начала работы Avalanche).

Для qtdump запуск с некорректными данными оказался более эффективным – было найдено 5 дефектов, в то время как при запуске с частично корректными данными – только один, уже входящий в число 5 ранее найденных. Аналогична ситуация и с sndfile-mix-to-mono: запуск с частично корректными данными привел к созданию ограничений, проверка выполнимости которых заняла слишком много времени, в результате за отведенное на работу время прироста покрытия получить не удалось, следовательно, не удалось и найти дефекты. Зато запуск sndfile-mix-to-mono с некорректными данными привел к обнаружению дефекта.

Для flvdumper и mpeg2dec ситуация полностью противоположная – в результате поиска дефектов при начальном запуске с некорректными данными ошибок найдено не было. Зато второй запуск привел к успешному обнаружению дефектов. Вообще, начальный запуск приложения с ожидаемыми данными сразу увеличивает просматриваемую часть дерева условных переходов программы. Это может привести как к более быстрому обнаружению дефектов, так и к существенному замедлению работы из-за роста количества и размера проверяемых ограничений.

Для swfdump, avibench, xmllint, mpeg3dump и speexenc существенной разницы между двумя запусками обнаружено не было. Для cjpeg дефект был найден существенно быстрее при запуске с частично корректными данными.

Стоит отметить, что сравнительно большой процент времени выполнения приходится на работу компонента Covgrind. Это может быть связано, во-первых, с тем, что для измерения метрики Covgrind запускается существенно чаще, чем Tracegrind, а во-вторых, с использованием таймера для обнаружения бесконечных циклов. Если один и тот же бесконечный цикл может возникать на разных трассах программы, это приводит к значительной потере времени на прогоне приложения по каждой из этих трасс через Covgrind, поскольку при каждом запуске Covgrind ожидает срока истечения установленного таймера.

### 6.3. *Краткий обзор дефектов*

**qtdump (libquicktime-1.1.3).** Три дефекта связаны с разыменованием нулевого указателя, один – с наличием бесконечного цикла, еще в одном случае имеет место обращение по некорректному адресу (ошибка сегментации). Часть дефектов исправлена разработчиком.

**flvdumper (gnash-0.8.6).** Непосредственное возникновение дефекта связано с появлением исключительной ситуации в используемой приложением библиотеке boost (один из внутренних указателей библиотеки boost оказывается равен нулю). Поскольку само приложение не перехватывает возникшее исключение, выполнение программы завершается с сигналом SIGABRT. Дефект исправлен разработчиком.

**cjpeg (libjpeg7).** Приложение читает из файла нулевое значение и без соответствующей проверки использует его в качестве делителя, что приводит к возникновению исключения плавающей точки и завершению программы с сигналом SIGFPE. Дефект исправлен разработчиком.

**sndfile-mix-to-mono (sndfile-tools-1.02)**. Аналогично предыдущему пункту, приложение осуществляет деление на нуль. Разработчик сообщил, что дефект уже был ранее исправлен в текущей разрабатываемой версии приложения.

**swfdump (swftools-0.9.0).** Возникновение обоих дефектов связано с разыменованием нулевого указателя.

**avibench (avifile).** Разыменование нулевого указателя.

**xmllint (libxml2-2.7.6).** Дефектов не обнаружено.

**mpeg2dec (libmpeg2-0.5.1).** Аналогично cjpeg и sndfile-mix-to-mono, найденный дефект заключается в возможности деления на нуль.

**mpeg3dump (libmpeg3-1.8).** Возникновение обоих дефектов связано с разыменованием нулевого указателя.

**speexenc (speex-1.2rc1).** Непосредственных дефектов не обнаружено. Avalanche рапортует о 24 найденных дефектах из-за того, что при запуске приложения на каждом из этих файлов истекает установленный таймер (300 секунд). Однако ни в одном из случаев это не соответствует бесконечному циклу – можно считать эти сообщения ложными срабатываниями.

