Наука и Образование
МГТУ им. Н.Э. Баумана
Сетевое научное издание
ISSN 1994-0408

Ссылка на статью:
// Наука и Образование. МГТУ им. Н.Э. Баумана. Электрон. журн. 2015. № 8. С. 281–300.
DOI: **10.7463/0815.0793227**
Представлена в редакцию: 24.07.2015

УДК 004.4'2

Новая модификация метода анализа кодов программ на основе резюме для тестирования сложных программных комплексов

Сидорин А. В.[1,2,*], **Романова Т. Н.**[1]

[*]alexey.v.sidorin@ya.ru

[1]МГТУ им. Н.Э. Баумана, Москва, Россия
[2]ООО «Московский исследовательский центр Samsung», Москва, Россия

В связи с распространением средств автоматического тестирования возникает необходимость подвергать тщательному автоматизированному тестированию крупные программные и программно-технические комплексы, включая тестирование интерфейсов взаимодействия компонентов системы между собой. Для этого необходимо преодолевать проблемы масштабируемости различных методов анализа программ, связанные с тем, что полный анализ всех путей выполнения программы невозможен. Целью проводимого исследования является построение метода межпроцедурного анализа с производительностью, позволяющей выполнять анализ крупных программных систем (таких, как ОС Android) за приемлемое время (не более 4-х часов). В работе впервые исследована возможность использования метода резюме для решения проблемы масштабируемости метода символьного выполнения. Разработана математическая модель, которая позволит оценить потенциальный прирост производительности при использовании межпроцедурного анализа на основе резюме в сравнении с методом встраивания. Для этого в качестве целевой системы для реализации метода резюме выбран свободно распространяемый статический анализатор Clang Static Analyzer. Приведены предварительные расчеты, показывающие возможность увеличения быстродействия при использовании данного метода и рассмотрены основные источники увеличения производительности.

Ключевые слова: C++; статический анализ; символьное выполнение; межпроцедурный анализ; метод резюме; Clang Static Analyzer

Введение

Для больших и сложных программно-технических комплексов полное покрытие всех путей выполнения программы становится невозможным, поскольку эта задача соотносится с проблемой останова. Ресурсы, выделенные на тестирование сложных программных комплексов, всегда ограничены, что приводит к необходимости рационального их использования. Проблема поиска подходящего компромисса между повышением надежности разрабатываемых программных средств и эффективным использованием ресурсов становится все актуальнее. Для обеспечения надежности программных средств активно ведется разработка

новых эффективных методов и средств автоматического тестирования, позволяющих за реальное время предупредить и выявить как можно большее количество дефектов в программе. В настоящее время все большее распространение получают инструменты, предназначенные для поиска дефектов в программном коде.

Обычно различают статический, динамический и смешанный анализ. Под статическим анализом понимают анализ программы, не требующий ее непосредственного выполнения. Часть инструментов, таких, как Clang Static Analyzer [1], PVS-Studio [2], Cppcheck [3], Lint [4], исследует непосредственно код программы или структуры данные, строящиеся на его основе, — абстрактное синтаксическое дерево или граф потока управления. Другая часть инструментов статического анализа использует для анализа более низкоуровневое представление программы — скомпилированный объектный или промежуточный код (Coverity Prevent [5], Svace [6], FindBugs [7]). В отличие от статического анализа, для динамического анализа программы требуется ее выполнение — на специальных входных данных, в виртуальной машине (Valgrind [8]), с использованием инструментации (AddressSanitizer [9], ThreadSanitizer [10], UndefinedBehaviorSanitizer), с использованием дополнительных библиотек или их подменой. Наконец, смешанный анализ представляет собой комбинацию статического и динамического анализа и используется в таких инструментах как Mayhem [11], KLEE [12], а также других автоматических генераторов контрпримеров.

Перечисленные виды анализа имеют свои достоинства и недостатки, в частности, различные виды анализа наиболее эффективны для поиска различных видов ошибок.

• Динамический анализ наиболее хорошо зарекомендовал себя для поиска ошибок, связанных с многопоточностью и управлением памятью, однако крайне затратен в случае больших проектов. Значительным недостатком динамического анализа является необходимость явного выполнения программы, что влечет за собой необходимость подготовки входных данных (или их автогенерации), и быстрый рост длительности такого анализа с увеличением объема проекта. Это также затрудняет интеграцию инструментов, использующих динамический анализ, в процесс разработки, что снижает шансы быстрого обнаружения ошибки.

• Статический анализ позволяет эффективно производить поиск различных видов дефектов: опечаток, некорректного использования типов, проблем безопасности, неопределенного или недокументированного поведения и многих других видов. Инструменты для выполнения статического анализа могут быть легко интегрированы в процесс разработки. При этом они могут быть использованы как индивидуальные вспомогательные инструменты разработки (например, для подсветки кода, содержащего потенциальную ошибку), так и в качестве инструментов, использующихся группой разработчиков (например, для развертывания и интеграции в систему непрерывной сборки). Сравнительно небольшое время, затрачиваемое на анализ, вкупе с интеграцией в рабочий процесс позволяет быстро находить дефекты в разрабатываемых программах. Недостатком статических анализаторов является возможность выдачи ими некорректных сообщений об ошибках — ложных срабатываний (ошибок первого рода) и возможность пропуска имеющихся дефектов (ошибки второго рода), веро-

ятность которых стараются снизить при разработке анализаторов. Вред от ошибок второго рода очевиден, но и ошибки второго рода играют не меньшую роль при оценке качества анализатора, поскольку их большое количество отвлекает разработчика на длительное время для просмотра ложных срабатываний, поэтому при большом количестве ложных срабатываний инструмент может стать практически непригодным для использования. Однако при небольшом количестве ложных срабатываний польза от применения анализатора в виде снижения времени, затрачиваемого на обнаружение ошибки, быстро перевешивает недостаток в виде времени, затрачиваемого на просмотр ложных срабатываний.

