УДК 004.4’2

МЕТОД РЕЗЮМЕ ДЛЯ РАЗРАБОТКИ УНИВЕРСАЛЬНОГО МНОГОЦЕЛЕВОГО АНАЛИЗАТОРА КОДОВ ПРОГРАММ С ВОЗМОЖНОСТЬЮ ОБНАРУЖЕНИЯ РАЗЛИЧНЫХ КЛАССОВ ДЕФЕКТОВ В ПРОГРАММАХ, СОЗДАННЫХ С ИСПОЛЬЗОВАНИЕМ ЯЗЫКОВ С И С++

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

[1]МГТУ им. Н.Э. Баумана, Москва, Российская Федерация
e-mail: rtn51@mail.ru

[2]Московский исследовательский центр Samsung, Москва, Российская Федерация
e-mail: alexey.v.sidorin@yandex.ru

Приведено подробное описание разработанного метода межпроцедурного анализа с использованием резюме для метода символьного выполнения. Этот метод реализован на модели анализатора Clang Static Analyzer, что позволило использовать метод резюме для разработки универсального многоцелевого анализатора с возможностью поиска различных классов дефектов в программах, разработанных с использованием языков С и С++. Описаны методы сбора и применения резюме для метода символьного выполнения, перевода символьных значений из контекста вызывающей функции в контекст вызываемой и обратно. Разработан и описан метод построения отчета о дефекте с использованием метода резюме для межпроцедурного анализа. Исследование проведено в целях построения анализатора, способного осуществлять межпроцедурный анализ крупных программных комплексов масштаба операционной системы Android за приемлемое время. Приведены данные измерений времени анализа и выполнено сравнение результатов методов резюме и метода встраивания.

***Ключевые слова*:** метод резюме, межпроцедурный анализ, символьное выполнение, Clang Static Analyzer, поиск дефектов, построение отчета, C/C++.

SUMMARY-BASED INTERPROCEDURAL ANALYSIS METHOD FOR IMPLEMENTATION IN MULTI-PURPOSE STATIC C/C++ CODE ANALYZER

T.N. Romanova[1], **A.V. Sidorin**[1,2]

[1]Bauman Moscow State Technical University, Moscow, Russian Federation
e-mail: rtn51@mail.ru

[2]Samsung R&D Institute Rus, Moscow, Russian Federation
e-mail: alexey.v.sidorin@yandex.ru

The paper gives a detailed describtion of the summary-based interprocedural analysis method developed for the symbolic execution method. This method is implemented in the analyzer model of Clang Static Analyzer, using its framework. This allows using the summary-based method for developing a multi-purpose analyzer with the ability to search for various defects in programs with C and C++ software codes. The author describe summary collecting methods and their application in symbolic execution, as well as in translating symbolic meanings from a callee context to a caller one and vice versa. The method of the defect report building with the help of the summary-based interprocedural analysis method is also developed and presented in the paper. The research aims at developing an analyzer suitable for doing an interprocedural

analysis of large software projects, such as Android, in reasonable time. The results of measuring the analysis time are given. The summary-based and inlining methods are compared.

***Keywords*:** summary-based method, interprocedural analysis, symbolic execution, Clang Static Analyzer, search for defects, report building, C/C++.

Введение. Метод символьного выполнения, впервые описанный в 1976 г. Дж. Кингом [1], по-прежнему активно используется в различных инструментах анализа кода. Суть метода символьного выполнения заключается в разбиении множества входных данных на классы эквивалентности, что позволяет оперировать при анализе не отдельными входными значениями (их число может быть очень большим и экспоненциально возрастает в зависимости от числа входных аргументов) и их перебором, а целыми классами эквивалентности, число которых также может оказаться большим, но не превышает общее число комбинаций отдельных входных значений. Однако, как правило, число классов эквивалентности комбинаций входных данных оказывается значительно ниже числа всех возможных комбинаций входных данных, что резко увеличивает возможности анализатора по обработке путей выполнения.

Основное преимущество метода символьного выполнения — 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: хорошо зарекомендовали себя инструменты, использующие подход concolic testing (символьно-конкретное — concrete+symbolic). При использовании этого подхода на основе символьного выполнения генерируются тестовые данные, при применении которых программа проявляет некорректное поведение.

Наряду с преимуществами, метод имеет ряд недостатков. Так, существует проблема экспоненциального роста числа проходимых путей (path explosion), приводящая к проблемам с масштабируемостью метода [2]. Возникают также проблемы при моделировании циклов, поскольку зачастую число итераций цикла точно неизвестно — оно также является символьной величиной. Тем не менее метод символьного выполнения активно используется, в том числе, многими популярными инструментами анализа программ. Таким образом, разработка подходов для улучшения указанного метода — актуальная и практически важная задача.

Одна из основных проблем с масштабируемостью метода — трудности, возникающие при реализации межпроцедурного анализа. В случае выполнения контекстно-чувствительного межпроцедурного анализа методом встраивания функции граф выполнения быстро разрастается. Это резко замедляет анализ и увеличивает потребление

памяти анализатором. Для преодоления этой проблемы в настоящей статье применен метод резюме для контекстно-чувствительного межпроцедурного анализа. К наиболее известным инструментам, использующим метод анализа на основе символьного выполнения, относятся, например, KLEE, Mayhem и Otter [3–5]. Различным способам улучшения его производительности, в том числе, использованию резюме, посвящено много научных работ. Исследователи Научно-технического университета Китая применили подход резюме для символьного выполнения, чтобы осуществить поиск утечек памяти в коде программ на языке C [6]. Многие исследователи пытались использовать подход резюме для решения узких задач: от поиска дефектов, связанных с многопоточностью [7], до изучения активности объектов кучи [8].

Цель настоящей работы — построение многоцелевого статического анализатора для языков C и C++, пригодного для решения различных классов задач, с возможностью реализации широкого класса проверок.

Модель анализатора. Построение анализатора, досконально моделирующего все низкоуровневые действия, являющиеся результатом выполнения операторов программы (таких, как содержимое регистров, распределение памяти), не только трудоемко и ресурсоемко, но и не дает значительных преимуществ при анализе программы. Поэтому для анализа, в том числе при анализе методом символьного выполнения, используются абстрактные модели.

В качестве целевой абстрактной модели анализатора для проведения эксперимента в настоящей работе выбран Clang Static Analyzer (CSA) [9], являющийся модулем компилятора Clang [10], который в свою очередь представляет собой основной компилятор проекта LLVM [11]. Компилятор Clang является фронтэндом — транслятором в промежуточное представление с поддержкой языков C, C++, Objective-C и Objective C++, Clang Static Analyzer — статический анализатор, поддерживающий анализ исходного кода на всех перечисленных языках. Однако в настоящей работе реализация метода резюме была осуществлена только для языков C и C++. Языки Objective-C и Objective-C++ не были рассмотрены в связи с трудностями при поисках крупных открытых программных комплексов, разработанных с использованием этих языков.

