Основанный на резюме метод реализации произвольных контекстно-чувствительных проверок при анализе исходного кода посредством символьного выполнения

А.В. Дергачёв <dergachev.a@samsung.com>
А.В. Сидорин <a.sidorin@samsung.com>,
Исследовательский центр Samsung,
127018, Россия, г. Москва, ул. Двинцев, дом 12, корпус 1

Аннотация. Описывается методика, позволяющая реализовать поиск дефектов достаточно общего и произвольного вида при использовании межпроцедурного анализа методом резюме при анализе исходного кода программы на высокоуровневых языках программирования, таких как C и C++. Основное внимание уделено трудностям, возникающим при построении и применении резюме в процессе анализа исходного кода (по сравнению с анализом низкоуровневого представления программы), а также достижению гибкости метода анализа, необходимой для поиска дефектов произвольного вида.

Ключевые слова: статический анализ, символьное выполнение, межпроцедурный анализ, контекстно-чувствительный анализ, резюме, C, C++, Clang Static Analyzer

DOI: 10.15514/ISPRAS-2016-28(1)-3

Для цитирования: Дергачёв А.В., Сидорин А.В. Основанный на резюме метод реализации произвольных контекстно-чувствительных проверок при анализе исходного кода посредством символьного выполнения. Труды ИСП РАН, том 28, вып. 1, 2016 г., стр. 41-62. DOI: 10.15514/ISPRAS-2016-28(1)-3

1. Введение

Автоматизация поиска ошибок в программах и генерации тестов посредством статического анализа исходного кода является одним из методов контроля качества программного продукта, позволяющим выявлять дефекты, трудно обнаруживаемые другими традиционными методами [1, 2]. Символьное исполнение — простая и мощная методика, изначально сформулированная Кингом [3] и заключающаяся в том, что автоматический анализатор пытается

смоделировать поведение программы шаг за шагом вдоль всех возможных путей в графе потока управления, вводя для неизвестных величин символьные обозначения ("символы"), и объявляя возможные комбинации конкретных значений символов эквивалентными, если они ведут программу по одному и тому же пути исполнения. Всякое ветвление в программе, таким образом, разбивает множество "состояний программы" — возможных комбинаций значений символов — на классы эквивалентных состояний, и при последующем обходе эти классы состояний анализируются независимо. Подобное поведение метода символьного выполнения, будучи естественным, представляет собой одновременно и существенную проблему масштабируемости, поскольку каждое ветвление означает удвоение сложности дальнейшего анализа, приводя к экспоненциальному росту сложности. Улучшение масштабируемости анализа является, таким образом, важной практической проблемой, от решения которой зависит возможность производить все более и более исчерпывающий анализ все более и более сложных программ.

Символьное исполнение может применяться при статическом анализе скомпилированного исполняемого файла программы [4], текста программы в упрощенном промежуточном представлении, таком как Java-байткод [5] или LLVM [6], либо непосредственно исходного кода программы [7], а также исполняемого файла при смешанном статическом и динамическом анализе [8]. При этом символьное исполнение исходного кода, написанного на высокоуровневом языке программирования, является наиболее сложным из-за богатства языковых конструкций, поведение которых должно быть смоделировано с учетом того, что значения величин, которыми они оперируют, являются, вообще говоря, символьными; однако результаты подобного анализа являются наиболее полезными, поскольку позволяют получить наглядное и полноценное описание найденных дефектов в терминах исходного языка и исходного кода.

Межпроцедурный анализ означает возможность учитывать контекст вызова функции, благодаря чему корректно находить дефекты, не локализованные ни в одной отдельно взятой функции, но охватывающие несколько вызовов. Например, две проблемы двойного освобождения памяти в простейшем примере кода, приведенном ниже, не могут быть обнаружены без достаточно мощной реализации межпроцедурного анализа:

```
01    void foo(void *p) {
02      free(p);
03    }
04    void bar() {
05      void *q = malloc(1);
06      free(q);
07      foo(q);
```

```
08    }
09    void baz() {
10      void *q = malloc(1);
11      foo(q);
12      free(q);
13    }
```

Листинг 1. Примеры дефектов, для обнаружения которых необходим межпроцедурный анализ
Listing 1. Examples of defects, detection of which requires interprocedural analysis

При этом дефект в функции `bar()` находится в момент анализа функции `foo()` в контексте функции `bar()`, а дефект в функции `baz()` находится во время анализа самой функции `baz()`, однако правильная эмуляция вызова функции `foo()` все равно необходима для успешного обнаружения дефекта. В дальнейшем мы будем неоднократно обращаться к этому примеру.

Контекстная чувствительность, будучи необходимой для поиска более глубоких дефектов, ставит еще острее проблему масштабируемости, поскольку возможные пути выполнения, подвергающиеся анализу, существенно усложняются. Особенно проблематичным является анализ крупных программных комплексов, таких как Android [9], при поиске дефектов, затрагивающих сразу несколько компонентов анализируемой системы. Подобный анализ будет не только межпроцедурным, но и «межмодульным» — охватывающим сразу несколько единиц трансляции. Наивная реализация межпроцедурного анализа путем встраивания [10], при которой всякий вызов функции подразумевает моделирование этой функции заново с учетом контекста, является наиболее простой и в то же время наиболее плохо масштабируемой. Более перспективной выглядит подход, основанный на резюме: каждая функция моделируется единожды, по результатам анализа составляется резюме внешних эффектов функции, а при каждом моделировании вызова пересчету с учетом контекста подвергаются лишь внешние эффекты.

Идея оптимизации межпроцедурного анализа при помощи метода резюме упомянута еще в монографии [11]. Первые эффективные реализации метода появились не столь давно. В частности, поиску дефектов методом символьного выполнения бинарного кода и промежуточного представления посвящены работы [12, 13]. В работах [14, 15] описывается методика поиска отдельно взятых видов дефектов при помощи символьного исполнения исходного кода методом резюме. В настоящей работе описывается методика, позволяющая реализовать поиск дефектов достаточно общего и произвольного вида в условиях реализации межпроцедурного анализа методом резюме, при анализе высокоуровневого исходного кода программы. Основное внимание будет уделено специфике построения и применения

резюме при анализе исходного кода, которое оказывается существенно более сложным, чем в случае анализа низкоуровневого кода, а также достижению гибкости метода анализа, необходимой для поиска дефектов произвольного вида.