Первоначально распространение у разработчиков получили инструменты, использующие методы на основе анализа синтаксического дерева программы и ее графа потока управления. Преимуществами данных методов анализа программного кода являются:

- высокая скорость работы;
- незначительное потребление памяти;
- возможность его реализации в компиляторе для выполнения дополнительных проверок и предупреждения программиста о потенциально некорректном поведении компилируемого кода. Это становится возможным благодаря малому потреблению системных ресурсов, позволяющему лишь незначительно снижать производительность компилятора;
- возможность интеграции в среды разработки для осуществления анализа «на лету», непосредственно в процессе набора кода программистом, или в качестве дополнительного инструмента для быстрого обнаружения дефекта.

Аналогичные методы применяются в компиляторах для предупреждения программиста о потенциально некорректном поведении программы, поскольку и синтаксическое дерево, и граф потока управления являются основными структурами данных, с которыми работает компилятор. Однако проверка, включаемая в состав компилятора, должна исключать возможность ложных срабатываний, т. е. являться консервативной. Инструменты же статического анализа могут включать также и неконсервативные проверки, с возможностью выдачи ложных срабатываний.

Данные методы могут обнаруживать лишь очень узкие классы дефектов в программном коде: простые ошибки, затрагивающие лишь несколько операторов, расположенных в пределах одной функции. Это может быть простейший поиск использования неинициализированных переменных, ошибок при преобразовании типов, потенциально лишние операции, а также другие дефекты, для поиска которых не требуется анализировать циклы и условные переходы. При наличии циклов и переходов в анализируемой функции эффективность видов анализа, нечувствительных к путям выполнения, резко падает, поскольку данные методы позволяют корректно определить достижимость одних операторов из других операторов при выполнении программы лишь в тривиальных случаях.

Значительно более ресурсоемким, но и более подробным является анализ на основе обхода путей выполнения программы. Основы этих методов были заложены еще в 70-х годах. Метод символьного выполнения был предложен Джеймсом Кингом в 1976 году [13].

В основе метода лежит идея разбиения входных данных на классы эквивалентности в зависимости от встречаемых по пути выполнения условий. Метод абстрактной интерпретации, предложенный в 1977 году супругами Кузо [14], предполагает использовать абстрагирование данных и их анализ на основе алгебры решеток. Однако данные походы стали получать распространение только в последнее время. Это связано с увеличившейся мощностью компьютеров: время анализа растет пропорционально количеству путей выполнения, что означает экспоненциальный рост времени анализа с увеличением размера программы. (Вообще говоря, абсолютно полный и точный анализ программы невозможен в связи с проблемой останова, независимо от применяемого подхода.) В отличие от базового анализа графа потока управления, анализ путей выполнения способен учитывать условия выполнения тех или иных ветвей программы, следствием чего являются преимущества данного вида анализа — его более высокая точность и способность покрыть намного больший класс дефектов. Такие методы, как абстрактная интерпретация и символьное выполнение, нашли применения в известных инструментах для поиска дефектов, например, Coverity SAVE, Clang Static Analyzer и многих других.

Одними из наиболее актуальных целевых языков для статического анализа традиционно являются языки C и C++. Причин для этого несколько. Во-первых, это связано с большим количеством видов потенциальных ошибок, которые может допустить программист, ведущий разработку с использованием этих языков. Наиболее специфичными среди таких ошибок являются ошибки, связанные с неправильной работой с указателями — переполнение буфера, обращение к неинициализированной памяти или к памяти по некорректному адресу. Во-вторых, стандарты языков трактуют достаточно большое количество ситуаций как не имеющих определенного поведения (например, порядок вычисления аргументов функций может быть произвольным), что, с одной стороны, позволяет компилятору проводить более глубокие оптимизации и получить наибольшую скорость выполнения результирующего кода, но, с другой стороны, требует от программиста повышенного внимания в процессе написания кода программы для учета этих особенностей. В-третьих, эти языки являются одними из самых распространенных и известных, с их использованием было разработано и продолжает создаваться большое количество как системного, так и прикладного программного обеспечения. Кроме того, язык C является практически единственным выбором при разработке низкоуровневых компонентов, например, компонентов операционных систем и драйверов, что также предъявляет повышенные требования к качеству программного кода.

1. Модифицированный метод резюме для символьного выполнения

В данной работе исследуется метод символьного выполнения [13], применяемый для анализа путей выполнения программ. Этот метод подразумевает абстрактное движение по путям программы, имитирующее ее выполнение в зависимости от входных данных, сопровождающееся изменением состояния программы в различных точках. Суть метода

символьного выполнения заключается в разбиении множества входных данных на классы эквивалентности, что позволяет оперировать при анализе не отдельными входными значениями (число которых может быть очень большим и экспоненциально растет в зависимости от количества входных аргументов) и их перебором, а целыми классами эквивалентности, число которых может оказаться и не конечным, но не превышает общее количество комбинаций отдельных входных значений. Однако, как правило, количество классов эквивалентности комбинаций входных данных оказывается значительно ниже числа всех возможных комбинаций входных данных, что резко увеличивает возможности анализатора по обработке путей выполнения.

Основной алгоритм символического выполнения [13] заключается в следующем.

1. При старте анализа функции входные значения ее аргументов и внешних по отношении к ней переменных неизвестны, т. е. они потенциально могут принимать любые значения. Начальное состояние является корневым узлом специального графа — дерева выполнения программы.

2. Каждой неизвестной величине назначается абстрактное значение, называемое символьным значением.

3. Обработка операторов языка изменяет значения переменных программы, т. е. их символьные значения. Считается, что над символьными значениями уже определен необходимый набор операций для вычисления новых символьных значений на основе уже имеющихся. Выполнение каждого оператора добавляет узел к дереву выполнения программы с входящим ребром от предыдущего оператора.

4. При обработке условных операторов в дерево выполнения программы добавляется не один, а два узла: в первом узле условие выполняется, во втором — нет. Совокупность условий, при которых достижим данный узел графа выполнения программы, определяет класс эквивалентности входных данных программы или функции. Таким образом, каждый лист дерева выполнения программы соответствует классу эквивалентности входных данных, которая приводит программу в конечное состояние, обозначаемое данным листом.