Clang Static Analyzer поддерживает различные виды анализа: с использованием AST; на основе графа потока выполнения и символьного выполнения, а также допускает комбинации видов анализов. Широкий выбор методов анализа и простота внедрения новых методов позволяет применять его в различных производных работах [12, 13]. В настоящем исследовании рассмотрен именно анализ на основе символьного выполнения. В CSA подобный анализ реализован как межпроцедурный с помощью метода встраивания, поэтому особенно интересным выглядит сравнение реализации метода встраивания и метода резюме.

В терминологии CSA, описанной в работе [14], выполнение программы представляет собой множество последовательных переходов между состояниями из одного состояния в другие. Каждому состоянию соответствует точка выполнения (ProgramPoint), для которого это состояние актуально[1]. Переходы между состояниями обусловлены либо эффектами интерпретации отдельных выражений, определенными стандартом языка, либо событиям, связанными с выполнением проверок проверяющими модулями (Checker). Из одной точки выполнения может идти один и более переход в другое. Несколько переходов происходит в случаях, когда условие перехода невозможно однозначно разрешить в пользу выбора какой-либо одной ветви выполнения, например, при обработке условных операторов (это и есть разделение на классы эквивалентности). Кроме того, проверки также могут разделять состояние программы, сохраняя в разных структурах состояния различающиеся данные состояния. Результирующее множество узлов в виде состояний и переходов из одного состояния программы в другое образует граф выполнения программы (ExplodedGraph).

За базовое моделирование без каких-либо проверок корректности исходного кода отвечает ядро анализатора. Ядро анализатора представляет собой, во-первых, набор методов, связанных с построением графа состояний выполнения программы (класс CoreEngine), а, во-вторых, набор методов, отвечающих за моделирование эффектов, специфичных для языка программирования, т.е. моделирование эффектов выражений и правил их выполнения (ExprEngine). Кроме того, в процедуре построения графа выполнения могут принимать участие проверяющие модули, анализируя события, которые наступают в процессе выполнения. Эти проверки могут останавливать построение графа на заданном пути в случае обнаружения критического дефекта, разделять состояние программы и вносить в него дополнительную информацию для работы проверяющего модуля.

Структура состояния — представление состояния программы в точке выполнения. Структура состояния включает следующие данные.

1. Содержимое памяти программы (модель памяти, *RegionStore*) [15], представляемое как отображение между регионами памяти и символьными значениями, связанными с этими регионами. Для создания записи в модели памяти необходимо провести непосредственное связывание региона памяти и его значения, например, при обработке оператора присваивания. Операциями, изменяющими содержимое модели памяти, являются непосредственное связывание региона со значением (выполняющееся, например, при присваивании переменной значения),

[1]Здесь и далее в скобках приведены названия соответствующих классов фреймворка CSA.

пометка регионов памяти как не содержащих первоначальное значение (инвалидация) и удаление имеющихся привязок, происходящее при потере регионом памяти активности, а также иногда используемое вместо инвалидации. Если необходимо получить символьное значение для региона памяти, не имеющего записи в модели памяти (например, для аргументов функции), происходит неявное связывание с помощью символьного значения специального вида, при этом записи в модели памяти не создается.

2. Окружение (*Environment*) ставит в соответствие активным выражениям их как левосторонние символьные значения, так и правосторонние (левосторонними значениями выражений являются абстрактные области памяти кода программы, где располагаются выражения, а правосторонними — вычисленные символьные значения выражений).

3. Нетипизированное хранилище (Generic Data Map, GDM) — контейнер для хранения данных проверок, а также для хранения некоторых специфических структур данных ядра анализатора, связанных с состоянием программы. Наиболее важными такими данными является карта соответствия символов и их диапазонов возможных значений, с помощью которой проводится анализ достижимости. За помещение данных в GDM и удаление их оттуда отвечают непосредственно использующие эти данные модули — проверки и модули ядра анализатора (в частности, ответственный за арифметические и логические вычисления решатель ConstraintManager).

Символьное значение в терминологии CSA — это абстракция переменного или константного значения какого-либо типа данных. Абстрактным значением можно представить, например, значение выражения, содержимое области памяти, саму область памяти и указатель на нее. В модели CSA символьное значение может иметь несколько основные видов: целочисленная константа; символьное выражение; регионы памяти.

Эффекты, учитываемые в резюме функции. Каждый оператор при выполнении производит эффект, заключающийся в изменении состояния программы. В случае анализа речь идет о моделировании эффектов операторов, т.е. о моделировании действия, которое моделируемый оператор оказывает на модель состояния программы.

С учетом описанной модели анализатора сократить время анализа при использовании резюме в сравнении с временем анализа при применении метода встраивания можно 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, так как 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 отразится в изменении состояния. Учет этих упрощений и ограничений анализатора позволяет устранить непроизводительные затраты времени, поскольку при применении резюме непроизводительные вычисления не выполняются повторно. Возможно, что некоторые эффекты самого применения резюме не могут быть учтены моделью анализатора и также будут отнесены к непроизводительным вычислениям. Тем не менее время, затрачиваемое на применение резюме, по-прежнему не будет превышать время, требуемое на анализ вызова функции методом встраивания. Это объясняется тем, что набор эффектов, получаемых в результате применения резюме, включается строго или совпадает с набором эффектов, моделируемых при анализе методом встраивания.

В настоящей работе рассмотрен следующий набор эффектов, влияющих на состояние анализируемой программы в процессе ее выполнения.

1. Принятие решений о выборе пути выполнения. Выбор пути выполнения сопровождается наложением ограничений на символьные значения, относительно которых принимается решение о выборе пути. Если эти символьные значения содержат ссылки на внешние по отношению к вызываемой функции регионы памяти, то накладываемые ограничения должны быть отражены в резюме. Кроме того, как было показано выше, каждое принятие решения влияет на присутствие и порядок операторов в последовательности выполнения, а следовательно, и на набор эффектов, включаемых в резюме. Наконец, наложение ограничений на входные данные функции в зависимости от выбора пути выполнения позволяет сохранить контекстную чувствительность при анализе, поскольку определенные пути выполнения могут быть достижимы лишь при ограниченном наборе входных значений аргументов функции и значений, находящихся во внешней по отношении к ней памяти.

2. Модификация регионов памяти с областью видимости, отличной от локальной, т.е. находящихся в статической или глобальной области видимости, принадлежащих куче, а также модификация аргументов, переданных по неконстантному указателю или неконстантной ссылке, и областей памяти, относящихся к ним (возможно, с использованием арифметики указателей).