В качестве примера готового фреймворка, на который мы будем ориентироваться при описании нашей реализации, мы будем использовать Clang Static Analyzer [16] (далее CSA) — статический анализатор исходного кода, созданный на базе компилятора Clang. В CSA реализован метод символьного выполнения исходного кода программы на языках C, C++ и Objective C, чувствительный к путям выполнения и потоку данных, и реализован межпроцедурный анализ методом встраивания. В рамках настоящей работы на базе CSA был реализован межпроцедурный анализ методом резюме.

2. Терминология и особенности CSA

Как упоминалось выше, символьное выполнение подразумевает исследование всех возможных путей сквозь граф потока управления. В случае CSA это означает, что основным объектом анализа является граф выполнения (реализуемый классом `ExplodedGraph`), состоящий из всех пройденных до настоящего момента путей графа потока управления (CFG). Вершинами графа выполнения являются пары, состоящие из точки выполнения (`ProgramPoint` — элемент блока CFG, конкретный оператор программы) и состояния программы (`ProgramState` — класс эквивалентных с точки зрения символьного выполнения возможных состояний программы). Ребро графа выполнения между вершинами A и B означает, что исполнение оператора, соответствующего точке выполнения вершины A, корректирует состояние точки A до состояния точки B и переводит программу в точку выполнения вершины B. Граф выполнения, вообще говоря, не является деревом, поскольку иногда можно двумя различными путями дойти до одной и той же точки в одном и том же состоянии. Однако для простоты мы будем пользоваться терминологией, которую обычно применяют к деревьям, например, называть пути выполнения ветвями графа, а вершины, не имеющие исходящих ребер — листьями.

С точки зрения CSA анализ представляет собой процедуру построения `ExplodedGraph`'a. CSA имеет каркасную структуру, в которой отдельные проверяющие модули (`Checkers`) реализуют различные виды проверок, а их выполнением управляет ядро системы. Проверяющие модули активно влияют на построение `ExplodedGraph`'a, подписываясь на определенные события, происходящие при его построении (такие как анализ точек программы заданного типа), уточняя состояние программы, отсекая недостижимые ветви выполнения, выдавая отчеты о найденных дефектах.

Состояние программы содержит срез состояния памяти (`Store`), реализующей чувствительность к потоку данных и приписывающей конкретным или символьным регионам памяти конкретные или символьные значения, а также среду выполнения (`Environment`), приписывающую символьные значения текущим выражениям в коде.

Также оно содержит множества возможных значений символов: в соответствии с методикой символьного выполнения, при моделировании оператора ветвления в графе выполнения появляются вершины, соответствующие ветвям исполнения, и состояния в них различаются ограничениями на символьное значение условия: на одной ветви условие предполагается истинным, а на другой — ложным, что и отражается в ограничениях. Различия в значениях символов, не влияющие на выбор ветви исполнения, считаются несущественными. Если накладываемые ограничения противоречат уже имеющимся в состоянии ограничениям на тот же символ (множества значений, описывающие эти ограничения, не пересекаются), то соответствующая ветка недостижима, и анализ по ней не производится.

Третий важный элемент состояния программы — нетипизированное хранилище данных (`GenericDataMap`, далее GDM), принадлежащих проверяющим модулям. Состояния программы, в которых проверяющие модули сохранили различные данные, считаются разными; однако ядро CSA не может точнее судить о содержимом GDM. Одно из основных применений этого механизма заключается в возможности производить анализ графа состояний объектов (“typestate analysis”, [17]). К примеру, проверяющий модуль, ищущий дефекты типа двойного освобождения памяти, подобные приведенным на листинге 1, может хранить в GDM состояние всех указателей на выделенную память (освобождены или нет), и состояния с освобожденным и неосвобожденным указателем будут считаться различными при прочих равных.

3. Составление резюме

3.1 Предварительные замечания

Резюме функции должно содержать достаточно информации для того, чтобы достаточно точно — либо, во всяком случае, достаточно «консервативно» (без добавления недостаточно ненадежных предположений о состоянии программы) — смоделировать влияние вызова функции на дальнейший анализ. Основной смысл резюме — в том, чтобы смоделировать лишь существенные изменения состояния, и не моделировать изменения, не имеющие внешнего эффекта, такие как, например, изменения состояния локальных переменных.

Реализации межпроцедурного анализа методом резюме предполагает однократное символьное выполнение каждой функции вне контекста. В

случаях, когда при составлении резюме встречается вызов другой функции, резюме которой еще составлено не было, анализ текущей функции прерывается на составление резюме вызываемой функции. В случае возникновения рекурсии, когда требуется применить резюме функции, построение которого на момент вызова начато, но не закончено, вызов функции приходится осуществить «консервативно» — без применения межпроцедурного анализа.

3.1 Особенности реализации

В случае CSA к внешним эффектам функции, требующим моделирования, относятся:

1. Наложения ограничений на символы. Если в процессе символьного выполнения функции выясняется, что различные множества значений функции направляют ее выполнение по разным путям и приводят к разным внешним эффектам, то в соответствии с методом символьного выполнения анализ вызывающей функции должен также разделить имеющийся `ProgramState` на соответствующие подклассы. Это означает, что резюме должно содержать информацию о разных ветвях функции, и все ветви должны рассматриваться независимо.
2. Для каждой ветви резюме должна храниться информация о символьных величинах, записываемых функцией в различные регионы памяти, а также о возвращаемом значении.
3. Проверяющие модули должны иметь возможность оставить в резюме пометки произвольного содержания, чтобы иметь возможность смоделировать свое влияние на GDM в процессе выполнения функции.

Хотя и возможно упростить резюме функции до содержимого пометок, позволяющих смоделировать эти три вида внешних эффектов, сама по себе процедура упрощения в значительной степени оказывается излишней. Поэтому, как только функция проанализирована, *мы будем понимать под резюме функции сам законченный* `ExplodedGraph` *этой функции.* Этот тривиальный способ составления резюме позволяет не тратить время на содержательное упрощение результатов анализа, хотя и подразумевает хранение всех графов всех функций, что увеличивает потребление памяти. Более того, вся информация, существенная для моделирования вызова функции, содержится в состоянии программы, соответствующем листу каждой ветви графа: информация, необходимая для наложение ограничений на символы, содержится в виде самих ограничений на символы, информация о записях во внешние регионы памяти содержится в окончательном `Store`, а проверяющие модули могут по ходу составления резюме сохранять в `ProgramState` свои пометки, и эти пометки будут доступны в финальном состоянии.