5. Обычно каждое из условий, определяющих класс эквивалентности входных данных, можно представить в виде уравнения или неравенства. В результате наложения на путь выполнения программы нескольких условий в соответствие каждому классу эквивалентности входных данных ставится система уравнений или неравенств. Решениями этих систем уравнений являются множества реальных значений входных величин, при которых будут выполнены ветви выполнения программы. Если система является несовместной, т. е. не имеет ни одного решения, то путь выполнения, соответствующий этой системе, недостижим.

6. Анализатор производит обход всех путей получившегося дерева выполнения с целью поиска ситуаций, которые могли бы трактоваться как некорректное поведение. В случае обнаружения такой ситуации анализатор сообщает о дефекте и при этом указывает набор условий, при выполнении которых программа проявит некорректное поведение.

Данный алгоритм можно рассмотреть на следующем примере. Пусть имеется следующая программа чтения из файла на языке C:

```
1  int test(int a, int b) {
2    FILE *f = fopen("file.txt");
3    int result;
4    fscanf(f, "%d", &result);
5    fclose(f);
6    if (a == 0 && b > 2) {
7      fclose(f);
8      return 0;
9    }
10   return result;
11 }
```

В результате выполнения этой программы будут достижимы три конечных состояния с уравнениями $a \neq 0, b \in [\text{INT\_MIN}, \text{INT\_MAX}]$; $a = 0, b \leqslant 2$; $a = 0, b > 2$. Соответствующий граф представлен на рис. 1.

В результате получены три пути с соответствующими ограничениями. Теперь анализатор просматривает каждый из этих путей и обнаруживает дефект: на одном из путей файл `f` закрывается дважды, что приводит к неопределенному поведению. Этому пути соответствует система $a = 0, b > 2$. Таким образом дефекту сопоставляется множество условий, при котором он проявляется при выполнении программы.

Основным преимуществом метода символьного выполнения является простота и очевидность концепции, на которой он основан: метод использует идею ««симуляции»» выполнения программы, так, как это делает программист. Метод символьного выполнения получил распространение не только в инструментах статического, но и смешанного анализа: так, хорошо зарекомендовали себя инструменты, использующие подход concolic testing [16] (символьно-конкретное — concrete+symbolic). Concolic testing — это метод поиска дефектов, осуществляющий генерацию тестовых данные, при использовании которых программа проявляет некорректное поведение, на основе символьного выполнения.

Наряду с преимуществами, метод имеет ряд недостатков. Так, существует проблема экспоненциального роста количества проходимых путей (path explosion), приводящая к проблемам с масштабируемостью метода. Есть также проблемы при моделировании циклов, поскольку зачастую количество итераций цикла точно неизвестно — оно также является символьной величиной. Тем не менее, метод символьного выполнения активно применяется, в том числе, целым набором широко используемых инструментов анализа программ. Таким образом, разработка подходов для улучшения данного метода является актуальной и практически важной задачей.

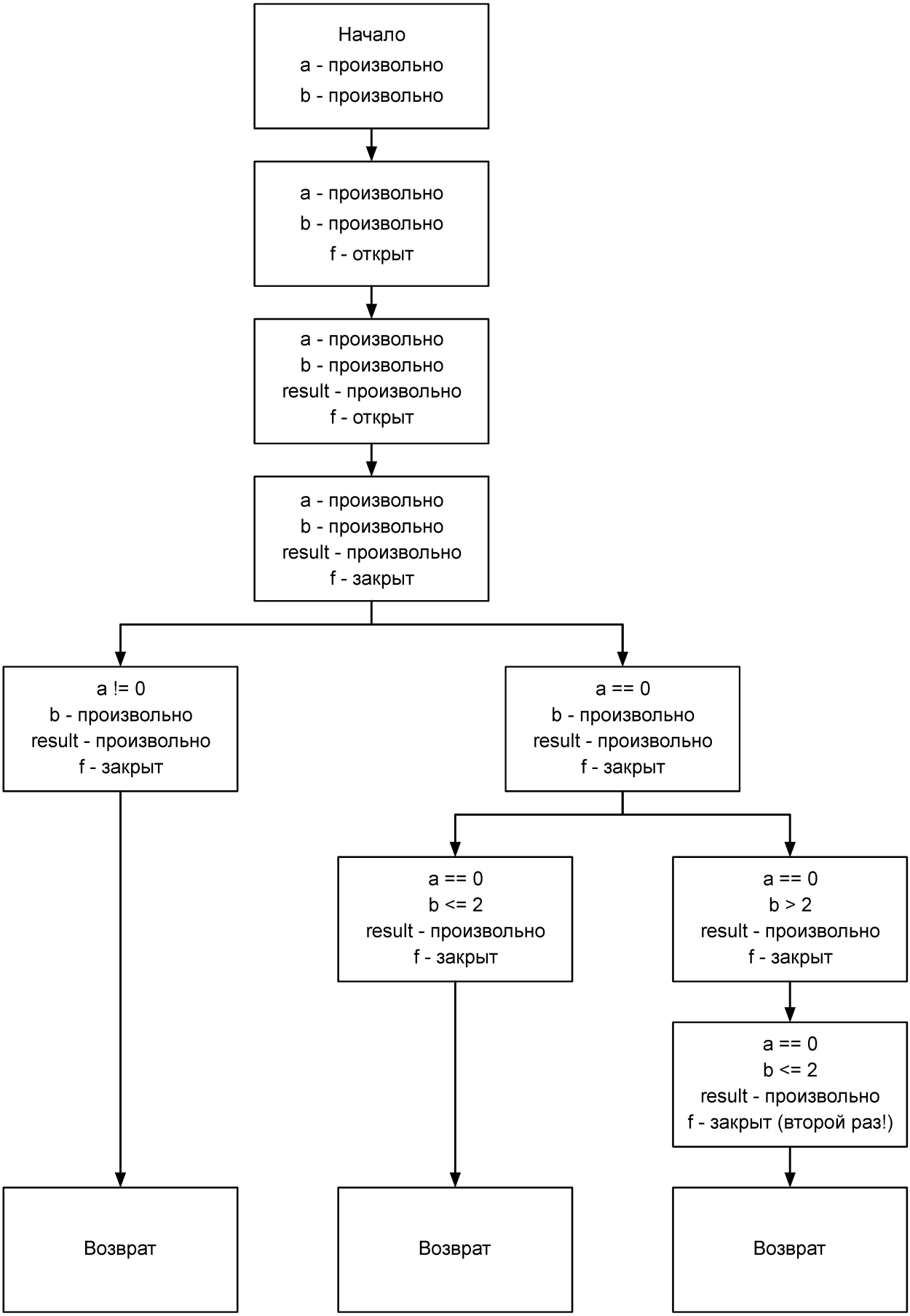

Рис. 1. Граф выполнения программы чтения из файла

Основные проблемы масштабируемости метода связаны с двумя факторами.

