Обзор подходов к улучшению качества результатов статического анализа программ[1]

А.Ю. Герасимов <agerasimov@ispras.ru>
Институт системного программирования РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25

Аннотация. В настоящий момент индустрия создания программ для всевозможного рода вычислительных устройств находится в состоянии бурного развития. Постоянно увеличивающаяся мощность вычислительных систем предоставляет всё новые возможности для создания высокопроизводительных, в том числе – параллельных, программ и программных комплексов. В связи с этим постоянно возрастает сложность программного обеспечения, управляющего вычислительными системами. Из-за высокой сложности программных систем процесс обеспечения качества разрабатываемого программного обеспечения требует новых подходов к процессу проверки корректности программ как на соответствие требованиям пользователей, так и на наличие критических дефектов и уязвимостей безопасности. Одним из методов контроля качества программного обеспечения является применение инструментальных средств программиста, предназначенных для анализа программ. Отрасль создания инструментальных средств статического и динамического анализа программ активно развивается с начала 2000-х годов. Разрабатывается большое количество академических и промышленных сред и инструментов анализа программ. В связи с фундаментальными ограничениями и инженерными компромиссами в угоду производительности и масштабируемости инструменты статического анализа не всегда могут обеспечить отсутствие ошибок первого рода в результатах своей работы. При этом анализ предупреждений инструмента может отнимать значительное время высококвалифицированного эксперта в области разработки и обеспечения качества программного обеспечения. В связи с этим возникает задача улучшения качества результатов работы статических анализаторов программ. Данная статья посвящена обзору методов анализа программ и подходов к улучшению качества работы статических анализаторов. Особое внимание в статье уделяется методам совмещения подходов статического и динамического анализа программ.

Ключевые слова: статический анализ программ; динамический анализ программ; комбинированный анализ программ

[1] Работа проводится при финансовой поддержке Российского фонда фундаментальных исследований. Проект 07-17-00702.

DOI: 10.15514/ISPRAS-2017-29(3)-6

Для цитирования: Герасимов А.Ю. Обзор подходов к улучшению качества результатов статического анализа программ. Труды ИСП РАН, том 29, вып. 3, 2017 г., стр. 75-98. DOI: 10.15514/ISPRAS-2017-29(3)-6

1. Введение

Область исследований, посвященная методам автоматического анализа программ с целью обнаружения дефектов, активно развивается в последнее время. Появляется большое количество инструментов статического и динамического анализа, а также инструментов, совмещающих оба этих подхода в том или ином виде. В данной работе рассматривается понятие программной ошибки, даётся обзор проблем методов статического анализа программ и приводится обзор подходов к улучшению результатов работы инструментов статического и динамического анализа программ.

2. Программные ошибки

В 1842 году Чарльз Бэббидж (*Charles Babbadge*), английский математик и создатель Аналитической машины (Большой разностной машины), был приглашен в Туринский университет провести семинар о созданной им машине. Луиджи Менабреа (*Luigi Federico Menabrea*), итальянский инженер, записал этот семинар на французском языке, и этот текст в последствии был опубликован в Общественной библиотеке Женевы в октябре 1942 года. Друг Бэббиджа, изобретатель Чарльз Уитстон (*Sir Charles Wheatstone*), попросил графиню Аду Августу Лавлейс (*Ada Augusta, countess of Lovelace*) перевести эти записи Менабреа на английский и сопроводить текст комментариями. С учётом 52 страниц комментариев графини Ады Августы Лавлейс труд под названием «Очерк об аналитической машине, представленной Чарльзом Бэббиджем» [1], опубликованный под акронимом ААЛ оказался более обширным, чем записи Менабреа. В этом очерке Ада Августа графиня Лавлейс в частности заметила: «…анализирующий процесс должен быть одинаково произведен в соответствии с предоставленными Аналитической Машине необходимыми управляющими данными, и это при сём также может быть источником возможной ошибки. Правда в том, что механизм безошибочен в своих процессах, но карты (*прим.* с управляющими данными) могут давать ошибочные команды». Это означает, что уже на заре эры программируемых вычислительных устройств, которые используют внешние данные как управляющие команды, осознавалась возможность наличия ошибок в этих управляющих данных, которые мы в настоящее время называем программами.