3. Инвалидация регионов памяти, т.е. пометка некоторых регионов как изменивших значение на неизвестное. Обычно выполняется

при моделировании оператора, все эффекты которого учесть по каким-либо причинам невозможно, например, при вызове функции с недоступным определением.

4. Возврат вызываемой функцией некоторого значения, которое связывается с выражением вызова функции как элемент окружения.

5. Пометки проверяющих модулей, к которым относятся пометки символов, регионов памяти и символьных значений, события, которые необходимо проверить отложенно, когда контекст вызываемой функции станет достаточно определен для того, чтобы утверждать наличие потенциального дефекта, и иные действия, связанные с процедурами проверок (в зависимости от логики работы проверяющего модуля).

Поскольку проверяющие модули самостоятельно отвечают за свои данные, логику обработки резюме для проверок имеет смысл включать непосредственно в логику работы этих модулей.

Порождение новых ветвей выполнения программы и отсечение недостижимых путей. Один из результатов сбора резюме — пары регион памяти–символьное значение. В результате актуализации символьных значений из резюме могут получиться символьные значения, имеющие диапазон, отличный от диапазона этого символьного значения в контексте вызывающей функции. Это является следствием того, что при моделировании условий внутри вызываемой функции может произойти разделение входных данных функции (аргументов и внешних переменных) на классы эквивалентности. Рассмотрим пример.

Пусть вызывается функция

```
1  void f(int a) {
2    if (a > 10) {
3      ...
4    } else {
5      ...
6    }
7  }
```

В результате анализа этой функции в ее резюме войдут две ветви выполнения. В первой ветви a будет иметь диапазон конкретных значений от INT_MIN до 10, во второй — от 11 до INT_MAX.

Пусть происходит вызов функции при a 5. . . 13. Тогда в первой создаваемой ветви выполнения a будет иметь диапазон 5. . . 10, а во второй — 11. . . 13.

Далее пусть в вызывающей функции a имеет диапазон конкретных значений 5. . . 9. Тогда в первой ветви выполнения a будет иметь диапазон 5. . . 9, а во второй ветви выполнения множество значений будет пустым. Наличие символического значения с пустым множеством допустимых значений означает, что вторая ветвь недостижима

и может не рассматриваться. Действительно, при вызове функции для заданного a выполняется только else-ветвь, но не if-ветвь.

Введем обозначения:

- число n ветвей выполнения, полученных в резюме анализа вызываемой функции;
- номер i ветви выполнения, $0 \leqslant i < n$;
- число p_i символьных правосторонних входных значений в i-й ветви выполнения;
- номер j символьного значения, $0 \leqslant j < p_i$;
- символьное значение s_{ij} с номером j в i-й ветви выполнения;
- множество значений $r_{\text{входные } ij}$ для значения s_{ij} в контексте вызывающей функции в точке непосредственно перед вызовом функции;
- множество значений $r_{\text{резюме } ij}$ для значения s_{ij} в контексте вызываемой функции;
- состояние программы $state_{\text{входное}}$ в контексте вызывающей функции в точке непосредственно перед вызовом функции;
- состояние программы $state_{\text{выходное}}$ после вызова функции (после применения резюме).

Тогда при применении i-й ветви выполнения резюме $\forall i \in [0; n]$, $\forall\, j \in [0; p_i] : r_{\text{выходные } ij} = r_{\text{входные } ij} \cap r_{\text{резюме } ij}$, т.е. результирующее множество — пересечение множеств входных конкретных значений символического значения и множества конкретных значений символического значения из применяемой ветви резюме.

Если результирующее множество конкретных значений является пустым хотя бы для одного символического значения, то такая ветвь выполнения недостижима и не принимается в рассмотрение, что может быть выражено следующей формулой: $(\exists i, j : r_{\text{входные } ij} \cap r_{\text{резюме } ij} = \varnothing) \Rightarrow (state_{\text{входное}} \nrightarrow state_{\text{выходное}})$.

Сбор данных для создания резюме. Пусть некоторое значение относится к региону памяти. Поскольку при передаче аргумента функции не по значению его значение может измениться, необходимо различать входное значение региона до его изменения в функции и выходное значение. Для получения информации о разбиении данных на классы эквивалентности можно отслеживать события ветвлений (assume), однако это неудобно на практике, так как, во-первых, требует отдельного отслеживания событий изменений региона для разделения входных и выходных значений, а, во-вторых — активного участия сборщика резюме в процессе принятия решения о ветвлении. Вместо этого в настоящем решении предложен и использован подход, основанный на проверке активности символов и регионов. Входные данные, внешние по отношении к анализируемой функции, всегда являются символическими, тогда отслеживая событие потери актуальности (активности), можно определить диапазон возможных значений символа,

связанного с данным регионом, в данной ветви выполнения. При этом первое событие потери активности соответствует диапазону входных значений, а последнее — диапазону выходных значений, если они совпадают, то никаких присваиваний или инвалидаций региона входного символа не было, входной диапазон также является выходным. Такой метод позволяет избежать использования сложных алгоритмических схем для сбора входных и выходных значений аргументов функций, входных значений глобальных регионов памяти.

Сбор выходных значений иногда требуется проводить по окончании пути выполнения функции. Это необходимо для обработки символьных значений, привязанных к региону памяти непосредственным присваиванием или иным видом связывания. Для этого используется итерация по хранилищу с сохранением диапазонов внешних по отношении к функции регионов памяти в резюме.

Инвалидация региона памяти в терминологии CSA означает связывание с данным регионом нового символа без наложенных на него ограничений, т.е. способного принимать произвольные значения. Поскольку символьные значения, связанные с регионами памяти, обрабатываются при завершающем проходе по хранилищу, а значения регионов, актуальные до инвалидации, обрабатываются по событию потери активности, непосредственно предшествующему событию связывания нового значения, инвалидация обрабатывается автоматически, дополнительных действий для обработки инвалидаций регионов памяти не требуется.

Обработка события возврата функцией значения достаточно тривиальна. Результирующее символьное значение сохраняется в резюме целиком, а ограничения, накладываемые на него и на его части, обрабатываются отдельно.

За хранение данных проверок проверяющие модули отвечают самостоятельно. Основными видами данных проверок являются отметка отложенной проверки и данные состояния проверки. Отложенные проверки используются для выдачи предупреждений в тех ситуациях, когда вследствие отсутствия данных о контексте вызова невозможно однозначно утверждать наличие дефекта или его отсутствие. Данные состояния необходимы для построения нового состояния проверки при применении резюме.