1. При моделировании циклов время анализа линейно зависит от количества итераций, проходимых программой при выполнении цикла. Даже если число итераций известно, но велико, анализ программы выполняется длительное время, резко возрастающее при наличии в программе вложенных циклов. При использовании смешанного анализа обычно эту проблему анализом выполнения программы для установления реального количества итераций. При статическом анализе наиболее распространенным решением является ограничение количества итераций циклов каким-либо максимальным константным значением. Этот подход позволяет ограничить время анализа, однако не решает проблему роста времени анализа при наличии вложенных циклов. Кроме того, это ограничение приводит к потере точности моделирования, что, в свою очередь, приводит к ложным срабатываниям или отсутствию срабатываний в тех случаях, когда они ожидаются.

2. Время анализа быстро растет при использовании межпроцедурного анализа.

Отличие межпроцедурного анализа (МПА) от внутрипроцедурного (ВПА) заключается в том, что анализатор позволяет использовать доступные определения пользовательских функций для моделирования эффектов их вызовов. МПА используется для решения двух основных проблем, связанных с использованием внутрипроцедурного анализа. Во-первых, межпроцедурный анализ позволяет определить эффект, оказываемый на состояние программы в результате вызова из анализируемой функции другой функции. В отсутствие межпроцедурного анализа вызов функции можно моделировать либо самостоятельно (с помощью спецификаций эффектов), либо приближенно. Первый подход, как правило, используется для функций, эффекты которых специфицированы. Таковы, например, функции различных публичных интерфейсов взаимодействия. Наиболее распространено такое моделирование для Posix API, встречаются также реализации, моделирующие вызовы Windows API. Кроме того, хорошими кандидатами на специфицирование являются функции, принадлежащие стандартной библиотеке языка, поскольку она, как правило, стандартизирована, реже — функции других распространенных библиотек (например, STL и др.).

Во-вторых, в случае, если полная спецификация функции недоступна, при внутрипроцедурном анализе может использоваться приближенное моделирование. В этом случае считается, что вызов функции может произвести любые действия с данными, которые доступны внутри функции (для языков, имеющих операции арифметики с указателями, например, C/C++, в общем случае можно считать, что программа может модифицировать любые данные), и вернуть произвольное значение. Для уточнения эффектов может использоваться анализ атрибутов доступной декларации функции, например, информация о модификаторах типов аргументов функции, атрибуты аргументов и самой функции. Так, например, функция, объявленная с GNU-атрибутом `__attribute__((pure))`, не имеет прав на изменение глобальной памяти и аргументов, а функция с атрибутом `__attribute__((noreturn))` никогда не вернет управление в вызывающую функцию. Могут также использоваться различные

эвристики. Вместе с тем, приближенное моделирование может решить проблему анализа лишь частично. Из-за невозможности оценки влияния вызова на состояние программы анализатор может сделать некорректные выводы о текущем состоянии выполнения программы, что может привести как к ложным срабатываниям анализатора (ошибка первого рода), так и к отсутствию срабатывания в условиях, когда анализатор должен выдавать диагностическое предупреждение (ошибка второго рода).

Существует два основных подхода к реализации контекстно-чувствительного межпроцедурного анализа. Первый подход — анализ на основе встраивания функции. Этот подход характеризуется высокой точностью анализа, поскольку он, фактически, полностью имитирует процесс реального выполнения вызова функции. Однако данный метод имеет недостаток, который делает его малоприменимым для крупных проектов — низкую масштабируемость, поскольку моделирование вызовов одной и той же функции может выполняться много раз подряд. Следствием многократного встраивания является при этом как увеличение времени анализа, так и увеличение объема потребляемой памяти, поскольку при каждом моделировании вызова функции создается новый фрагмент графа анализа, представляющий путь выполнения вложенного вызова функции в контексте вызова. Таким образом, проведенное исследование показало, что ввиду низкой масштабируемости метода встраивания функции его невозможно применять для анализа сложных программно-технических комплексов.

Второй подход реализации межпроцедурного анализа — анализ на основе резюме. Под резюме в случае межпроцедурного анализа понимают некоторую информацию о функции, использование которой позволяет рассчитать эффект вызова этой функции. Фактически, резюме является передаточной функцией, применяемой для аппроксимации вместо вычисления точного эффекта вызова путем встраивания функции. Основная идея подхода, связанного с применением резюме, заключается в применении некоторого конечного набора эффектов вызываемой функции вместо прохождения ее путей выполнения (как правило, многократного, поскольку одна и та же функция может вызываться в различных участках программы, а также в циклах). Идеальная передаточная функция должна порождать новые пути в количестве, равном таковому при использовании встраивания. Отсюда следует, что использование межпроцедурного анализа на основе резюме не позволяет полностью уйти от экспоненциального роста времени анализа, если не применять дополнительных упрощений передаточной функции и эвристик анализа. Тем не менее, использование резюме может значительно уменьшить время, которое затрачивается на анализ вызова функции, поскольку исключает анализ значительного количества узлов графа путей выполнения, связанных с выполнением встраиваемой функции.

На основе проведенного исследования существующих подходов к тестированию крупных программных комплексов в данной работе предложена новая модификация метода межпроцедурного анализа с использованием резюме для метода символьного выполнения и разработан алгоритм его реализации.