В статье «Что мы знаем о методах обнаружения ошибок?» [2] предлагается трактовка программной ошибки (дефекта), как некоторой сущности, которая приводит к одной или нескольким ошибкам в таком артефакте как программный код. В то же время проводится анализ двенадцати исследований

на тему сравнения инспекций кода и тестирования, в которых выделяются дефекты на уровне требований к программному обеспечению, на уровне спецификаций программного обеспечения и в коде программного обеспечения.

Общепринятого определения программного дефекта на данный момент обнаружить не удалось. В различных работах, посвященных обнаружению дефектов в программах, даётся своё определение с целью использования его в исключительно рамках работы. В 2010 году подразделением Компьютерного сообщества (IEEE Computer Society) Института инженеров электротехники и электроники (IEEE – Institute of Electrical and Electronics Engineers) выпущен стандарт IEEE 1044-1999 «Стандартная классификация для программных аномалий» (IEEE Standard Classification for Software Anomalies) [3], в котором даётся несколько определений для терминов, используемых в рамках данной классификации:

- *дефект (defect)* – несовершенство или недостаток в работающем продукте, когда этот работающий продукт не соответствует требованиям или спецификациям и требуется либо его исправление, либо замена;
- *оплошность (error)* – действие человека, которое приводит к некорректному результату;
- *повреждение (failure)* – прекращение возможности продукта выполнять требуемую функцию или неспособность выполнять функцию в ранее указанных ограничениях;
- *ошибка (fault)* – сообщение об ошибке в программе;
- *проблема (problem)* – (A) трудность или неясность, испытанная одним или несколькими пользователями, которая случилась в процессе использования системы. (B) Негативная ситуация, которую необходимо преодолеть.

Стоит отметить, что признать данный набор определений полным не представляется возможным, в связи с тем, что существуют правила, нарушение которых может трактоваться как программный дефект, но его описание не подходит ни под одно из вышеперечисленных определений.

Например, в работе [4] есть «Правило 4», которое гласит, что длина кода функции не должна превышать один стандартный лист, где каждая декларация и каждое выражение размещено на одной строке. Обычно это означает, что текст функции должен быть не более 60 строк. Считается, что трудно оперировать фрагментами кода более этого размера. Однако, нарушение этого правила вряд ли однозначно приведет к наличию ошибки в программе. А стандарт Ассоциации надёжности программного обеспечения в автомобильной индустрии (Motor Industry Software Reliability Association) содержит правило MISRA 2004 [5] 5.1: «Идентификаторы (внутренние и внешние) не должны полагаться на значимость более чем 31 символа».

Нарушение этого правила вряд ли приведет к нарушению работоспособности программного обеспечения, если используемый разработчиками компилятор языка Си поддерживает идентификаторы большего размера. Но программа, предназначенная для работы на борту автомобиля и содержащая идентификаторы с длиной более 31 символа, не пройдёт сертификацию по стандарту MISRA, что может оказаться достаточным условием для отказа прохождения программой и автомобилем в целом внутреннего или внешнего аудита безопасности. В связи с этим, для производителя автомобиля, которому важно соответствие программы правилам MISRA, нарушение правила будет считаться критической ошибкой, пропуск которой может привести к серьезным финансовым потерям. Но, если разработчиками используется компилятор, который учитывает только первые 31 символ идентификатора, и не выводит предупреждение о том, что используется идентификатор большей длины, то это может привести к серьезному нарушению логики работы программы.

С другой стороны, наибольший интерес представляют дефекты в программном обеспечении, которые приводят к некорректной работе или сбою в работе программы.

3. История развития методов статического анализа программ

На заре компьютерной эры достаточно большое внимание уделялось тестированию, как методу обеспечения качества программных систем. В 60-х годах XX века тесты пытались проводить таким образом, чтобы покрыть все возможные участки программы на всех возможных входных данных и показать корректность работы программы (так называемое исчерпывающее тестирование). Но количество возможных путей исполнения сколь-нибудь сложной программы и количество значений входных данных оказывалось настолько велико, что подобный подход к обеспечению качества программного обеспечения был признан непрактичным [6]. В лекции «О надёжности программ» Эдскер Вибе Дейкстра утверждает, что тестирование может использоваться очень эффективно для того, чтобы показать наличие ошибки, но не может показать их отсутствие [7].

Как отмечается в статье «Что мы знаем о методах обнаружения ошибок?» [2], в процессе инспекции кода программ обнаруживается в среднем от 25% до 50% дефектов в программе, а в процессе тестирования от 30% до 60%. То есть в среднем в проверенной опытными разработчиками программного обеспечения и хорошо протестированной программе остаётся около половины дефектов, не обнаруженных на этапах, предназначенных для обеспечения качества программы. При этом экспертом, проводящим инспекцию кода, в среднем обнаруживается от 1 до 2,5 дефектов в час.