Есть и еще одно преимущество в хранении полного графа выполнения: полный путь выполнения позволит выдать более подробный отчет о найденном дефекте, как будет показано ниже. Тем не менее, вершины графа, не являющиеся листьями или ветвлениями, ссылки на которые не хранятся нигде в анализаторе или проверяющих модулях, можно удалить. Кроме того, потребление памяти можно заметно сократить, очищая структуры состояния программ в узлах результирующего графа от более ненужных записей. Каждое из этих упрощений на практике приводит к снижению потребления памяти примерно на 10%. Чтобы гарантировать наличие необходимой информации в окончательном состоянии программы, при анализе методом резюме приходится отключить ряд механизмов сборки мусора: так, даже если символ больше нигде не хранится в данном состоянии программы, удалить ограничения на него при построении резюме нельзя, хотя при обычном анализе методом встраивания это допустимо.

4. Применение резюме

Применение резюме происходит в момент вызова функции, для которой имеется построенное резюме, то есть построен готовый `ExplodedGraph`, быть может допустимым образом упрощенный. Процедура применения резюме включает в себя несколько этапов; задача этой процедуры — для каждой ветки резюме (листа графа выполнения вызываемой функции) корректным образом объединить информацию, содержащуюся в резюме, с информацией, содержащейся в текущем состоянии программы, то есть получить своего рода композицию этих двух состояний программы. Слиянию подвергается вся информация, содержащаяся в состоянии программы, хотя часть информации, хранящейся в резюме, может быть в этом процессе признана несущественной и пропущена. Информация об ограничениях, наложенных на символы, и о значениях, записанных в регионы памяти, будет перенесена в новое состояние ядром анализатора. Информация в GDM, имеющая смысл только для проверяющих модулей, будет в процессе применения резюме перенесена в новое состояние самими проверяющими модулями, отвечающими за нее. Это важное и неизбежное соображение означает, что в отличие от анализа методом встраивания, анализ методом резюме в качестве платы за эффективность и масштабируемость подразумевает существенную поддержку со стороны проверяющих модулей, приводящую, вообще говоря, к усложнению логики модулей и дополнительным трудностям при реализации новых проверок.

4.1 Актуализация символьных величин

Одной из основных трудностей при применении резюме является пересчет символов из контекста вызываемой функции ("формальных" символьных величин) — из той формы, в которой они записаны в резюме — в

"актуальные" с учетом контекста вызывающей функции символьные величины. Эту процедуру мы назовем актуализацией символьных величин. Она устанавливает соответствие между различными символами, возникшими при анализе различных функций. Актуализация символьных величин требуется на всех этапах применения резюме.

В CSA реализована подробная иерархия классов символьных величин, отражающая их применение к описанию разнообразных объектов языка высокого уровня. Помимо конкретных величин, имеющих известные численные, строковые или структурные значения, анализатор рассматривает собственно символы, которыми в процессе анализа обозначаются неизвестные численные значения, и регионы памяти, являющиеся единицами модели памяти CSA [18] и используемые как для хранения текущего состояния памяти в `Store`, так и для изображения неизвестных значений указателей.

Так, например, объявлению параметра функции в каждом контексте вызова соответствует регион памяти типа `VarRegion`, построенный по ссылке на объявление переменной в синтаксическом дереве программы. Здесь «контекст вызова» (`StackFrameContext`) может означать либо начало анализа, либо контекст на момент вызова подфункции. В начале анализа содержимое региона, соответствующего параметру, неизвестно, и обозначается специальным символом типа `SymbolRegionValue`, содержащим ссылку на данный `VarRegion`. Далее, если, например, параметр является указателем, то регион памяти, на который он указывает, будет «символьным регионом» (`SymbolicRegion`), хранящим ссылку на символ типа `SymbolRegionValue`, обозначающем значение этого указателя. А если с данным регионом в дальнейшем обращаются как с массивом, то регион элемента этого массива будет иметь класс `ElementRegion`, и хранить ссылку на объемлющий `SymbolicRegion` и на величину индекса элемента, причем последняя также может быть не только конкретной, но и символьной. Подобная иерархия отражает разнообразие конструкций высокоуровневого языка программирования и является существенно более сложной, чем требуется для анализа низкоуровневого представления программы, для которого обычно достаточно оперировать простыми адресами и сдвигами в памяти, как в [4].

Процедура актуализации принимает на вход два объекта — состояние программы в контексте вызывающей функции и символьную величину, которую требуется актуализировать. Если актуализируемая величина содержит в своем определении ссылки на другие величины, они рекурсивно подвергаются актуализации. Таким образом, актуализация представляет собой альфа-переименование символьной величины по некоторому конкретному шаблону. Затем процедура актуализации пытается восстановить символьную величину, являющуюся аналогом исходной, но состоящую из актуализированных подвеличин. Требуемые шаги при этом сильно различаются в зависимости от класса исходной величины. Как будет показано

ниже, тип актуализированной величины не обязан совпадать с классом исходной. Однако некоторые инварианты сохраняются: так, величина типа lvalue всегда актуализируется в lvalue, и аналогично для rvalue.

Другими словами, альфа-переименование подразумевает замену некоторых символьных величин на другие и последующий пересчет всех зависящих от них символьных величин. Говоря упрощенно, под актуализацией мы понимаем частный случай альфа-переименования, в котором *замене подвергаются регионы памяти в пространстве стековых аргументов функции* (сюда включаются `VarRegion`'ы от переменных-параметров и особый регион `CXXThisRegion`, содержащий значение указателя `this` в C++-коде), *и заменяются они на такие же регионы с откорректированным контекстом вызова* (`StackFrameContext`). Все остальные символьные величины, зависящие от этих регионов (такие как `SymbolRegionValue` от этих регионов), будут рекурсивно переименованы, а не зависящие — оставлены без изменения.

Продемонстрируем смысл и метод актуализации символьного значения на примере.

```
01    void foo(int x) {
02      if (x == 0) {}
03    }
04    void bar() {
05      foo(1);
06      foo(y);
07    }
```

Листинг 2. Пример вызова функции, для моделирования которого необходима актуализация символьных величин

Listing 2. Example of a function call, for which modeling an actualization of symbolic values is necessary

Резюме функции `foo()`, описанной в листинге 2, содержит две ветви. Их финальные состояния различаются ограничениями на символ, обозначающийся в терминах CSA как `reg_$0<x>`. Это символ класса `SymbolRegionValue`, обозначающий неизвестное значение, содержащееся в регионе памяти, соответствующем переменной-параметру `x`, в момент начала анализа, то есть начала построения резюме. В одной из ветвей на `reg_$0<x>` наложены ограничения `[0, 0]`, а в другой — `[INT_MIN, -1] U [1, INT_MAX]`. Регион памяти, соответствующий переменной `x`, является в терминах CSA регионом типа `VarRegion`.