Математическая модель. Разработанный алгоритм метода резюме для символьного выполнения. Рассмотрим основные источники затрат при встраивании и при использовании резюме. При встраивании функций вызываемая функция анализируется каждый раз при ее вызове. При этом в случае контекстно-чувствительного анализа функция анализируется не полностью: анализируются лишь те пути выполнения, которые являются достижимыми при контексте на момент вызова. Суммарное время, затраченное на анализ функции при допустимой степени вложенности, равной 1, можно вычислить по формуле

$$T_{\text{встраивания}} = \sum_{i=0}^{n} t_i, \tag{1}$$

где i — номер вызова; t_i — время, затраченное на анализ i-го вызова (с учетом контекста). С учетом того, что сама вызываемая функция также анализируется отдельно, формула приобретает вид:

$$T_{\text{встраивания полное}} = T_{\text{анализа}} + \sum_{i=0}^{n} t_i. \tag{2}$$

При использовании подхода, основанного на резюме, временные затраты вычисляются следующим образом. Вызываемая функция анализируется один раз, но полностью (независимо от вида анализа). Однократный характер также носят затраты, связанные с составлением резюме функции. Применение резюме выполняется в каждой точке вызова функции. Таким образом, общие затраты рассчитываются по формуле

$$T_{\text{резюме}} = T_{\text{анализа}} + T_{\text{сбора}} + \sum_{i=0}^{n} t_{i,\text{применения}}\ , \tag{3}$$

что, с учетом примерного равенства времен применения резюме (поскольку резюме имеет не меняющийся размер), приближенно равно

$$T_{\text{резюме}} = T_{\text{анализа}} + T_{\text{сбора}} + nt_{i,\text{применения}}\ . \tag{4}$$

Таким образом, выигрыш от применения резюме будет получен при выполнении следующего соотношения:

$$T_{\text{сбора}} + nt_{i\text{применения}} < \sum_{i=0}^{n} t_i, \tag{5}$$

откуда следует, что для получения ускорения необходимо выполнение соотношения

$$t_{\text{ср. применения}} < t_{\text{ср.}}. \tag{6}$$

Пусть $s_1, \ldots, s_n$ — некоторая последовательность операторов программы. Каждый оператор имеет набор эффектов, который он оказывает на состояние выполнения программы. Таким образом, каждый оператор программы представляет собой передаточную функцию: $p_i = s_i(p_{i-1})$, где p_{i-1} — состояние программы непосредственно перед выполнением оператора, p_i — состояние программы непосредственно после выполнением оператора s_i (и перед выполнением оператора s_{i+1}). Тогда в результате выполнения всего

блока операторов программа из начального состояния p_0 перейдет в состояние p_n: $p_n = s_n(s_{n-1}(\ldots s_i(\ldots(s_1(p_0))\ldots))$. Тогда суммарный эффект последовательности операторов можно представить в виде композиции их передаточных функций:

$$s = s_1 \circ s_2 \circ \ldots \circ s_n.$$

Данная формула справедлива для последовательности операторов без переходов, то есть для базовых блоков программы. Кроме того, формула справедлива для последовательности операторов, содержащей безусловный переход, поскольку такая последовательность также представляет собой путь выполнения без ветвлений. Однако при наличии условных переходов в блоке последовательности операторов, оказывающих эффект на выполнение программы, могут различаться. Это означает, что суммарный эффект выполнения блока зависит от пути выполнения внутри блока, а следовательно, и от значения выражения в условии.

Пусть c_j — условие, принадлежащее анализируемому блоку, $0 \leqslant j \leqslant m$, в ветках `if` и `else` которого находятся непрерывные последовательности операторов $s_0, \ldots, s_k$ и $s_{k+1}, \ldots, s_n$ соответственно, возможно, пустые. Тогда будет справедливо следующее соотношение:

$$s_c = \begin{cases} s_1 \circ \ldots \circ s_k, & c_j \equiv \texttt{true}; \\ s_{k+1} \circ \ldots \circ s_n, & c_j \equiv \texttt{false}. \end{cases}$$

С использованием данных правил можно строить композиции эффектов произвольных последовательностей операторов.

Поскольку тело функции также является последовательностью операторов языка, эффект от вызова функции можно рассчитать по тем же правилам. На основе зависимости полученного эффекта от условий на пути выполнения внутри функции и строится резюме. При этом в резюме сохраняются не все эффекты, производимые операторами, содержащимися в теле функции, а лишь те из них, которые сохраняются после выхода из нее и могут повлиять на дальнейшее выполнение программы после выхода из функции. Таким образом, сократить время анализа при использовании резюме в сравнении со встраиванием можно получить за счет отсутствия необходимости затрачивать время на анализ эффектов, действия которых локальны или не учитываются при дальнейшем анализе. Так, связывание символьного значения с выражением имеет только локальный эффект, поскольку все выражения становятся неактивными при выходе из контекста анализа функции. Аналогично, локальный эффект имеют записи в локально видимую память и т. д. Кроме того, модель анализатора заведомо допускает упрощения, поскольку анализатор не может досконально смоделировать поведение программы. Это означает, что ряд эффектов операторов не будет учтен, т. е. на моделирование некоторых эффектов операторов будет затрачено время, однако результат этого моделирования не будет отражен в изменении состояния. Учет этих упрощений и ограничений анализатора позволяет устранить непроизводительные затраты времени, поскольку при применении резюме непроизводительные вычисления не выполняются повторно. Возможно, однако, что некоторые эффекты самого применения резюме

не могут быть учтены моделью анализатора и также будут отнесены к непроизводительным вычислениям. Но, поскольку набор эффектов, получаемых в результате применения резюме, включается строго или совпадает с набором эффектов, моделируемых при анализе методом встраивания, время, затрачиваемое на применение резюме, по-прежнему не будет превышать время, требуемое на анализ вызова функции методом встраивания.

Алгоритм метода резюме для символьного выполнения. В результате проведенного анализа в данной работе построен алгоритм метода межпроцедурного анализа с помощью резюме для метода символьного выполнения.

1. Провести анализ вызываемой функции, получив в результате ее граф выполнения.

2. Для каждого конечного узла графа выполнения функции осуществить сбор эффектов, оказываемых на выполнение программы при выполнении данной ветви выполнения. Полученным результатом является набор ветвей резюме.

3. В каждой точке вызова проанализированной функции создать новые узлы графа выполнения (узлы применения резюме) со следующими характеристиками:

• Дуги графа выполнения ведут из узла, соответствующего вызову функции (узел вызова) в каждый из узлов применения резюме.