В статье, посвященной истории создания языка Си [8], Деннис Ритчи (Dennis M. Ritchie) отмечает, что несмотря на то, что в первой редакции книги «Язык

программирования Си» было указано большинство правил, которые привели систему типов языка Си к его современной форме, многие программы были написаны в старом, более свободном стиле, и компиляторы это позволяли. Для того, чтобы обратить больше внимания разработчиков программ на официальные правила языка, обнаруживать разрешенные, но подозрительные конструкции, и помочь найти несоответствие интерфейсов, не обнаруживаемые простыми механизмами в процессе раздельной компиляции, Стив Джонсон (Steve Johnson) адаптировал свой компилятор *pcc* [9] для создания инструмента *lint* [10], который сканировал файлы исходного кода программы и отмечал сомнительные конструкции. Особенностью программы *lint* являлась возможность сравнивать соответствие и находить отсутствие противоречий во всей программе, путём сканирования множества исходных файлов и сравнения типов аргументов функций в месте вызова с типами аргументов функции, указанных в определении.

Согласно утверждениям технического отчёта «Следующее поколение статического анализа» [11] программу *lint* можно назвать первым инструментом статического анализа кода, но на самом деле *lint* не разрабатывался с целью обнаружения дефектов, которые приводят к проблемам времени исполнения. Скорее, целью создания инструмента было обнаружение подозрительных и непортируемых конструкций в коде с целью помощи разработчикам создавать более согласованный код. Под «подозрительными конструкциями» подразумевается технически корректный, с точки зрения спецификации языка, исходный код (например, Си, Си++), который может привести к такому выполнению программы, которое не подразумевалось разработчиком. Проблема заключалась в том, что такой подозрительный код мог исполняться и часто исполнялся корректно. Поэтому из-за ограниченных возможностей инструмента *lint* уровень шума (ложноположительных предупреждений об ошибках) был экстремально высок, часто превышая соотношение шума и реальных дефектов 10:1.

В начале 2000-х стали появляться инструменты статического анализа программ второго поколения [11], такие как: Coverity Prevent, Klocwork и другие [12]. Основной особенностью данных инструментов было смещение фокуса с «подозрительных конструкций» на «дефекты времени исполнения», что потребовало комбинации изощрённого анализа путей исполнения с межпроцедурным анализом, чтобы понять, что происходит, когда управление передаётся от одной функции к другой внутри программной системы [13]. В связи с постоянной борьбой за точность и масштабируемость анализа инструменты второго поколения могли находить ограниченный набор дефектов достаточно точно, но либо анализ не масштабировался на миллионы строк исходного кода, либо мог работать достаточно быстро, но обладал малой точностью, как инструмент *lint*, привнося уже известные проблемы шума предупреждений [13, 14]. Борьба за иллюзорный баланс между точностью, масштабируемостью и производительностью анализа привела к проблеме ложноположительных предупреждений о дефектах в инструментах

второго поколения. Если для инструментов первого поколения проблема шума в результатах работы препятствовала их внедрению в индустрии, то инструменты второго поколения сдвинули развитие инструментов статического анализа программ с технологической мёртвой точки, позволяя обнаруживать осмысленные дефекты в программах, но разработчики программного обеспечения всё чаще ожидают лучшей точности результатов анализа.

В связи с развитием алгоритмов и появлением инструментов для решения формул в булёвых ограничениях (решателей) [15], в последнее время стали появляться инструменты статического анализа третьего поколения, которые объединяют классические методы анализа путей исполнения программы в комбинации с применением решателей. Согласно исследованию Чельфа и Чу [11] применение подобной комбинации позволило получить дальнейшие технологические преимущества для статического анализа и снизить высокие уровни предупреждений ошибок первого рода (ложноположительных предупреждений) на 15%.

4. Методы улучшения результатов работы инструментов статического анализа программ

Несмотря на постоянное улучшение качества результатов работы инструментов статического анализа программ, например, коммерческого инструмента Coverity Prevent [16], с постоянным уменьшением соотношения ложноположительных предупреждений к общему количеству предупреждений до 9,7% в среднем по всем типам дефектов, для отдельных проектов этот показатель оказывается на достаточно высоком уровне (менее 20% [17]). Это означает, что в среднем каждое пятое предупреждение о дефекте является ложным.