На строке 05 производится вызов функции `foo()` с аргументом `1`. При моделировании этого вызова необходимо придать смысл ограничению на `reg_$0<x>`, различающему состояния двух ветвей резюме. Однако сам символ ни разу не встречается в контексте вызывающей функции. Регион

переменной `x` с точки зрения вызывающей функции есть ячейка стека, в которой передан параметр `1` — в этот момент и происходит собственно переименование одного региона параметра на другой: момент начала анализа, заложенный в определении `reg_$0<x>`, оказывается с точки зрения вызывающей функции моментом вызова `foo()`; на момент вызова функции `foo()` в регионе переменной `x` записано значение `1`; следовательно, правильным результатом актуализации символа `reg_$0<x>` при вызове функции `foo()` на строке `05` является конкретное значение `1`. Из этого примера ясно и общее правило: чтобы актуализировать символ типа `SymbolRegionValue`, нужно актуализировать соответствующий ему регион, и взять значение, записанное в этот регион. Отметим, что результатом актуализации того же самого символа `reg_$0<x>` при вызове функции `foo()` на строке `06` является символ `reg_$0<y>`. Более того, результатом актуализации региона `x` на строках 05 и 06 являются разные регионы, соответствующие одному и тому же (с точки зрения абстрактного синтаксического дерева программы) параметру `x` в разных пространствах (`MemorySpace`) аргументов разных вызовов функции `foo()`.

Приведем другой пример, демонстрирующий рекурсивный обход символьной величины в процессе актуализации.

```
01    void foo(const char *x) {
02      if (x[2] == 'a') {}
03    }
04    void bar(const char *y) {
05      foo(y);
06      foo("aaa");
07    }
```

Листинг 3. Пример вызова функции, для моделирования которого необходимо рекурсивное альфа-переименование символьных вечилин

Listing 3. Example of a function call, for modelling of which a recursive alpha-renaming of character values is necessary

При анализе кода из листинга 3 при построении резюме функции `foo()` возникает указатель `reg_$0<x>`, указывающий на заренее неизвестную строку символов. Функция затем разветвляется в зависимости от значения, записанного в один из регионов элементов этой строки. Символ, относительно которого производится ветвление, представляет собой `SymbolRegionValue` от `ElementRegion` от `SymbolicRegion` от `SymbolRegionValue` от параметра `x`. В процессе альфа-переименования при моделировании вызова на строке 05 параметр `x` будет принимать значение `reg_$0<y>`, по нему будет построен другой символьный регион, в котором, в свою очередь, в соответствующем элементе будет записано другое значение. На строке 06 результатом актуализации величины `reg_$0<x>` будет неизвестное значение

адреса строкового литерала `"aaa"` в памяти, результатом актуализации символьного региона — сам регион строкового литерала, а значения его элемента — конкретная (не символьная) величина `'a'`. На рисунках 1 и 2 изображена соответствующая этим двум процедурам иерархия регионов памяти. На рисунке 2 видно, что актуализированный регион может занять другой `MemorySpace` и перестать быть символьным.

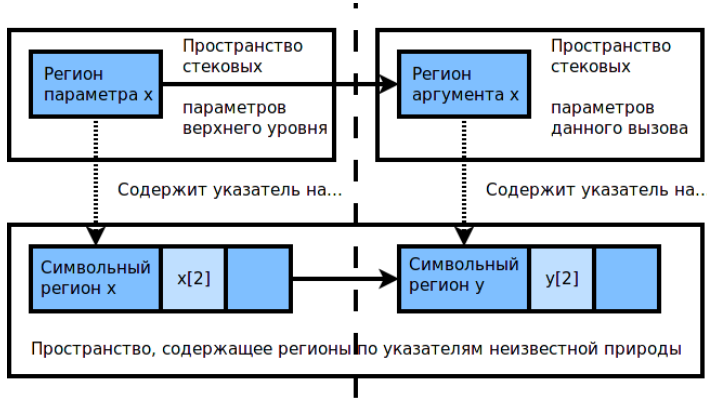

Рис. 1. Процесс рекурсивной актуализации символьного значения `x[2]` *в листинге 3 на строке 05*

Fig.1. The process of recursive actualization of character values x [2] on line 05 of Listing 3

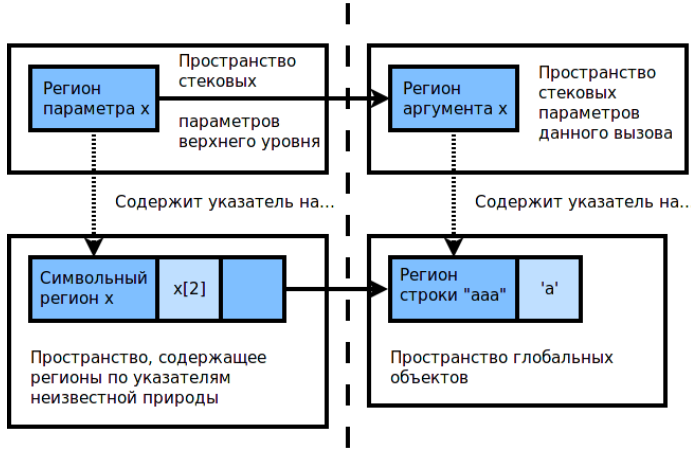

Рис. 2. Процесс рекурсивной актуализации символьного значения `x[2]` *в листинге 3 на строке 06*

Fig.2. The process of recursive actualization of character values x [2] on line 06 of Listing 3

Из приведенных примеров ясно, как описывается процедура актуализации в общем виде. Регионы памяти глобальных и статических переменных не актуализируются — остаются неизменными. Для регионов памяти, соответствующим параметрам функций, актуализация означает перемещение их в соответствующий `MemorySpace`. Подрегионы, такие как регион поля структуры или элемента массива, актуализируются посредством актуализации их базового региона и вычисления в нем аналогичного подрегиона. Регионы, соответствующие символьным указателям, такие как регионы на куче или регионы вокруг указателей неизвестного в контексте вызываемой функции происхождения, актуализируются посредством актуализации символа, лежащего в их основе; при этом они в результате актуализации могут перестать соответствовать символам, то есть стать конкретными — скажем, если происхождение символьного указателя было неизвестным в контексте вызываемой функции, но в контексте вызывающей функции этот указатель актуализировался в адрес известной переменной.