• Каждая точка применения резюме соответствует листу графа выполнения вызываемой функции и, соответственно, своей ветви резюме.

• Состояние программы в каждой точке применения резюме есть композиция состояния программы в узле вызова и функции, описываемой соответствующей ветвью резюме.

Таким образом, множество узлов графа выполнения вызываемой функции отображается в множество узлов применения резюме. Поскольку множество всех узлов графа выполнения, как правило, многократно превосходит по количеству элементов множество листов графа, данный метод имеет значительно большую потенциальную масштабируемость.

2. Выбор инструментария

Одной из наиболее известных открытых систем для анализа кода на языках C и C++ является программный комплекс LLVM [17] и входящий в ее состав компилятор Clang [18]. LLVM позволяет анализировать код любых языков при условии, что он транслируется в промежуточное представление LLVM (LLVM IR), являясь, таким образом, как анализатором промежуточного представления, так и оптимизирующей виртуальной машиной для его исполнения и генератором объектного кода. Clang является свободным (имеет лицензию MIT) компилятором (фронтэндом, т. е. транслятором в промежуточное представление) для языков C, C++, Objective-C и Objective-C++. В отличие от известных компиляторов, Clang позиционируется не только как кодогенератор, но и как полноценный фреймворк для анализа программного кода с большим количеством возможностей. Clang может производить анализ непосредственно исходных кодов программ, а также предоставляет возможность написания собственных инструментов для анализа кода с использованием его программных

интерфейсов (Clang API). Кроме того, Clang является основным компилятором в некоторых дистрибутивах операционных систем и программно-аппаратных комплексов (в частности, FreeBSD, PlayStation) ввиду простоты его модификации и добавления новых возможностей, и известен как компилятор/фреймворк с наиболее быстрым внедрением новых языковых стандартов.

В состав Clang в качестве модуля входит Clang Static Analyzer [19] — статический анализатор, поддерживающий анализ исходного кода на языках C, C++, Objective C и Objective C++. Clang Static Analyzer (CSA) поддерживает следующие виды анализа.

1. Анализ кода на основе абстрактного синтаксического дерева (AST-based) — поиск фрагментов синтаксического дерева, удовлетворяющих заданным условиям. При данном подходе выполняется поиск фрагмента кода (на основе представляющего его фрагмента синтаксического дерева), который удовлетворяет критерию дефектности. Применительно к Clang, данный анализ может быть реализован как непосредственно обходом синтаксического дерева (с помощью Visitor-интерфейсов: TypeVisitor, StmtVisitor, DeclVisitor), так и с помощью специальных поисковых предикатов в функциональном стиле (ASTMatcher). Данный вид проверок выполняется очень быстро, однако с помощью исключительно анализа синтаксического дерева можно найти лишь ограниченный набор дефектов.

2. Анализ графа потока выполнения программы. Данный анализ выполняется с помощью обхода узлов графа потока выполнения с использованием различных алгоритмов и поиска таким образом фрагментов графа, удовлетворявшим критерию наличия дефекта в программе. Данный анализ является чувствительным к потоку управления (flow-sensitive), но не является ни чувствительным к пути (path-sensitive), ни чувствительным к контексту анализа (context-sensitive). Анализ графа потока выполнения позволяет выполнять более сложные проверки, однако, поскольку он не учитывает реальную достижимость путей выполнения, для реализации проверки и уменьшения количества ложных срабатываний может потребоваться реализация дополнительных эвристик.

3. Поиск дефектов на основе анализа путей выполнения с использованием символьного выполнения. Время выполнения анализа путей примерно в 8-10 раз больше, чем время выполнения других видов анализа, однако его возможность в сравнении с другими видами анализа гораздо выше. Анализ путей выполнения позволяет определить конкретный путь выполнения программы и соответствующий ему набор условий, при выполнении которых программа проявит поведение, определяемое как ошибочное. Благодаря возможность полностью отобразить путь выполнения программист имеет возможность провести детальный анализ условий, приводящих к ошибке, даже на достаточно длительной трассе выполнения. Учет условий позволяет также уменьшить количество потенциальных ложных срабатываний.

Все виды анализов также позволяют использовать встроенные анализы Clang, включая ряд проверок на основе анализа потока данных. Так, для получения дополнительной информации о программе и улучшения реализации проверки можно воспользоваться анализом

активности переменных, базовыми анализами потокобезопасности и достижимости, разбором форматных строк. Кроме того, различные виды анализов можно комбинировать, например, использовать обход синтаксического дерева при реализации проверки на основе анализа путей для получения более полной информации о выполняемом операторе.

Наибольший интерес представляет анализ на основе путей выполнения, поскольку он реализован на основе метода символьного выполнения. Кроме того, данный анализ реализован в Clang как межпроцедурный. Межпроцедурный анализ в CSA также является контекстно-чувствительным, то есть он выполняет моделирование эффектов операторов вызываемой функции с учетом состояния программы на момент вызова функции. Это делает анализ более точным, поскольку позволяет не включать в рассмотрение заведомо недостижимые пути выполнения и уменьшать таким образом количество ложных срабатываний. Вместе с тем, в Clang Static Analyzer реализован подход на основе встраивания, недостатки которого затрудняют его эффективное использование для анализа крупных программных проектов.

В настоящее время единицей анализа для Clang Static Analyzer является транслируемый модуль (translation unit). Clang является компилятором, принцип использования которого аналогичен GCC, и совместим с GCC (имеет место практически полная взаимозаменяемость, за исключением отдельных опций компилятора), и его стандартным использованием в качестве компилятора является компиляция файлов по одному. Хотя Clang содержит в себе некоторые механизмы работы с несколькими модулями трансляции, их использование ограничено инструментами на основе Clang API. Поскольку практика программирования на C/C++, как правило, подразумевает использование многофайловых проектов с индивидуальной компиляцией каждого модуля трансляции (объединение различных модулей происходит на этапе компоновки результирующего объектного файла), определения функций, расположенные в других модулях трансляции, остается недоступным для анализатора. Это, с одной стороны, несколько уменьшает точность анализа, но, с другой стороны, уменьшает его время. Но этот баланс соблюдается далеко не всегда. Так, в случае анализа входных единиц трансляции с большим количеством функций, определенных внутри них (такими являются, например, библиотека sqlite3, некоторые части Android Runtime и отдельные файлы из пакета Webkit), анализ отдельных файлов может длиться более 30 минут, что зачастую неприемлемо.