Актуализация символьных значений. В результате сбора резюме на предыдущем шаге получено некоторое множество регионов памяти, с которыми связаны некоторые символьные значения. Кроме того, регионы памяти могут входить в символьные значения как их составная часть. Однако полученные регионы памяти адресуются в контексте объявлений имен внутри функции. В контексте вызывающей функции эти регионы могут иметь уже другое значение, т.е. регионы, используемые внутри функции, являются относительными по отношению к

вызывающей функции. Так, в контексте вызываемого метода класса регион памяти, связанный с указателем this, будет адресоваться безотносительно какого-либо объекта, а в контексте вызывающей функции — будет регионом с находящимся внутри объектом, метод которого вызывается. Аналогично, аргумент функции, фигурирующий в ней как самостоятельная переменная (и, соответственно, как самостоятельный регион памяти), может быть подрегионом в контексте вызывающей функции — полем структуры, элементом массива. С регионом памяти в контексте вызываемой функции может быть связан не символ, относящийся к региону памяти, а константа, символьное выражение или иное значение, не имеющее в своей основе регион аргумента. Все это означает, что для корректного применения резюме необходимо проводить актуализацию символьных значений: их перевод из контекста имен и значений вызываемой функции в контекст имен и значений вызывающей функции.

В соответствии с типами символьных значений можно выполнять актуализацию символьных значений в зависимости от их типа.

Константные значения не изменяются при их актуализации, поскольку они не содержат элементов, зависящих от контекста. Вместе с тем, типы константного значения в контексте вызываемой функции и в контексте вызывающей функции могут быть различающимися, поэтому может понадобиться осуществление дополнительного приведения типов для константы.

Символьные значения, относящиеся к регионам памяти, можно актуализировать, используя следующие правила для различных категорий регионов памяти.

Регионы, относящиеся к пространству аргументов вызываемой функции. Можно выделить два различных случая передачи в зависимости от того, является ли передаваемый тип ссылочным или типом указателя, либо не является. Если аргумент передается по ссылке, то левостороннее значение фактического аргумента становится левосторонним значением формального параметра, а правостороннее значение фактического параметра — правосторонним значением формального параметра. Это означает, что значение объекта в результате выполнения функции может отличаться от значения на момент вызова, так как в результате передачи по ссылке с фактическим аргументом может быть связано другое значение.

Если аргумент передается по указателю, то правостороннее значение фактического параметра становится правосторонним значением формального параметра. Для левосторонних значений это утверждение неверно: выполнение присваивания формальному аргументу внутри функции не влияет на значение фактического аргумента. Однако, если указываемый тип не является константным, то в результате вызо-

ва функции может измениться привязка региона памяти, адрес которой задает указатель, и его субрегионов.

Если аргумент передается по значению, то правостороннее значение фактического параметра становится правосторонним значением формального параметра.

В случае передачи в качестве аргумента структурных типов по ссылке или указателю правила изменения их полей аналогичны. Так, при передаче по значению структуры, содержащей ссылку в качестве поля, такое поле можно полагать передающимся по ссылке. Фактически для структурных типов можно считать, что передается набор аргументов по их типам с тем отличием, что при потере актуальности их окружающего региона памяти эти поля также могут потерять актуальность.

Регионы памяти внешней области видимости. Кроме передаваемых аргументов вызываемая функция может иметь доступ к другим данным: переменным, имеющим области видимости выше, чем область видимости функции. К этим регионам относятся глобальные переменные, члены класса и его предков, если вызываемой функцией является метод класса. Их изменения и наложения ограничений на них также необходимо отслеживать.

Регионы глобальных переменных (включая статические) сохраняются неизменными, дополнительные действия по их актуализации предпринимать не требуется, поскольку регионы глобальных переменных не зависят от контекста вызова и связаны лишь с объявлением соответствующих переменных.

Методам класса, включая конструкторы, деструкторы и операторы — члены класса, могут быть доступны для чтения и записи поля как самого класса, так и его предков в иерархии наследования. При актуализации регионы памяти, относящиеся к статическим полям класса, не изменяются, так как они не относятся к конкретному объекту поля и, следовательно, их адресация не зависит от контекста вызова. Нестатические поля в контексте вызываемого метода адресуются относительно условного объекта, связанного с указателем this, поскольку в контексте вызываемой функции фактическим объектом будет объект, метод которого вызывается, при актуализации эти поля становятся соответствующими полями вызываемого объекта. Если определение нестатического поля принадлежит классу, определение которого находится ниже по иерархии наследования, то поле отображается в соответствующее поле родительского класса объекта в соответствии с компоновкой полей дочернего класса.

Актуализация составных и служебных символьных значений. Актуализация символьных значений, обозначающих бинарные операции над символами (бинарные символьные значения), выполняется

следующим образом. Сначала актуализируются правая и левая части символьного значения. Если результатом бинарной операции является константа, то эта константа становится результирующим символьным значением. В противном случае результат актуализации — новое символьное значение в виде бинарной операции. Фактически бинарные символы актуализируются рекурсивно с возможными упрощениями в виде свертывания отдельных элементов или всего выражения в константу. Это свертывание объясняется уменьшением диапазонов входных значений каждого символьного значения, входящего в бинарное символьное значение, при уточнении контекста вызова.

Актуализация метасимволов представляет собой отдельную подзадачу. Под метасимволами понимают символические значения, относящиеся к объекту анализа, но не являющиеся его непосредственной характеристикой. Выделение метасимволов связано с тем, что источником метаданных является не ядро анализатора, а сами проверяющие модули, которые связывают необходимую информацию в виде метасимволов с интересующими их регионами памяти и выражениями. Соответственно, регионы памяти и выражения могут иметь более одного связанного с ними метасимвола. Поскольку каждый проверяющий модуль может реализовывать свой подход к использованию метаинформации и обозначениям интересующих его объектов, для актуализации метасимволов применяется отдельное событие, на которое должны подписываться проверяющие модули, использующие метаданные. В результате проверяющие модули самостоятельно обновляют представление связанных с ними символьных значений и нарушение принципа инкапсуляции данных не происходит. В качестве примера можно привести проверку длины строки. Длина строки не входит в число данных, о которых известно анализатору — за ее моделирование отвечает проверяющий модуль, связывающий символьное значение (метасимвол) с регионом памяти этой подстроки. Таким образом, задача проверяющего модуля при актуализации метазначения, связанного с регионом, — поиск существующего метазначения для этого региона и его возврат.

Построение и актуализация сложных структур данных в резюме проводится с помощью сохранения цепочки родительских регионов. Для каждого региона памяти, имеющего связанное с ним символьное значение (как явно, так и неявно), строится упорядоченный список родительских регионов, начиная от региона верхнего уровня — $M_0 \dots M_n$. При этом регионы $M_1 \dots M_n$ могут быть только регионами элемента массива, регионами поля структуры или регионами данных базового класса. Этот список сохраняется в резюме и используется для актуализации значений по следующему алгоритму.