## 7. БЛИЗКИЕ РАБОТЫ

Общие принципы работы Avalanche (генерация входных данных для обхода новых частей программы, стратегия обхода дерева условных переходов программы и метрика прироста покрытия) аналогична инструменту SAGE, описанному в работе [2]. Однако, в отличие от Avalanche, инструмент SAGE не осуществляет целенаправленный поиск ошибок, фокусируясь только на обходе как можно большей части дерева условных переходов. В Avalanche целенаправленное воспроизведение ошибок обеспечивается за счет создания и проверки ограничений при выполнении опасных операций. К числу других заметных отличий Avalanche от SAGE стоит отнести следующее:

- SAGE создает ограничения при обработке предварительно собранной трассы выполнения приложения в виде последовательности инструкций `x86`. Avalanche осуществляет генерацию ограничений во время исполнения программы за счет ее динамической инструментации на основе Valgrind. Использование промежуточного представления Valgrind как основы для перевода в декларации STP позволяет существенно упростить генерацию ограничений. Совмещение генерации ограничений с непосредственным выполнением программы позволяет существенно ускорить работу.
- Avalanche демонстрирует существенно более высокую точность предсказания (точность предсказания SAGE не превосходит 40%).
- Avalanche позволяет задавать пороговое значение для глубины просмотра.

Схож с Avalanche и инструмент EXE, описанный в работе [3]. Существенным отличием EXE является инструментация исходного кода приложения (в отличие от инструментации исполняемого кода в случае Valgrind) и требование ручной разметки всех источников помеченных данных в программе. Преимуществом EXE является возможность более легкой генерации ограничений на большее число опасных операций, так, например, для EXE генерация условий для обнаружения факта переполнения буфера не представляет существенных затруднений (в исходном коде в явной форме всегда доступны и длина массива, и индекс обращения к массиву).

Инструмент Catchconv [6], как и Avalanche, использует связку Valgrind-STP для сбора ограничений и проверки их выполнимости. Однако Catchconv ограничивается генерацией условий для проверки возможности возникновения сравнительно узкого класса ошибок – ошибок некорректного использования знаковых и беззнаковых типов. При этом поиск ошибок осуществляется на единственной трассе, определяемой исходными входными данными, и создание входных данных для обхода других путей выполнения не осуществляется.

Существенно отличается от Avalanche инструмент Flayer [4]. Flayer также использует Valgrind для отслеживания потока помеченных данных в программе, однако Flayer не создает и не проверяет выполнимость каких-либо ограничений, инструментируя вместо этого код таким образом, чтобы условные переходы выполнялись (или не выполнялись) без проверки выполнения соответствующих условий. Такой подход, во-первых, требует человеческого контроля работы анализатора, во-вторых, делает возможным появление ложных срабатываний и, в-третьих, не приводит к созданию входных данных, демонст-

рирующих возникновение дефекта, даже в случае истинности обнаруженной ошибки.

Создание ограничений и проверка их выполнимости используется также и в статическом анализе. В работе [7] описывается инструмент статического анализа, моделирующий поток данных и управления каждой функции, а также обращения к динамической памяти при помощи булевских формул.

## 8. ДАЛЬНЕЙШЕЕ НАПРАВЛЕНИЕ ИССЛЕДОВАНИЙ

Результаты испытаний инструмента Avalanche на проектах с открытым исходным кодом демонстрируют эффективность Avalanche при поиске дефектов. Однако существенным ограничением является экспоненциальная сложность решаемых задач – проверки выполнимости булевских формул и обхода дерева условных переходов программы. Наличие подобного ограничения приводит к тому, что эффективно обнаруживаются лишь те ошибки, которые находятся близко к началу пути выполнения программы. В связи с этим наиболее перспективным направлением для дальнейших исследований представляется разработка методик, позволяющих обнаруживать ошибки и в других частях программы.

В частности, планируется исследовать возможности распараллеливания работы различных компонент Avalanche. Многие из этапов работы Avalanche полностью независимы от результатов работы друг друга. В частности, полностью независимы проверки выполнимости всех ограничений, получаемых при выполнении одной из веток дерева условных переходов программы, выполнение неперекрывающихся веток дерева и т.д. Это позволяет надеяться на возможность организации эффективной параллельной работы компонент Avalanche.

Кроме того, планируется изучить возможность проведения частичного анализа программы, желательно без существенных потерь в точности обнаружения ошибок. Было бы интересно рассмотреть различные варианты для обхода отдельных фрагментов дерева условных переходов программы. Например, можно проводить анализ отдельных функций приложения. Или использовать эвристики для определения наиболее "интересных" с точки зрения обнаружения ошибок мест программы. Рассматривается предложение о возможности комбинации статического и динамического анализа: статический анализатор может при помощи тех или иных средств выделять некоторые участки программы, в которых затем Avalanche будет осуществлять поиск ошибок.