В связи с этим, представляет интерес сравнение методов встраивания и метода резюме с использованием данного программного комплекса.

Заключение

В данной работе проведено аналитическое исследование современных подходов к тестированию сложных программных комплексов. Более детально рассмотрен метод символьного выполнения для поиска дефектов в коде программ. Описаны его преимущества и недостатки. Данный метод является активно используемым, как в статическом, так и в смешанном ана-

лизе программ. Поэтому задача его улучшения по-прежнему является актуальной. Метод символьного выполнения имеет известные проблемы с производительностью при анализе крупных программ, причиной которых является, в частности, экспоненциальный рост количества путей с увеличением размера программы, особенно заметно проявляющиеся в случае использования межпроцедурного анализа.

Для уменьшения времени анализа и решения проблем с производительностью при анализе крупномасштабных проектов и программных комплексов предложена новая модификация метода анализа кодов программ на основе резюме. Разработанный подход позволяет осуществлять контекстно-чувствительный межпроцедурный межмодульный анализ крупных программных комплексов, разрабатываемых с использованием языков C/C++, уровня ОС Android, за приемлемое время (не более 4-х часов на полный анализ программной системы).

Разработана и описана математическая модель, используемая для оценки потенциального прироста производительности при использовании межпроцедурного анализа на основе резюме в сравнении с методом встраивания.

Разработан алгоритм, реализующий предложенный метод, и на его основе с использованием статического анализатора Clang Static Analyzer разработано ПО, которое может применяться для анализа дефектов в кодах программ сложных программно-технических комплексах. Разработанное ПО может применяться для анализа крупных программных комплексов. В настоящее время разработанное ПО проходит внедрение в центрах разработки ПО компании Samsung и используется для анализа программных продуктов различного назначения, разработанные с использованием языков C и C++.

Список литературы

1. Matsumoto Ню. Applying Clang Static Analyzer to Linux Kernel // 2012 LinuxCon Japan, 6–8 июня 2012 г.: тез. докл. Йокогама, 2012.
2. Описание PVS-Studio. Режим доступа: http://www.viva64.com/ru/pvs-studio (дата обращения 13.05.2015).
3. Rothstein M. Cppcheck design // Secure Programming: cource for the Spring 2012. Режим доступа: http://www.cs.kent.edu/~rothstei/spring_12/secprognotes/cppcheck-design.pdf (дата обращения 14.05.2015).
4. Johnson S.C.. Lint, a C Program Checker // Comp. Sci. Tech. Rep. Bell Laboratories, 1978. Режим доступа: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841 (дата обращения 13.05.2015).
5. Almossawi A., Lim K., Sinha T. Analysis tool evaluation: Coverity Prevent. Final Report. Pittsburgh, PA: Carnegie Mellon University, 2006. 19 p.

6. Иванников В.П., Белеванцев А.А., Бородин А.Е., Игнатьев В.Н., Журихин Д.М., Аветисян А.И., Леонов М.И. Статический анализатор Svace для поиска дефектов в исходном коде программ // Труды Института системного программирования РАН. 2014. Т. 26. № 1. С. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7

7. Hovemeyer D., Pugh W. Finding Bugs is Easy // ACM Sigplan Notices. 2004. Vol. 39, no. 12. P. 92–106. DOI: 10.1145/1052883.1052895

8. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation // ACM Sigplan Notices. 2007. Vol. 42, no. 6. P. 89-100. DOI: 10.1145/1273442.1250746

9. Serebryany K., Bruening D., Potapenko A., Vyukov D.. AddressSanitizer: A Fast Address Sanity Checker // Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12), Boston, MA, USA, June 13–15, 2012. USENIX Association, 2012. P. 309–318. Режим доступа: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf (дата обращения 13.05.2015).

10. Serebryany K., Iskhodzhanov T. ThreadSanitizer — data race detection in practice // Workshop on Binary Instrumentation and Applications (WBIA'09), New York, NY, USA, December 12, 2009. ACM New York, NY, USA, 2009. P. 62–71. DOI: 10.1145/1791194.1791203

11. Cha S.K., Avgerinos T., Rebert A., Brumley D. Unleashing Mayhem on Binary Code // Proc. of the 33rd IEEE Symposium on Security and Privacy, San Francisco, CA, USA, May 20–23, 2012. IEEE, 2012. P. 380–394. DOI: 10.1109/SP.2012.31

12. Cadar C., Dunbar D., Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs // Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI'08), San Diego, CA, USA, December 8–10, 2008. USENIX Association, 2008. P. 209–224. Режим доступа: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf (дата обращения 13.05.2015).

13. King J.C. Symbolic execution and program testing // Communications of the ACM. 1976. Vol. 19, no. 7. P. 385–394. DOI: 10.1145/360248.360252

14. Cousot P., Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints // Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77), Los Angeles, CA, USA, 1977. ACM New York, NY, USA, 1977. P. 238–252. DOI: 10.1145/512950.512973

15. Reps Th., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability // Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, USA, 1995. ACM New York, NY, USA, 1995. P. 49–61. DOI: 10.1145/199448.199462

16. Sen K., Marinov D., Agha G. CUTE: a concolic unit testing engine for C // ACM SIGSOFT Software Engineering Notes. 2005. Vol. 30, iss. 5. P. 263–272. DOI: 10.1145/1095430.1081750
17. The LLVM compiler infrastructure. Режим доступа: http://llvm.org (дата обращения 14.05.2015).
18. clang: a C language family frontend for LLVM. Режим доступа: http://clang.llvm.org (дата обращения 15.05.2015).
19. Clang Static Analyzer. Режим доступа: http://clang-analyzer.llvm.org (дата обращения 15.05.2015).