1. Актуализация региона M_0.

2. Для всех регионов $M_1 \dots M_n$ согласно их положению в списке:

- если M_i — регион элемента массива, то в резюме сохраняется символьное значение индекса, а результат актуализации — элемент массива от региона, полученного на $(i-1)$-м шаге алгоритма, с символьным значением индекса, определенным в результате актуализации сохраненного значения индекса;
- если M_i — регион поля структуры, то в резюме сохраняется объявление поля структуры, а результат актуализации — поле структуры от региона, полученного на $(i-1)$-м шаге алгоритма, с тем же определением;
- если M_i — регион данных базового класса, то в резюме сохраняется ссылка на определение базового класса, а результат актуализации — подрегион данных базового класса от региона, полученного на $(i-1)$-м шаге, с тем же определением.

Цепочка родительских регионов может строиться как явно — при построении резюме для региона памяти, так и неявно — при сохранении и последующем разборе символьного значения, для которого строится резюме.

Применение резюме проверяющими модулями. Схемы применения резюме, описанные выше, затрагивают отсечение недостижимых ветвей выполнения программы и уточнение множества конкретных значений для символьных значений. Однако для того, чтобы анализатор имел возможность выполнять проверки при вложенном вызове функции, необходима доработка проверяющих модулей. Для получения проверяющими модулями возможностей анализа при использовании резюме введем две дополнительных функции обратного вызова. Первая (evalSummaryPopulate) вызывается для сбора резюме проверяющим модулем, вторая (evalSummaryApply) — при применении резюме.

Проверяющий модуль, имеющий возможность выполнения действий при событии SummaryPopulate, должен сохранить информацию, которая может понадобиться для обновления состояния или выполнения отложенной проверки. Информация, содержащаяся в резюме, не освобождается до окончания работы анализатора, поэтому проверяющий модуль может использовать произвольный формат хранения данных, лучшим образом отвечающий задаче проверки. Как показала практика модификации проверяющих модулей, для каждой проверки, проводимой модулем, в GDM обычно помещается две дополнительных записи, которые затем используются для заполнения резюме: для обновления состояния и отложенной проверки. В качестве примера рассмотрим проверку двойного закрытия файлового дескриптора. В резюме помещаются две секции: первая отвечает за обновление состояния дескриптора (является ли он открытым или закрытым), а вторая — за выполнение отложенной проверки (в ней запоминаются

события закрытия дескрипторов, исходное состояние которых неизвестно).

При обработке события SummaryApply проверяющий модуль должен выполнить обновление состояния в соответствии с информацией, хранящейся в выбранной ветви резюме. Так, если при выполнении вызова функции дескриптор был закрыт, он должен быть помечен как закрытый в состоянии вызывающей функции. Если в контексте вызывающей функции уже известно, что дескриптор закрыт, то отложенная проверка должна выдать предупреждение. Кроме того, если проверяющий модуль использует метаданные для хранения информации о состоянии контролируемых объектов, он может потребовать реализации функциональности актуализации метасимвола. Для этого вводится функция обратного вызова evalSummarySVal. Реализующие эту функцию модули должны определять, на какое символьное значение метаданных отображается актуализированный метасимвол, и возвращать это символьное значение в качестве результата функции обратного вызова.

Построение отчета о дефекте. Построение отчета — важный элемент работы статического анализатора. Недостаточно просто найти дефект и указать место его возникновения. В случае анализа путей выполнения программы между различными значимыми для дефекта точками программы (например, открытие и закрытие файла) может проходить длинный путь через большое количество операторов. При этом путь выполнения может включать в себя условные операторы, циклы и вызовы других функций. Однако субъективная сложность отслеживания пути выполнения быстро растет при увеличении длины пути. Если дефект не локализован в пределах нескольких строк кода или небольшой функции, то разработчику невозможно определить путь выполнения, на котором проявляется дефект. Таким образом, при анализе программы методом анализа ее путей выполнения необходимо выполнять построение подробного отчета, однозначно указывающего условия, при которых возникает дефект, и соответствующий им путь выполнения.

При использовании межпроцедурного анализа методом встраивания путь, проходимый внутри функции, отображается в графе выполнения как часть общего пути выполнения, поэтому проблем при построении пути при генерации отчета не возникает. Однако при применении метода резюме существует проблема потери информации о части пути, проходимом внутри вызываемой функции, поскольку явного построения поддеревьев графа выполнения программы более не происходит. В результате замены встраивания функции на применение ее резюме граф выполнения программы изменяется следующим образом. Вместо подграфа вложенного вызова в графе выполнения

появляются отдельные точки выполнения, условно соответствующие конечным точкам подграфа вложенного вызова функции. В связи с этим при построении отчета о найденном дефекте нельзя традиционным способом указать путь, который прошло выполнение программы при вызове функции. Для построения отчета при межпроцедурном анализе методом резюме в настоящей работе предложен следующий метод.

Задача проверяющих модулей — отложенная проверка: на основе резюме вызываемой функции и состояния на момент вызова проверяющий модуль должен сделать вывод о необходимости выдачи предупреждения (о срабатывании). Существует два варианта срабатывания проверки. В первом варианте выдача предупреждения происходит внутри вызываемой функции. Допустим, что уровень вложенности вызова равен единице. Такое допущение возможно, поскольку к нему можно свести стек вызовов произвольного уровня вложенности. В этом случае конечной точкой трассы является узел графа выполнения вызываемой функции. Если этот узел известен, то от него можно построить трассу до корневого узла графа выполнения. Такая трасса-подграф будет путем, проходимым внутри функции при выполнении программы до критической точки.

Второй вариант предполагает построение пути при необходимости показать путь внутри вызванной функции целиком, от точки входа до точки выхода. Эта задача сводится к первой при условии, что в качестве конечной точки выбирается лист графа выполнения, соответствующий выбранной ветви выполнения резюме. Таким образом, для корректного построения пути при моделировании вложенного вызова функции достаточно иметь информацию об узле графа выполнения вызываемой функции, для которого выдается срабатывание, или о листе графа выполнения, к которому относится выбранный путь выполнения при применении резюме. Для этого достаточно сохранить ссылку на этот узел. В случае построения пути изнутри вызываемой функции за хранение ссылки может отвечать проверяющий модуль, генерирующий срабатывание. При построении полного пути по вложенному вызову ссылку на лист графа выполнения вызываемой функции можно хранить в узле применения резюме в качестве дополнительной информации.