В связи с ограничениями инструментов статического анализа, связанными с компромиссом между обеспечением точности, производительности и масштабируемости анализа, а также принимая во внимание фундаментальную теорему об алгоритмической неразрешимости задачи об определении останова работы алгоритма, сформулированную Аланом Тьюрингом ещё в 30-х годах XX века, возникла задача повышения точности результатов статического анализа путем постобработки результатов работы инструментов анализа и комбинирования статических и динамических методов анализа.

В октябре 2016 года на конференции, посвященной анализу и манипуляциям над исходным кодом программ, была представлена обширная классификация методов постобработки результатов статического анализа [18]. Авторы классификации, применяя методы автоматизированной обработки научных статей и докладов на научных конференциях, собрали и классифицировали 79 статей по предлагаемым методам постобработки результатов работы инструментов статического анализа. Авторы обзора выделяют семь основных категорий методов, из которых один метод – упрощение инспекций

результатов работы инструментов статического анализа на основе обратной связи от пользователя, – является автоматизированным. Ниже кратко будут рассмотрены автоматические методы улучшения результатов работы инструментов статического анализа с целью повышения их качества, приведенные в этой статье. Особое внимание будет уделено методам совмещения статического и динамического анализа. Этот раздел будет расширен дополнительными примерами подходов и инструментов.

4.1 Кластеризация, основанная на схожести или связях

Задача методов кластеризации – объединить в одно множество (кластер) группу дефектов, с дальнейшей целью оценки одного предупреждения о дефекте из группы как ложноположительное или истинно-положительное и применить эту оценку для всей группы сразу. Кластеризация разделяется авторами на прямую и косвенную. Различие заключается в том, что для прямой кластеризации гарантируется зависимость или связи между похожими или связанными предупреждениями, сгруппированными вместе. Например, в работе Ли и др. [19] кластеризация предупреждений о дефектах производится на основе арифметической связи между индексными переменными при доступе к буферу. При косвенной кластеризации похожие предупреждения группируются по синтаксической или структурной схожести кода или характеристик предупреждения, вычисленных самим инструментом статического анализа кода или независимо от него. Так в работе Фрая и других [20] предлагается подход к вычислению метрики близости на основе различных характеристик предупреждения о дефекте, таких как: название функции, контекст строки, в которой обнаружен потенциальный дефект, и другие.

4.2 Ранжирование

Целью ранжирования является выстраивание предупреждений в выводе статического анализатора таким образом, чтобы те предупреждения, которые имеют большую вероятность быть истинно-положительными, оказывались наверху списка предупреждений, а с меньшей – внизу списка.

Статистическое ранжирование – наиболее распространенная техника приоритезации предупреждений. Например, в работе Кременека и Энглера [21] применена простая статистическая модель ранжирования дефектов. Она основана на наблюдении, что в коде, содержащем много успешных проверок и малое количество предупреждений, как правило содержится ошибка. С точки зрения работы статического анализатора, проводящего анализ потока данных в программе, успешная проверка останавливает распространение факта об инициализации ошибочной ситуации или, при обратном проходе, – факта о потенциально опасной операции, на пути исполнения программы.

Ранжирование по истории изменений. В этой категории дефекты ранжируются с использованием истории исправлений дефектов. Ким и Эрнст

[22, 23] дают предупреждению более высокий приоритет, если оно относится к категории дефектов, которые быстро исправляются программистами, то есть считаются более важными.

Самоадаптирующееся ранжирование, основанное на обратной связи. В этой категории дефекты ранжируются на основе обратной связи от пользователя. Например, Шен и др. [24] впервые предложили назначать каждому шаблону дефекта предопределенную вероятность того, что предупреждение о наличии дефекта истинно и ранжировать дефекты в зависимости от этой вероятности. А в последствии изначальное ранжирование изменяется на основе реакции пользователя в процессе инспекции предупреждений о дефектах.

Из других методов стоит выделить ранжирование предупреждений на основе статического вычисления вероятности исполнения мест, для которых предупреждения даны [25], и ранжирование предупреждений на основе слияния результатов нескольких инструментов статического анализа [26, 27].