Science & Education of the Bauman MSTU
Electronic journal
ISSN 1994-0408

Science and Education of the Bauman MSTU, 2015, no. 8, pp. 281–300.
DOI: **10.7463/0815.0793227**
Received: 24.07.2015

A new modification of summary-based analysis method for large software system testing

Sidorin A. V.[1,2,*], **Romanova T. N.**[1] [*]alexey.v.sidorin@ya.ru

[1]Bauman Moscow State Technical University, Russia
[2]Moscow Samsung Research Center, Russia

Keywords: C++, static code analysis, symbolic execution, interprocedural analysis, summary-based method, Clang Static Analyzer

The automated testing tools becoming a frequent practice require thorough computer-aided testing of large software systems, including system inter-component interfaces. To achieve a good coverage, one should overcome scalability problems of different methods of analysis. These problems arise from impossibility to analyze all the execution paths. The objective of this research is to build a method for inter-procedural analysis, which efficiency enables us to analyse large software systems (such as Android OS codebase as a whole) for a reasonable time (no more than 4 hours). This article reviews existing methods of software analysis to detect their potential defects. It focuses on the symbolic execution method since it is widely used both in static analysis of source code and in hybrid analysis of object files and intermediate representation (concolic testing). The method of symbolic execution involves separation of a set of input data values into equivalence classes while choosing an execution path. The paper also considers advantages of this method and its shortcomings. One of the main scalability problems is related to inter-procedural analysis. Analysis time grows rapidly if an inlining method is used for inter-procedural analysis. So this work proposes a summary-based analysis method to solve scalability problems. Clang Static Analyzer, an open source static analyzer (a part of the LLVM project), has been chosen as a target system. It allows us to compare performance of inlining and summary-based inter-procedural analysis. A mathematical model for preliminary estimations is described in order to identify possible factors of performance improvement.

References

1. Matsumoto Hю. Applying Clang Static Analyzer to Linux Kernel. *2012 LinuxCon Japan, June 6–8, 2012*, Yokohama, 2012.

2. *Opisanie PVS-Studio* [PVS-Studio description]. Available at: http://www.viva64.com/ru/pvs-studio, accessed 13.05.2015.
3. Rothstein M. Cppcheck design. In: *Secure Programming: cource for the Spring 2012*. Available at: http://www.cs.kent.edu/~rothstei/spring_12/secprognotes/cppcheck-design.pdf, accessed 14.05.2015.
4. Johnson S.C.. Lint, a C Program Checker. *Comp. Sci. Tech. Rep.*, Bell Laboratories, 1978. Available at: http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.56.1841, accessed 13.05.2015.
5. Almossawi A., Lim K., Sinha T. *Analysis tool evaluation: Coverity Prevent. Final Report*. Pittsburgh, PA, Carnegie Mellon University, 2006, 19 p.
6. Ivannikov V.P., Belevantsev A.A., Borodin A.E., Ignatiev V.N., Zhurikhin D.M., Avetisyan A.I., Leonov M.I. Static Analyzer Svace for Finding of Defects in Program Source Code. *Trudy Instituta sistemnogo programmirovanija RAN = Proceedings of ISP RAS*, 2014, vol. 26, no. 1, pp. 231–250. DOI: 10.15514/ISPRAS-2014-26(1)-7 (in Russian)
7. Hovemeyer D., Pugh W. Finding Bugs is Easy. *ACM Sigplan Notices*, 2004, vol. 39, no. 12, pp. 92–106. DOI: 10.1145/1052883.1052895
8. Nethercote N., Seward J. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan Notices*, 2007, vol. 42, no. 6, pp. 89-100. DOI: 10.1145/1273442.1250746
9. Serebryany K., Bruening D., Potapenko A., Vyukov D.. AddressSanitizer: A Fast Address Sanity Checker. *Proc. of the 2012 USENIX Annual Technical Conference (USENIX ATC'12), Boston, MA, USA, June 13–15, 2012*, USENIX Association, 2012, pp. 309–318. Available at: https://www.usenix.org/system/files/conference/atc12/atc12-final39.pdf, accessed 13.05.2015.
10. Serebryany K., Iskhodzhanov T. ThreadSanitizer — data race detection in practice. *Workshop on Binary Instrumentation and Applications (WBIA'09), New York, NY, USA, December 12, 2009*, ACM New York, NY, USA, 2009, pp.62–71. DOI: 10.1145/1791194.1791203
11. Cha S.K., Avgerinos T., Rebert A., Brumley D. Unleashing Mayhem on Binary Code. *Proc. of the 33rd IEEE Symposium on Security and Privacy, San Francisco, CA, USA, May 20–23, 2012*. IEEE, 2012, pp. 380–394. DOI: 10.1109/SP.2012.31
12. Cadar C., Dunbar D., Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *Proc. of the USENIX Symposium on Operating System Design and Implementation (OSDI'08), San Diego, CA, USA, December 8–10, 2008*, USENIX Association, 2008, pp. 209–224. Available at: https://www.usenix.org/legacy/events/osdi08/tech/full_papers/cadar/cadar.pdf, accessed 13.05.2015.
13. King J.C. Symbolic execution and program testing. *Communications of the ACM*, 1976, vol. 19, no. 7, pp. 385–394. DOI: 10.1145/360248.360252

14. Cousot P., Cousot R. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. *Proc. of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL'77), Los Angeles, CA, USA, 1977*. ACM New York, NY, USA, 1977, pp. 238–252. DOI: 10.1145/512950.512973

15. Reps Th., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability. *Proc. of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Francisco, CA, USA, 1995*. ACM New York, NY, USA, 1995, pp. 49–61. DOI: 10.1145/199448.199462

16. Sen K., Marinov D., Agha G. CUTE: a concolic unit testing engine for C. *ACM SIGSOFT Software Engineering Notes*, 2005, vol. 30, iss. 5, pp. 263–272. DOI: 10.1145/1095430.1081750

17. The LLVM compiler infrastructure. Available at: http://llvm.org, accessed 14.05.2015.

18. Clang: a C language family frontend for LLVM. Available at: http://clang.llvm.org, accessed 15.05.2015.

19. Clang Static Analyzer. Available at: http://clang-analyzer.llvm.org, accessed 15.05.20157.