Поясним описанный метод примером. Рассмотрим функцию следующего вида:

```
1  void close_file(bool flag, FILE *f) {
2      ...
3      if (flag)
4          fclose(f);
5  }
```

Пусть задача проверяющего модуля — проверка двойного закрытия файлового дескриптора. Резюме функции close_file( ) будет состоять из двух ветвей: в первой ветви (flag ≡ false) никакой дополнительной информации не хранится, во второй (flag ≡ true) имеется событие закрытия файла. При анализе приведенной функции вне контекста вызывающей функции проверяющий модуль не может доказать, что внутри этой функции дескриптор закрывается во второй раз, поскольку на всех путях выполнения дескриптор закрывается не более одного раза. Отсюда следует необходимость выполнения отложенной проверки при применении резюме этой функции, когда контекст вызова позволит однозначно определить состояние файлового дескриптора при вызове fclose(f) внутри close_file( ).

Пусть имеется функция, вызывающая close_file( ):

```
1  void double_close(FILE *file) {
2      fclose(file);
3      close_file(true, file);
4  }
```

На момент вызова функции close_file( ) (и применения резюме) дескриптор file закрыт, т.е. его состояние однозначно определено. Это означает возможность выполнения отложенной проверки, которая покажет, что происходит закрытие уже закрытого дескриптора. В этом случае происходит срабатывание, путь к которому необходимо построить изнутри вызываемой функции. Применение этой схемы в рассматриваемом примере проиллюстрировано на рис. 1.

Рассмотрим более общий случай произвольной вложенности вызовов. В случае построения пути изнутри вызываемой функции информации об узле срабатывания может оказаться недостаточно, так как по одному узлу невозможно восстановить всю цепочку вложенных вызовов. Это объясняется тем, что резюме функции верхнего уровня хранит ссылки только на узлы следующего уровня вложенности. Для решения такой задачи проверяющий модуль должен самостоятельно хранить стек вызовов, для которого строится путь выполнения.

Рассмотрим пример. Модифицируем представленный в предыдущем примере набор функций, добавив в него промежуточный вызов:

```
1  void close_file(bool flag, FILE *f) {
2      . . .
3      if (flag)
4          fclose(f);
5      ...
6  }
7
8  void potential_double_close(bool flag, FILE *file) {
```

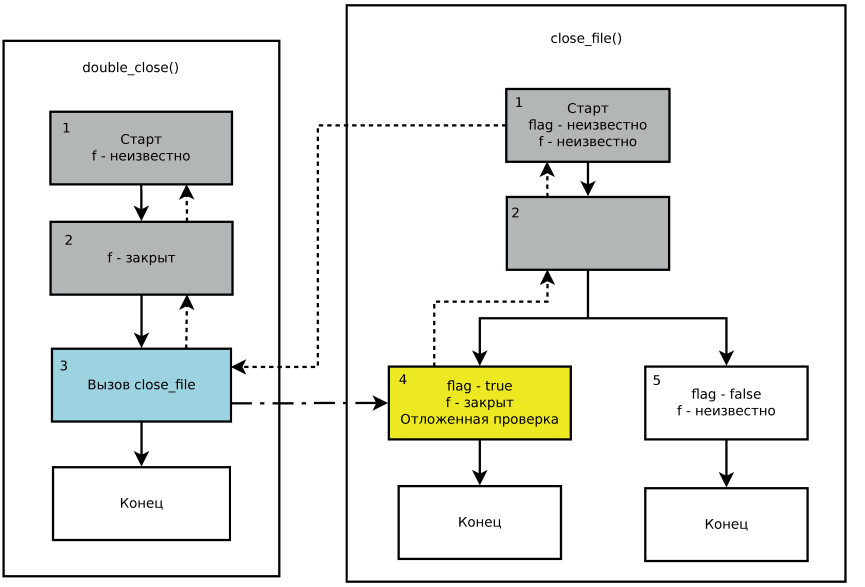

Рис. 1. Схема построения отчета при использовании резюме вложенного вызова функции (поток управления внутри функции показан сплошными линиями, отношение отложенной проверки в заданном узле — штрихпунктирной, трасса построения отчета — пунктирной)

```
9       close_file(flag, file);
10 }
11
12 void double_close(FILE *file) {
13       fclose(file);
14       potential_double_close(true, file);
15 }
```

Резюме функции close_file( ) аналогично предыдущему случаю. Резюме функции potential_double_close( ) состоит из двух ветвей: в первой ветви (flag ≡ false) никакой дополнительной информации не хранится, во второй (flag ≡ true) имеется отложенная проверка события закрытия файла, содержащая ссылку на узел № 4 графа выполнения функции close_file( ). В качестве точки отложенной проверки для функции potential_double_close( ) и в качестве целевого узла указывается не узел применения резюме, а его предшественник. Это выполняется для упрощения, поскольку узел применения резюме на момент сохранения информации еще не создан (сохранение информации проверяющих модулей является одним из действий по его созданию, и к этому моменту проведены не все действия по применению резюме), но информация об узле предвызова и целевом узле вызываемой функции позволяет однозначно идентифицировать ветвь резюме (или группу ветвей), затрагиваемую при прохождении пути до заданной