4.3 Отсечение или классификация предупреждений

Подходы, отнесенные к этой категории, предлагают классифицировать дефекты при помощи двоичной классификации как имеющие практическое значение (actionable) или нет. При этом стоит отметить, что данный подход в результате может привести к сокрытию предупреждений о реальных дефектах в программе.

Классификация на основе машинного обучения. В работе Ханама и др. [28] классификация достигается за счёт нахождения похожих шаблонов в коде программы, окружающем место, для которого получено предупреждение о возможной ошибке. Юскел и Созер [29] оценили 34 алгоритма машинного обучения в их исследовании, используя 10 различных характеристик предупреждения.

Идентификация дельты предупреждений. Метод заключается в идентификации различия предупреждений в результате работы инструмента статического анализа через сравнение результатов анализа кода на предыдущей и текущей версии программного обеспечения. Например, Спакко и др. [30] идентифицировали новые предупреждения в сравнении с предыдущей версией, сравнивая предупреждения двумя способами: *составление пар* и *подписей предупреждений*. Чимдялвар и Кумар [31] отсекали вновь появляющиеся ложноположительные предупреждения о дефектах в развивающихся программных системах путем анализа влияния внесенных изменений в текущей системе и подавляя те предупреждения, которые не зависят от внесенных изменений.

Также стоит отметить работу Муске и др. [32], в которой авторы применяют *логистическую регрессию* для определения вероятности каждого предупреждения быть ложным или нет с точки зрения необходимости исправления ошибки в коде. Авторы выделили несколько групп факторов, влияющих на вероятность отнесения предупреждения к истинно-

положительному или ложноположительному, а также на вероятность того, что обнаруженная ситуация в коде должна быть исправлена.

4.4 Избавление от ложноположительных предупреждений

В рамках этого подхода используются более точные техники, такие как проверка моделей и символьное исполнение, для идентификации и устранения ложноположительных предупреждений инструментов статического анализа.

Достижение масштабируемости. В своей работе Пост и др. [33] предлагают подход к инкрементальному расширению контекста для верификации начиная с минимального контекста одной функции с постепенным расширением его на функции, вызывающие данную функцию и далее, по мере необходимости.

Улучшение производительности и эффективности. В другой работе [34] Муске и другие отмечают низкую производительность методов проверки моделей. В связи с этим авторы предлагают техники предсказания результата проверки модели, в связи с чем добиваются лучшей производительности избавления от ложноположительных предупреждений за счёт уменьшения количества вызовов для проверки моделей.

Стоит отметить, что существует ряд работ [35], [36], [37] о комбинации статического анализа и проверки моделей, где описываются подходы к тесной интеграции этих двух методов, в процессе которой они итеративно обмениваются информацией.

4.5 Создание легковесных инструментов статического анализа

В эту категорию попали инструменты легковесного статического анализа, которые способны с увеличенной производительностью анализировать очень большие программные системы. Тем не менее данные инструменты не гарантируют нахождение всех дефектов определенного типа.

Ховемейер и Пуг [38] разработали автоматические обнаружители различных шаблонов ошибок в Java-программах. Получившийся инструмент FindBugs получил одобрение как академическим сообществом, так и индустрией, несмотря на то, что проводит неглубокий внутрипроцедурный анализ.

Splint [39] – пример другого инструмента, использующего легковесный статический анализ для обнаружения возможных уязвимостей. Анализ, проводимый инструментом Splint, подобен анализу, проводимому компилятором, он эффективен и масштабируем и при этом обнаруживает широкий диапазон реализационных изъянов.

4.6 Комбинация статического и динамического анализа программ

В основном в эту группу попали инструменты, объединяющие подходы статического и динамического анализа программ.

В работе [40] описывается инструмент Check'N'Crash, который объединяет инструмент статического анализа ESC/Java [41] и JCrasher [42] – генератор тестов, путем извлечения конкретных значений свойств, специфических для абстрактных условий ошибок, найденных статическим анализатором, при помощи решения системы ограничений, с последующей генерацией тестов для воспроизведения дефекта.

Как развитие данного подхода в работе [43] авторы описывают инструмент DSD-Crasher, в котором используется трехступенчатый подход, состоящий из: исполнения программы с целью динамического обнаружения заложенного поведения программы для ограничения множества входных значений программы; статического анализа с целью обнаружения потенциальных ошибок; и динамической верификации обнаруженных потенциальных ошибок для проверки их достижимости.