Конкретные значения не подвергаются актуализации. Символы подвергаются актуализации в соответствии с их назначением: для актуализации любого символа требуется получить величину, смысл которой передается назначением символа. Так, символ, которым обозначена неизвестная протяженность региона памяти (`SymbolExtent`), должен актуализироваться в протяженность актуализированного региона памяти, известную или неизвестную. Символ, обозначающий результат арифметической операции над другими символами (такой как `SymSymExpr`), после актуализации представляет собой результат этой же арифметической операции над другими символами — актуализированными левой и правой частью. Наконец, актуализация символов, смысл которых определяется проверяющими модулями (`SymbolMetadata`), неизбежно возлагается на сами проверяющие модули.

Особое внимание следует уделить актуализации символов, созданых для обозначения прочих неизвестных величин, таких как возвращаемое значение функции, тело которой недоступно для анализа (`SymbolConjured`). Эти символы нумеруются целыми числами, позволяющими их различать между собой, но по существу не несут никакой другой полезной информации о своем происхождении; при их актуализации требуется перенумеровать эти символы, чтобы они не конфликтовали с другими уже занятыми номерами.

Важно же здесь то, что при межпроцедурном анализе методом встраивания смысл (класс) символа, в отличие от смысла региона, несет сравнительно мало полезной информации. Однако для метода резюме иерархия символов жизненно необходима именно в связи с процедурой актуализации: так, в примере из листинга 2 мы не можем позволить себе потерять информацию о том, что `reg_$0<x>` является значением параметра `x`, а не просто неизвестной величиной. Таким образом, применение символов типа `SymbolConjured`, не хранящих информацию о своем происхождении, в

случае анализа методом резюме сильно ограничено и должно сводиться к случаям, когда содержательной актуализации заведомо не требуется. Другими словами, *требование возможности осуществить процедуру актуализации накладывает жесткие требования на иерархию символов*. К счастью, иерархия символов CSA претерпела лишь одно изменение: был введен символ, обозначающий адрес региона памяти, в который будет актуализирован символьный указатель, если в контексте вызываемой функции он указывает на известный регион. Без этого изменения иерархия символов CSA была недостаточно гибкой для отражения результата актуализации этого символа. В остальном можно утверждать, что иерархия символов CSA допускает актуализацию, при помощи техники, описанной выше. Описание в общем виде условий на иерархию символьных величин, необходимых и достаточных для проведения актуализации, выходит за рамки настоящей работы.

4.2 Применение резюме на стороне ядра анализатора

С точки зрения ядра анализатора применение ветки резюме выполняется следующим образом. Прежде всего анализу подвергаются ограничения на символы, которые однозначно определяют данную ветку. Всякое ограничение представляет собой пару из символа и множества принимаемых им конкретных значений. Символы, соответствующие ограничениям, хранящимся в резюме, подвергаются актуализации. Полученные в результате актуализации символы могут быть ограничены в контексте вызываемой функции; в противном случае будем считать их множеством их значений весь диапазон возможных значений их типа. Если множество значений символа в контексте вызываемой функции не пересекается с множеством значений актуализированного символа в контексте вызывающей функции, то ветка считается недостижимой и анализ этой ветки не производится; в противном случае новое состояние будет содержать ограничение актуализированного символа до пересечения этих двух множеств.

Затем вычисляется возвращаемое значение вызываемой функции. Это вычисление сводится к актуализации сохраненного в резюме возвращаемого значения.

Далее производится обращение к проверяющим модулям с целью слияния GDM резюме и GDM текущего состояния. Эта процедура скрыта от ядра анализатора. Проверяющие модули часто хранят в GDM символьные величины; на этом этапе они должны будут актуализировать эти величины и перенести в GDM нового состояния. Также в этот момент проверяющие модули могут произвести проверки, обнаружить дефекты в программе и выдать отчет о дефекте. Проверяющим модулям доступно возвращаемое значение функции, вычисленное на предыдущем шаге, а также множества значений всех возможно интересующих их символов, таким образом

запустить проверяющие модули раньше не представляется возможным. Поскольку проверяющие модули могут разветвить состояние на этом шаге, дальнейшие шаги выполняются отдельно для каждой вновь полученной ветки.

На последнем этапе происходит моделирование записей в глобальные переменные. Анализатор перебирает хранящийся в резюме `Store`, анализируя каждую пару, состоящую из региона памяти и связанной с ним символьной величины. Если регион является локальным для вызываемой функции, то он пропускается как несущественный. В противном случае он актуализируется, связанное с ним символьное значение также актуализируется, и связывается с актуализированным регионом в новом состоянии. Актуализация всех величин на этом шаге производится относительно одного и того же состояния; благодаря этому актуализация каждой связи в `Store` не влияет на актуализацию других связей. Это также объясняет, почему слияние `Store` выполняется последним. Так, при актуализации `SymbolRegionValue` необходимо иметь в текущем `Store` связанное с актуализированным регионом значение, записанное там до вызова вызываемой функции, а не после вызова.

4.3 Применение резюме на стороне проверяющих модулей

С точки зрения проверяющего модуля применение резюме является лишь еще одной функцей обратного вызова, позволяющей влиять на текущий анализ. В процессе применения резюме модулям доступна текущая вершина графа текущего анализа и конечная вершина графа резюме, соответствующая применяемой в этой вершине ветке. Как и в случае других функций обратного вызова, в результате применения резюме проверяющий модуль может никак не изменить текущее состояние, либо продолжить текущий анализ, добавив в него 0 (прервав тем самым анализ), 1 или более новых вершин, соотвествующих откорректированным им состояниям программы.

Проверяющий модуль также имеет возможность исследовать GDM состояния, сохранённого в конечной вершине ветви резюме. Это означает, что модуль имеет все необходимое для записи и воспроизведения всех внешних эффектов функции, проявляющихся во влиянии этой функции на содержимое раздела GDM, принадлежащего этому модулю: он может сохранить в состоянии программы подробный журнал внешних эффектов, а затем при применении резюме получить доступ к такому журналу и воссоздать все необходимые эффекты. Это позволит проверяющему модулю корректно продолжить анализ после вызова функции.