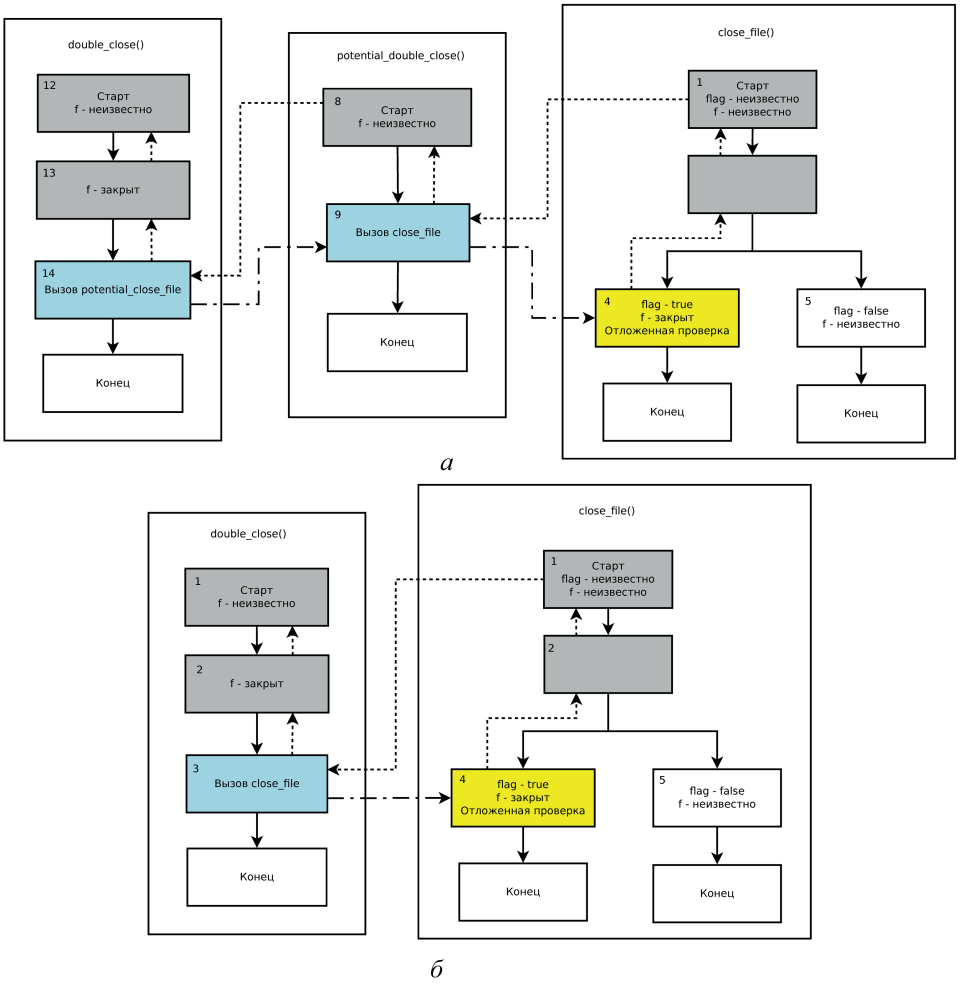

Рис. 2. Схемы построения отчета при использовании резюме вызова функции многократной вложенности (*а*) и полного пути выполнения внутри вызываемой функции (*б*)

точки. Схема построения пути в рассматриваемом случае показана на рис. 2, *а*.

При показе пути внутри вложенного графа (рис. 2, *б*) возникает проблема именования, поскольку внутри графа выполнения функции схема именования является локальной и не связана со схемой именования вызывающей функции. Указанную проблему можно решить описанным выше способом с помощью переименования именованных символьных значений из контекста вызываемой функции в контекст вызывающей. При этом можно выполнять переименование не для всех символьных значений, имеющихся внутри пути выполнения, а только для тех, информация о которых непосредственно представляется пользователю. Это уменьшает временны́е затраты, так как актуализация символьного значения может быть достаточно дорогой операцией.

Заключение. 1. Разработан и подробно описан метод межпроцедурного анализа с помощью резюме для модели анализатора, используемой в статическом анализаторе CSA. В результате получена новая модификация метода межпроцедурного анализа с помощью резюме для метода символьного выполнения, применимая для анализа программ в целях выявления различных классов потенциальных дефектов.

2. Полученный метод реализован в виде опциональной возможности статического анализатора CSA. Это позволяет как непосредственно проводить анализ программ, разработанных с использованием языков C и C++, для поиска дефектов с помощью разработанного метода, так и сравнивать производительность при применении разработанной модификации метода резюме с методом встраивания.

3. Для разработанного метода предложен новый метод отображения результатов анализа в целях получения возможности оценки качества анализатора, его отладки и доработки, а также в качестве потенциального решения для отображения результатов анализа конечному пользователю, чтобы сократить время, затрачиваемое на поиск и исправление найденных дефектов. Такой метод также реализован для анализатора CSA.

4. Проведено предварительное сравнение двух методов межпроцедурного анализа: метода встраивания, реализованного непосредственно в анализаторе; метода резюме, реализованного и описанного в настоящей работе. Результаты предварительных измерений показывают увеличение общей скорости анализа в однофайловом режиме приблизительно на 20 %: с 1,5 до 1,2 ч для анализа исходных файлов на языках C и C++ в составе операционной системы Android с использованием двухпроцессорного сервера Intel Xeon E5-2650 (2,6 ГГц, всего 16 физических и 32 виртуальных ядра).

В дальнейшем планируется провести подробное сравнение производительности как в однофайловом, так и в межмодульном режиме при различных опциях анализатора в целях получения достоверных результатов. Кроме того, поскольку разработанный метод показал хорошую применимость для различных задач, планируется расширить список проверяющих модулей, поддерживающих использование резюме.

ЛИТЕРАТУРА

1. *James C. King.* Symbolic execution and program testing // Communications of the ACM, 1976. Vol. 19. No. 7. P. 385–394.
2. *Ахо А.*, *Лам М.*, *Сети Р.*, *Ульман Д.* Компиляторы: принципы, технологии и инструментарий. М.: Вильямс, 2008.
3. *Cadar C.*, *Dunbar D.*, *Engler D.* KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs // In Proceedings of the USENIX Symposium on Operating System Design and Implementation. 2008.

4. *Cha Sang Kil*, *Avgerinos Th.*, *Rebert A.*, *Brumley D*. Unleashing Mayhem on Binary Code // In Proceedings of the 33rd IEEE Symposium on Security and Privacy. 2012.
5. *Using* Symbolic Evaluation to Understand Behavior in Configurable Software Systems / E. Reisner, Ch. Song, Kin-Keung Ma, Jeffrey S. Foster, A. Porter // In Proceedings of the 32nd International Conference on Software Engineering (ICSE). Cape Town, South Africa. 2010. P. 445–454.
6. *Xu Zh.*, *Zhang J.*, *Xu Zh.* Melton: a practical and precise memory leak detection tool for C programs // Frontiers of Computer Science in China. 2015. Vol. 9. No. 1. P. 34–54.
7. *Qadeer S.*, *Rajamani S.*, *Rehof J*. Summarizing procedures in concurrent programs // 31st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. 2004. Vol. 39. P. 245–255.
8. *Braberman V.*, *Garbervetsky D.*, *Hym S.*, *Yovine S.* Summary-based inference of quantitative bounds of live heap objects // Science of Computer Programming. 2013. Vol. 92. P. 56–84.
9. *Clang* Static Analyzer. URL: http://clang-analyzer.llvm.org (дата обращения: 15.04.2015).
10. *Clang*: a C language family frontend for LLVM. URL: http://clang.llvm.org (дата обращения: 15.04.2015).
11. *The LLVM* compiler infrastructure. URL: http://llvm.org (дата обращения: 14.04.2015).
12. *Игнатьев В.Н.* Использование легковесного статического анализа для проверки настраиваемых семантических ограничений языка программирования // Труды Института системного программирования РАН. 2012. Т. 22. С. 169–188.
13. *Масштабируемый* инструмент поиска клонов кода на основе семантического анализа программ / С. Саргсян, Ш. Курмангалеев, А. Белеванцев, А. Асланян, А. Балоян // Труды Института системного программирования РАН. 2015. Т. 27. С. 39–50.
14. *Reps T.*, *Horwitz S.*, *Sagiv M*. Precise interprocedural dataflow analysis via graph reachability // In POPL ’95 Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages. 1995. P. 49–61.
15. *Xu Z.*, *Kremenek T.*, *Zhang J*. A memory model for static analysis of C programs // In ISoLA’10 Proceedings of the 4th international conference on Leveraging applications of formal methods, verification, and validation. 2010. P. 535–548.