Слайсинг программ также может использоваться для эффективного объединения статического и динамического анализа. В работе [44] описывается инструмент SANTE, который объединяет подход статического анализа для обнаружения дефектов в программах и генерацию тестов для подтверждения найденных дефектов. В связи с тем, что генерация тестов для больших программ может занимать значительное время, авторы инструмента предлагают применять слайсинг программ с целью уменьшения исходного кода, а также исключения незначительных деталей программы с точки зрения генерации теста, до начала самого процесса генерации тестов.

Ли и другие в работе [45] предлагают подход «остаточного исследования», в котором используется динамический анализ, как сервис для статического анализа с целью подтверждения во время исполнения является ли ситуация, найденная статическим анализатором истинной ошибкой, или нет.

В дополнение к инструментам и подходам, рассмотренным в работе [19], стоит отметить обзор [46], в котором авторы сконцентрировались на инструментах и подходах, так или иначе использующих объединение техник контроля качества программного обеспечения, объединяющих статический анализ программ и динамический анализ программ.

В работе [47] предлагается подход к объединению статического анализа с целью обнаружения подозрительных мест в программе и генерации тестов для подтверждения обнаруженных в процессе статического анализа возможных уязвимостей при помощи динамического анализа. Подход основан на статическом анализаторе SVR [48], построенном на базе промышленного компилятора GCC, который начиная с версии 4, включает в себя среду Tree-SSA [49], созданную для облегчения построения статических анализаторов на базе универсального промежуточного представления GIMPLE. Статический анализатор по исходному коду программы строит модель для инструмента Moped [50], на основе которой проверяются свойства безопасности программы. В качестве результата статического анализатора получается набор путей в программе, которые приводят к потенциальному дефекту. Далее для

этих путей строятся входные данные и проверяется достижимость определенных состояний в программе в процессе её запуска.

В работе [51] рассматривается подход, названный авторами как обобщенный алгоритм анализа программы. В основе подхода лежит идея объединения статического и динамического анализа в рамках одного инструмента JNuke, который абстрагирует алгоритм работы анализатора от метода анализа, и предоставляет возможность анализировать различные свойства программы, вычисляемые статически или обнаруживаемые динамически через единый интерфейс.

В работе [52] описывается идея, лежащая в основе ранее упоминавшегося инструмента SANTE, который развивает идеи, заложенные в инструменте PathCrawler [53], и объединяет подход статического анализа исходного кода программ на языке Си Frama-C [54, 55] и динамическое символьное исполнение программ с целью построения тестовых наборов, на которых воспроизводятся найденные дефекты.

Инструмент Yogi [56], разработанный в научном подразделении Microsoft Research для проекта тестирования драйверов операционной системы Windows, представляет комбинацию статического анализатора свойств безопасности программы, описанных на языке Slic [57] и интерпретатора промежуточного представления программы для Yogi, который способен проводить как символьное выполнение Yogi-представления программы, так и симуляцию выполнения программы на базе Yogi-представления с целью подтверждения достижимости ошибочной ситуации описанной на Slic.

В работе [58] авторами предлагается подход к объединению результатов работы статического и динамического анализа программ на языке Си с целью обнаружения уязвимости выхода за границы буфера в памяти. В процессе статического анализа вычисляются последовательности помеченных зависимостей между входными данными и уязвимыми операциями, которые в последствии используются в процессе вычисления входных данных при помощи генетического алгоритма, использующего фитнес-функцию, основанную на концепции частотно-спектрального анализа помеченных зависимостей программы, представленной в работе [59].

В работе [60] описывается инструмент ConDroid, предназначенный для анализа программ операционной системы Android. Основной особенностью этого инструмента является использование статического анализа кода программы с целью систематического обнаружения критических, с точки зрения безопасности, регионов программы, путей достижения этих регионов и итеративного динамического анализа. Итеративный динамический анализ в этом инструменте реализует идею кокретно-символьного (*англ.* concolic от concrete and symbolic) исполнения, в процессе которого программа исполняется на конкретных входных данных с целью сбора трассы исполнившихся операций и вычисления новых входных данных на основе символьного решения ограничения пути, собранного в процессе исполнения

программы. Итеративный динамический анализ на каждой итерации вычисляет выходные данные для программы таким образом, чтобы в конечном итоге провести исполнение программы по пути, вычисленному на этапе статического анализа.