Проверяющий модуль также может выдать срабатывания непосредственно во время моделирования вызова функции, то есть во время применения резюме. Это связано с тем, что в отсутствие контекста проверяющий модуль обладает меньшим количеством информации, и не может сделать вывода о наличии дефекта, но в контексте вызывающей функции необходимая дополнительная информация может появиться.

В качестве мотивирующих примеров рассмотрим применение резюме функции `foo()` при анализе функций `bar()` и `baz()` в листинге 1 в присутствии проверяющего модуля, реализующего поиск двойных освобождений памяти. Функция `foo()` сама по себе не содержит дефекта. Однако, во время моделирования вызова функции `foo()` при анализе функции `bar()` должен быть выдан отчет о дефекте: функция `foo()` совершает повторное освобождение памяти, уже освобожденной ранее. Освобожденность памяти ранее в данном случае как раз и является дополнительной информацией, позволяющей судить о наличии дефекта. При анализе функции `baz()`, напротив, ни сама функция `foo()`, ни вызов функции `foo()` в контексте не содержат дефекта, однако функция `foo()` содержит внешний эффект — освобождение памяти, которое должно быть корректно промоделировано, иначе обнаружение дефекта при дальнейшем анализе оказывается невозможным.

Таким образом, проверяющий модуль, применяя резюме `foo()`, должен сделать две вещи: воссоздать внешние эффекты и произвести отложенные проверки. Проверяющий модуль может реализовать это, сохраняя в состоянии программы журнал всех операций выделения и освобождения памяти с символьными значениями указателей на эту память. В момент проведения каждой такой операции модуль будет запоминать эту операцию в журнале. При применении резюме модуль будет исполнять операции из журнала, извлекаемого из состояния, сохраненного в резюме, в том же порядке. Под исполнением операции будет пониматься, во-первых, проверка корректности операции и выдача сообщения о дефекте в случае некорректности, во-вторых учет эффекта операции на текущее состояние программы, то есть проставление отметок об освобожденности или неосвобожденности тех или иных указателей, и в-третьих заполнение журнала операций вызывающей функции, чтобы потом не потерять эту информацию при применении резюме вызывающей функции в контексте какой-либо третьей функции, в котором она сама станет вызываемой. Таким образом, решение о срабатывании может откладываться многократно. Заметим также, что журнал нельзя существенно упростить. Так, его нельзя сделать неупорядоченным, поскольку, например, разница между (`free()`, `free()`, `malloc()`) и (`free()`, `malloc()`, `free()`) слишком существенна.

Поддержка резюме была реализована для нескольких различных проверяющих модулей, отвечающих за нахождение таких дефектов, как целочисленные переполнения, запись в константную память, выход за границы массива, проблемы многопоточности, ошибки при работе с исключениями и корректность работы с вводом-выводом в файловые потоки. Во всех случаях она свелась к последовательному применению вышеописанной методики.

5. Составление отчета о дефекте

Одной из полезных функций CSA является составление подробного отчета о дефекте, показывающего не только место в коде, в котором происходит срабатывание проверяющего модуля, но и путь выполнения программы, предшествовавший срабатыванию. Проверяющие модули могут также активно участвовать в составлении отчета о дефекте, отмечая интересующие их события вдоль пути выполнения. Данный механизм существенно увеличивает полезность отчетов.

При анализе методом встраивания составление отчета о дефекте сводится к подъему от вершины графа выполнения, в которой произошло срабатывание, к корню графа. При проходе через вершину, в которой происходит интересное событие, такое как ветвление, вызов функции, или событие, важное с точки зрения проверяющего модуля, на позицию в коде, которая соответствует данной вершине, ставится дополнительное диагностическое сообщение.

Анализ методом резюме усложняет эту процедуру, и требует дополнительной, на этот раз сравнительно простой, поддержки на стороне проверяющих модулей. Вместо подъема по одному графу выполнения происходит склеивание пути из ветвей нескольких графов выполнения. Имеют место несколько возможных случаев.

Так, если путь выполнения обрывается в процессе применения резюме, как при анализе функции `bar()` в листинге 1, то отрезок пути от вершины, в которой производится отложенное срабатывание, до корня графа резюме добавляется в отчет о срабатывании в качестве конечного отрезка пути. При этом вершина, в которой производится отложенное срабатывание (в нашем примере это момент вызова `free()` в функции `foo()`), известна лишь проверяющему модулю; поэтому, выдавая отчет о дефекте в момент применения резюме, проверяющий модуль должен указать, в какой именно вершине резюме произошло срабатывание. Если отложенное срабатывание является многократно отложенным, то проверяющий модуль вынужден предоставить список конечных вершин во всем стеке функций.

Если же требуется построить отчет о дефекте сквозь промежуточную функцию, резюме которой было полностью применено в течение анализа, подобно тому, как отчет о дефекте в функции `baz()` в листинге 1 должен быть проложен сквозь функцию `foo()`, то вся ветвь резюме вызываемой функции целиком встраивается в отчет. Само встраивание отрезка пути не требует дополнительной поддержки на стороне проверяющих модулей. Однако, разумеется, дополнительный механизм, отвечающий за выдачу дополнительных диагностик, должен быть модифицирован с учетом особенностей анализа методом резюме. В нашем примере это требуется, поскольку вызов функции `free()` в `foo()` интересен. В этом случае вершины, содержащие интересные события, запоминаются в резюме, и при построении отчета при прохождении через эти вершины выдается

дополнительная диагностика. Запоминать весь стек вершин в этом случае не требуется.

6. Оценка масштабируемости метода.

Предварительный анализ практической эффективности и масштабируемости метода резюме по сравнению с методом встраивания производился посредством прогона анализа всей кодовой базы платформы Android [9].

Практическим показателем эффективности анализа является количество найденных дефектов. В данном случае исследовалось количество срабатываний проверяющих модулей, для которых был реализована поддержка метода резюме; доля реальных дефектов («истинных срабатываний») в обоих случаях составляла около 80%, независимо от метода анализа. Дефекты считаются одинаковыми, если выдается сообщение такого же вида в той же строчке кода; если один и тот же дефект найден в процессе анализа несколько раз, то он считается за один.