REFERENCES

[1] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 1976, vol. 19, no. 7, pp. 385–394.
[2] Aho A., Lam M., Sethi R., Ullman D. Compilers: Principles, Techniques, and Tools. Addison-Wesley, 1986.
[3] Cadar C., Dunbar D., Engler D. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *In Proc. of the USENIX Symposium on Operating System Design and Implementation*, 2008.
[4] Cha Sang Kil, Avgerinos Th., Rebert A., Brumley D. Unleashing Mayhem on Binary Code. *In Proc. of the 33rd IEEE Symposium on Security and Privacy*, 2012.
[5] Reisner E., Song Ch., Ma Kin-Keung, Foster Jeffrey S., Porter A. Using Symbolic Evaluation to Understand Behavior in Configurable Software Systems. *In Proc. of the 32nd International Conference on Software Engineering (ICSE)*, Cape Town, South Africa, 2010, pp. 445–454.
[6] Xu Zh., Zhang J., Xu Zh. Melton: a practical and precise memory leak detection tool for C programs. *Frontiers of Computer Science in China*, 2015, vol. 9, no. 1, pp. 34–54.

[7] Qadeer S., Rajamani S., Rehof J. Summarizing procedures in concurrent programs. *31st Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, 2004, vol. 39, pp. 245–255.

[8] Braberman V., Garbervetsky D., Hym S., Yovine S. Summary-based inference of quantitative bounds of live heap objects. *Science of Computer Programming*, 2013, vol. 92, pp. 56–84.

[9] Clang Static Analyzer. URL: http://clang-analyzer.llvm.org (accessed: 15.04.2015).

[10] Clang: a C Language Family Frontend for LLVM. URL: http://clang.llvm.org (accessed: 15.04.2015).

[11] The LLVM Compiler Infrastructure. URL: http://llvm.org (accessed: 14.04.2015).

[12] Ignat'ev V.N. Usage of the lightweight static analysis for checking adaptive semantic constraints of the software programming language. *Tr. Instituta sistemnogo programmirovaniya RAN* [Proc. of The Institute for Systems Programming, Russian Academy of Sciences], 2012, vol. 22, pp. 169–188 (in Russ.).

[13] Sargsyan S., Kurmangaleev Sh. Belevantsev, A., Aslanyan A., Baloyan A. Zoomed searching tool of code clones based on program semantics analysis. *Tr. Instituta sistemnogo programmirovaniya RAN* [Proc. of The Institute for Systems Programming, Russian Academy of Sciences], 2015, vol. 27, pp. 39–50 (in Russ.).

[14] Reps T., Horwitz S., Sagiv M. Precise interprocedural dataflow analysis via graph reachability. *In POPL '95 Proc. of the 22nd ACM SIGPLAN-SIGACT symp. on Principles of programming languages*, 1995, pp. 49–61.

[15] Xu Z., Kremenek T., Zhang J. A memory model for static analysis of C programs. *In ISoLA'10 Proc. of the 4th international conf. on Leveraging applications of formal methods, verification, and validation*, 2010, pp. 535–548.

Статья поступила в редакцию 25.05.2015

Романова Татьяна Николаевна — канд. физ.-мат. наук, доцент кафедры "Программное обеспечение ЭВМ и информационные технологии" МГТУ им. Н.Э. Баумана.
МГТУ им. Н.Э. Баумана, Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5.

Romanova T.N. — Ph.D. (Phys.-Math.), Associate Professor, Department of Software for Computers and Automation Systems, Bauman Moscow State Technical University.
Bauman Moscow State Technical University, 2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation.

Сидорин Алексей Васильевич — аспирант кафедры "Программное обеспечение ЭВМ и информационные технологии" МГТУ им. Н.Э. Баумана, старший инженер-программист Московского исследовательского центра Samsung.
МГТУ им. Н.Э. Баумана, Российская Федерация, 105005, Москва, 2-я Бауманская ул., д. 5.
Московский исследовательский центр Samsung, Российская Федерация, 125047, Москва, 1-я Брестская ул., д. 29/22.

Sidorin A.V. — Ph.D. student, Department of Software for Computers and Automation Systems, Bauman Moscow State Technical University, Senior programmer engineer, Samsung R&D Institute Rus (SRR).
Bauman Moscow State Technical University, 2-ya Baumanskaya ul. 5, Moscow, 105005 Russian Federation.
Samsung R&D Institute Rus, 1-ya Brestskaya ul. 29/22, Moscow, 125047 Russian Federation.

Просьба ссылаться на эту статью следующим образом:

Романова Т.Н., Сидорин А.В. Метод резюме для разработки универсального многоцелевого анализатора кодов программ с возможностью обнаружения различных классов дефектов в программах, созданных с использованием языков С и С++ // Вестник МГТУ им. Н.Э. Баумана. Сер. Приборостроение. 2015. № 5. С. 75–96.

Please cite this article in English as:

Romanova T.N., Sidorin A.V. Summary-based interprocedural analysis method for implementation in multi-purpose static C/C++ code analyzer. *Vestn. Mosk. Gos. Tekh. Univ. im. N.E. Baumana, Priborostr.* [Herald of the Bauman Moscow State Tech. Univ., Instrum. Eng.], 2015, no. 5, pp. 75–96.

В Издательстве МГТУ им. Н.Э. Баумана вышла в свет книга

МОДЕЛИ И МЕТОДЫ ОЦЕНКИ ОСТАТОЧНОГО РЕСУРСА ИЗДЕЛИЙ РАДИОЭЛЕКТРОНИКИ

Определены количественные показатели остаточного ресурса и установлены их точечные, доверительные и гарантированные оценки. Исследована достижимость гарантированных непараметрических оценок, найдены их коэффициенты смещения. Получены асимптотические и предельные оценки показателей остаточного ресурса, доказаны их экстраполяционные и интерполяционные оценки. С использованием физических моделей расходования ресурса изделий радиоэлектроники получены формулы для расчета и оценки показателей ресурса в штатном режиме эксплуатации через показатели для форсированного режима. Приведены примеры схемной избыточности радиоэлектронной аппаратуры в качестве источника увеличения ресурса.

Для научных работников. Может быть полезна преподавателям, аспирантам и студентам старших курсов технических вузов, а также специалистам в области надежности радиоэлектронной аппаратуры.