Указанные способы позволят повысить скорость работы инструмента. Другим интересным направлением развития Avalanche является повышение точности обнаружения ошибок. На данном этапе Avalanche осуществляет целенаправленный поиск (при помощи проверки соответствующих условий инструментом STP) достаточно узкого класса ошибок – деления на нуль и разыменования нулевого указателя. Очевидно, что классов возможных ошибок в программах намного больше:

- выход за границы буфера;
- использование неинициализированных значений;
- некорректное использование значений знаковых и беззнаковых типов;
- ошибки работы с динамической памятью – утечки памяти, повторные освобождения памяти и т.д.;
- уязвимости безопасности.

Во-первых, необходимо исследовать вопрос об организации целенаправленной генерации условий для проверки возможности возникновения таких ошибок (аналогично тому, как это делается с делением на нуль и разыменованием нулевого указателя). Есть основания предполагать, что такие условия можно создавать для обнаружения ошибок работы со знаковыми/беззнаковыми значениями [6], для некоторых случаев переполнения буфера. Во-вторых, необходима разработка инструмента обнаружения таких ошибок в анализируемой программе. (Напомним, что на текущем этапе разработки Avalanche факт возникновения ошибки фиксируется лишь при ненормальном завершении программы. Если же в программе есть ошибка, которая не приводит к аварийному завершению программы, то такая ошибка обнаружена не будет.) Возможно, стоит

интегрировать Avalanche с другими инструментами Valgrind.

Кроме того, можно оптимизировать уже имеющуюся функциональность инструмента Avalanche. Можно расширить список поддерживаемых источников помеченных данных в программе. Напомним, что пока источником помеченных данных может быть единственный файл, с которым работает программа. На самом деле список таких источников намного шире – программа может работать с несколькими файлами, получать информацию через аргументы командной строки, читать значения переменных окружения. Все эти данные также стоит считать помеченными. Еще одним интересным источником помеченных данных могут являться сокеты. Поддержка сокетов в качестве источника помеченных данных позволит использовать Avalanche для анализа сетевых приложений, что, безусловно, расширит его область применения и, весьма вероятно, позволит обнаруживать новые типы дефектов.

## 9. ЗАКЛЮЧЕНИЕ

В статье описана реализация инструмента Avalanche. Рассмотрены общие принципы создания входных данных для целенаправленного воспроизведения ошибок в программе и анализа изначально недоступных частей программы. Описаны стратегия обхода дерева условных переходов программы и использующаяся эвристика при выборе следующей ветки дерева для дальнейшего анализа, а также модель преобразования промежуточного представления Valgrind для исполняемой программы в декларации STP и реализация преобразования при помощи модуля расширения Valgrind.

Приведены результаты работы Avalanche на проектах с открытым исходным кодом и обнаруженными в процессе работы дефектами. Указаны основные препятствия для повышения эффективности поиска дефектов и проведения полного анализа программ (экспоненциальная сложность алгоритма обхода дерева условных переходов программы и алгоритма проверки выполнимости ограничений), перечислены направления дальнейших исследований.

## СПИСОК ЛИТЕРАТУРЫ


1. *Nethercote N., Seward J.* Valgrind: A framework for heavyweight dynamic binary instrumentation // PLDI. 2007.
2. *Godefroid P., Levin M., Molnar D.* Automated whitebox fuzz testing // NDSS. 2008.
3. *Cadar C., Ganesh V., Pawlowski P.M., 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. New York, NY, USA, 2006. New York: ACM Press, 2006. P. 322–335.
4. *Drewry W., Ormandy T.* Flayer: Exposing application internals. First Workshop On Offensive Technologies (WOOT). 2007.
5. *Ganesh V., Dill D.* A decision procedure for bit-vectors and arrays. CAV 2007 // LNCS. V. 4590. P. 519–531.
6. *Molnar D., Wagner D.* Catchconv: Symbolic execution and run-time type inference for integer conversion errors. 2007. UC Berkeley EECS. 2007-23.
7. *Xie Y., Aiken A.* Scalable Error Detection using Boolean Satisfiability. Proceedings of POPL 2005. Long Beach, CA.