Табл. 1. Результаты измерения производительности метода резюме
Table 1. The results of performance measurement of the summary method

Ограничение max-nodes	2000	4000	8000	16000	32000	64000	128000
Время анализа	0:02	0:05	0:12	0:25	0:45	1:31	2:53
Число срабатываний	1457	1591	1696	1773	1834	1886	1911
Срабатываний в секунду	12.14	5.30	2.35	1.18	0.68	0.34	0.18
Корректных срабатываний	82%	83%	82%	83%	83%	82%	81%

Табл. 2. Результаты измерения производительности метода встраивания
Table 2. The results of performance measurement of the incorporation method

Ограничение max-nodes	8000	16000	32000	64000	128000	256000	512000
Время анализа	0:04	0:10	0:21	0:43	1:23	2:44	5:20
Число срабатываний	1506	1616	1703	1771	1828	1895	1933
Срабатываний в секунду	6.28	2.69	1.35	0.68	0.36	0.19	0.10
Корректных срабатываний	82%	82%	82%	81%	81%	81%	81%

В таблицах 1 и 2 приведены результаты измерения эффективности, производительности и масштабируемости анализа по этим двум показателям. Различные столбцы соответствуют изменениям основного параметра, контролирующего в CSA соотношение между скоростью анализа и покрытием путей выполнения — ограничением “max-nodes” на максимальный размер одного графа выполнения; это ограничение в случае метода резюме ограничивает размер графа резюме, а в при анализе методом встраивания — размер одного графа с учетом встроенных подграфов. Это означает, что при одинаковых значениях max-nodes анализ методом резюме будет происходить медленнее и покрывать больше путей выполнения программы; однако если уменьшить значение max-nodes для метода резюме (на практике — примерно в 4 раза), то удается получить сравнимое покрытие путей выполнения программы за существенно меньшее время, что и подтверждается экспериментальными данными.

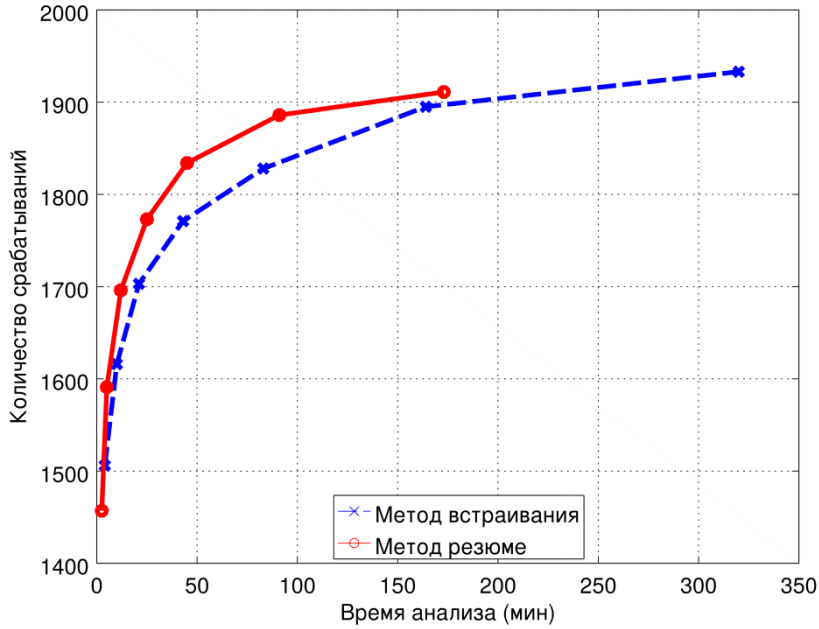

Рис. 3. Зависимость количества найденных дефектов от времени анализа при реализации межпроцедурного анализа методом встраивания и методом резюме
Fig. 3. The dependence of the number of defects found on the analysis run-time duration time in the implementations of interprocedural analysis by the incorporation and summary methods

На рисунке 3 изображены составленные по таблицам 1 и 2 графики зависимости количества дефектов, найденных различными методами, от продолжительности анализа. На графиках видно, метод резюме находит больше дефектов, чем метод встраивания, за то же время, либо позволяет найти аналогичное количество дефектов за существенно меньшее время.

7. Заключение.

В результате проведенного исследования был разработан метод межпроцедурного анализа, основанный на составлении и применении резюме функций, и обладающий существенным превосходством в производительности при анализе крупных программных проектов по сравнению с имеющимися методиками. Предлагаемый метод был реализован на базе статического анализатора Clang Static Analyzer, оперирующего исходным кодом на языках C и C++ и осуществляющим в процессе анализа символьное выполнение кода. Для данного анализатора ранее был реализован межпроцедурный анализ методом встраивания, что позволило провести сравнительный анализ двух подходов. Авторами статьи было продемонстрировано, что разработанный метод является достаточно мощным для анализа исходного кода на высокоуровневых языках программирования, и достаточно расширяемым для реализации различных видов проверок. Был разработан и реализован метод построения информативного отчета о найденных дефектах, учитывающий специфику метода резюме. Проведена оценка производительности и масштабируемости предложенного метода.

Список литературы

[1]. David A. Wheeler. How to Prevent the next Heartbleed. 2015. http://www.dwheeler.com/essays/heartbleed.html
[2]. John Carmack. Static Code Analysis. 2011. http://www.viva64.com/en/a/0087/
[3]. King, James C. Symbolic Execution and Program Testing. Commun. ACM, 19(7), 1976. P. 385–394.
[4]. Godefroid Patrice, Klarlund Nils, Sen Koushik. DART: Directed Automated Random Testing. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI ’05. New York, NY, USA: ACM, 2005. P. 213–223.
[5]. Saswat Anand, Păsăreanu Corina S, Willem Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2007.
[6]. В.П. Иванников, А.А. Белеванцев, А.Е. Бородин и др. Статический анализатор Svace для поиска дефектов в исходном коде программ. Труды Института системного программирования РАН. 2014, 26 (1). С. 231–250.
[7]. Matsumoto Hiroo. Applying Clang Static Analyzer to Linux Kernel. 2012 LinuxCon Japan. Yokohama: 2012. 6.
[8]. Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. Proceedings of the

8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008. P. 209–224.

[9]. Android Open Source Project. http://source.android.com/

[10]. Reps Thomas, Horwitz Susan, Sagiv Mooly. Precise Interprocedural Dataflow Analysis via Graph Reachability. Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL '95. New York, NY, USA: ACM, 1995. P. 49–61.

[11]. Компиляторы. Принципы, технологии и инструментарий. А.В. Ахо, М.С. Лам, Рави Сети, Д.Д. Ульман. Вильямс, 2003. — 1184 с.

[12]. Rojas José Miguel, Păsăreanu Corina S. Compositional Symbolic Execution through Program Specialization. BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2013.

[13]. Godefroid Patrice. Compositional Dynamic Test Generation. SIGPLAN Not. 2007. 42(1). P. 47–54.

[14]. Summary-based inference of quantitative bounds of live heap objects. Víctor Brabermana, Diego Garbervetskya, Samuel Hymc, Sergio Yovinea Science of Computer Programming, 92, 2013. P. 56–84.

[15]. Xu Zhenbo, Zhang Jian, Xu Zhongxing. Melton: a practical and precise memory leak detection tool for C programs. Frontiers of Computer Science in China, 9(1), 2015. P. 34–54.

[16]. Clang Static Analyzer. http://clang-analyzer.llvm.org/

[17]. Strom, Robert E.; Yemini, Shaula. Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering (IEEE), 12, 1986. P. 157–171.

[18]. Xu Zhongxing, Kremenek Ted, Zhang Jian. A Memory Model for Static Analysis of C Programs. Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation. Volume Part I. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010. P. 535–548.

Summary-based method of implementing arbitrary context-sensitive checks for source-based analysis via symbolic execution

A. Dergachev <dergachev.a@samsung.com>
A. Sidorin <a.sidorin@samsung.com>
Samsung R&D Institute Russia,
Dvintsev 12, 127018 Moscow, Russia

Abstract. A specific approach to summary-based interprocedural symbolic execution is described. The approach is suitable for analysis of program source code developed with high-level programming languages and allows executing arbitrarily complex checks during symbolic execution, including throwing reports in the callee function about defects that only become certain within the caller context. The structure of the function summary, procedure of applying the summary in a particular context, composition of symbolic values for particular contexts, effect of summary-based analysis on complexity of implementing specific checker modules, procedure for constructing path-sensitive bug reports, and other aspects of the implementation are discussed in detail. A particular implementation of the approach, based

on Clang Static Analyzer, is described. The implementation is scalable enough to allow analysis of large-scale software projects in reasonable time, finding bugs faster than the existing implementation of the inlining-based interprocedural analysis, without sacrificing correctness and soundness of the analysis. Particular checker modules, which find various defects, such as integer overflows, modifications of constant-qualified memory, multithreading issues, array bound checks, exception safety checks, and file stream errors, were updated to use the summary-based approach, demonstrating flexibility of the technique proposed. The implementation was tested by running full intra-unit inter-procedural analysis of the Android Open Source Project codebase.

Keywords: static analysis, symbolic execution, interprocedural analysis, context-sensitive analysis, summary-based analysis, C, C++, Clang Static Analyzer.

DOI: 10.15514/ISPRAS-2016-28(1)-3

For citation: Dergachev A.V., Sidorin A.V.. Summary-based method of implementing arbitrary context-sensitive checks for source-based analysis via symbolic execution. *Trudy ISP RAN/Proc. ISP RAS*, vol. 28, issue 1, 2016, pp. 41-62 (in Russian). DOI: 10.15514/ISPRAS-2016-28(1)-3

References

[1]. David A. Wheeler. How to Prevent the next Heartbleed. 2015. http://www.dwheeler.com/essays/heartbleed.html
[2]. John Carmack. Static Code Analysis. 2011. http://www.viva64.com/en/a/0087/
[3]. King James C. Symbolic Execution and Program Testing. Commun. ACM, 19(7), 1976. P. 385–394.
[4]. Godefroid Patrice, Klarlund Nils, Sen Koushik. DART: Directed Automated Random Testing. Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation. PLDI '05. New York, NY, USA: ACM, 2005. P. 213–223.
[5]. Saswat Anand, Păsăreanu Corina S, Willem Visser. JPF-SE: A Symbolic Execution Extension to Java Pathfinder. International conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 2007.
[6]. V.P. Ivannikov, A.A. Belevancev, A.E. Borodin et. al. Staticheskij analizator Svace dlja poiska defektov v ishodnom kode program. [Static analyzer Svace for finding defects in a source program code]. Trudy ISP RAN [Proceedings of ISP RAS]. 2014, vol. 26, issue 1, pp. 231–250 (in Russian). DOI: 10.15514/ISPRAS-2014-26(1)-7
[7]. Matsumoto Hiroo. Applying Clang Static Analyzer to Linux Kernel. 2012 LinuxCon Japan. Yokohama: 2012. 6.
[8]. Cadar Cristian, Dunbar Daniel, Engler Dawson. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008. P. 209–224.
[9]. Android Open Source Project. http://source.android.com/
[10]. Reps Thomas, Horwitz Susan, Sagiv Mooly. Precise Interprocedural Dataflow Analysis via Graph Reachability. Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium

on Principles of Programming Languages. POPL '95. New York, NY, USA: ACM, 1995. P. 49–61.

[11]. Compilers: Principles, Techniques, and Tools (2nd Edition). A.V. Aho, M.S. Lam, Ravi Sethi, D.D. Ullman. Pearson Education, Inc, 2006.

[12]. Rojas José Miguel, Păsăreanu Corina S. Compositional Symbolic Execution through Program Specialization. BYTECODE 2013, 8th Workshop on Bytecode Semantics, Verification, Analysis and Transformation, 2013.

[13]. Godefroid Patrice. Compositional Dynamic Test Generation. SIGPLAN Not. 2007. 42(1). P. 47–54.

[14]. Summary-based inference of quantitative bounds of live heap objects. Víctor Brabermana, Diego Garbervetskya, Samuel Hymc, Sergio Yovinea Science of Computer Programming, 92, 2013. P. 56–84.

[15]. Xu Zhenbo, Zhang Jian, Xu Zhongxing. Melton: a practical and precise memory leak detection tool for C programs. Frontiers of Computer Science in China, 9(1), 2015. P. 34–54.

[16]. Clang Static Analyzer. http://clang-analyzer.llvm.org/

[17]. Strom, Robert E.; Yemini, Shaula. Typestate: A programming language concept for enhancing software reliability. IEEE Transactions on Software Engineering (IEEE), 12, 1986. P. 157–171.

[18]. Xu Zhongxing, Kremenek Ted, Zhang Jian. A Memory Model for Static Analysis of C Programs. Proceedings of the 4th International Conference on Leveraging Applications of Formal Methods, Verification, and Validation. Volume Part I. ISoLA'10. Berlin, Heidelberg: Springer-Verlag, 2010. P. 535–548.