Похожий подход используется в инструменте DyTa [61], предназначенном для анализа программ, написанных на языке C#. Инструмент использует статический анализ для определения мест потенциальных дефектов с возможными предусловиями. Далее проводится инструментация кода программы с целью добавления операций проверки утверждений о предусловиях в определенных точках программы. После этого программа запускается на исполнение под управлением инструмента Pex [62], проводящего генерацию тестовых наборов методом прозрачного ящика, и в процессе работы проверяет утверждения, вставленные на этапе инструментации.

Инструмент IntelliDroid [63] использует комбинацию статического и динамического анализа с целью обнаружения уязвимостей безопасности в программах операционной системы Android. Статический анализатор проверяет код программы в формате байткода DEX [64] с целью обнаружения использования вызовов к функциям стороннего API. Динамический анализатор использует информацию о месте использования стороннего API вместе с путями достижения точек вызова и генерирует входные данные программы при помощи решателя формул в булёвых ограничениях Z3 [65] с целью обнаружения использования стороннего API в процессе исполнения программы.

Инструмент STAR [66] ещё один инструмент, созданный для генерации тестовых сценариев, воспроизводящих критические ошибки в Java-программах. Используя информацию об необработанной исключительной ситуации в Java-программе, инструмент пытается по сообщению об ошибке получить информацию о трассировке стека, далее пытается восстановить предусловия для ошибки при помощи символьного исполнения и затем вычислить ограничения пути с целью генерации входных данных для тестового сценария воспроизведения ошибки.

В работе, посвященной инструменту AEG [67] описывается подход генерации входных данных для эксплуатации уязвимости переполнения буфера или некорректного использования форматной строки. Подход построен на совмещении техники статическом анализа исходного кода, статического анализа бинарного кода, представленного в формате LLVM и динамического анализа исполняемого кода с целью получения конкретной информации об адресах функций и буферов для построения эксплойта, эксплуатирующего уязвимость, найденную на стадии статического анализа.

В инструменте [68] предлагается подход для классификации дефектов утечки памяти при помощи совмещения статического анализа исходного программы инструментом Fortify [69] с дальнейшим конкретно-символьным (*англ.*

concolic) анализом программы на основе модифицированного инструмента CREST [70] для генерации тестовых наборов для программ на языке C.

В работе [71] рассматривается подход для автоматической генерации тестов для дефектов связанных с выходом за границы буфера в памяти. При помощи статического анализа находятся точки потенциальных дефектов, которые в дальнейшем подтверждаются при помощи динамического символьного исполнения, результатом которого являются наборы входных данных, на которых воспроизводится дефект.

Классифицируя данные работы можно вывести несколько основных направлений, используемых для объединения подходов статического и динамического анализа программ:

- статический анализ и генерация тестов [40, 43, 45, 58];
- статический анализ и проверка моделей [47];
- статический анализ и верификация в процессе исполнения [52];
- статический анализ и симуляция исполнения [56];
- статический анализ и символьное исполнение [44, 60, 61, 63, 67, 68, 71].

5. Заключение

В связи с наличием известных фундаментальных и технологических ограничений на методы статического анализа программ проблема уменьшения ложноположительных предупреждений об ошибках статических анализаторов остаётся актуальной. При наличии инструментов статического анализа промышленного качества, таких как Klocwork и Coverity, а также сред динамического анализа программ, таких как Valgrind [72] и QEMU [73] и инструментов Memcheck [74] и S^2E [75], построенных на их основе, на данный момент не существует промышленных или широко используемых в индустрии инструментов, совмещающих статический и динамический анализ программ. Несмотря на обширную классификацию подходов к улучшению качества результатов работы инструментов статического анализа, наиболее перспективным направлением решения данной проблемы представляется разработка и применение методов совмещения статического и динамического анализа, которые позволят избежать с одной стороны неточности, присущей методам статического анализа, а с другой стороны – высокой вычислительной сложности, присущей методам динамического анализа программ.

Список литературы

[1]. Sketch of The Analytical Engine Invented by Charles Babbage by L. F. Menabea from the Bibliotheque Universalle de Geneve, October, 1942, No. 82. With notes upon the Memoir by the translator Ada Augusta, countess of Lovelace. https://www.fourmilab.ch/babbage/sketch.html, , дата обращения 05.05.2017

[2]. Per Runeson, Carian Andersson, Thomas Thelin, Anneliese Andrews, Tomas Berling. What Do We Know about Defect Detection Methods? *IEEE Software May/June 2006*

[3]. IEEE 1044-2009 Standard Classification for Software Anomalies. *IEEE. 3 Park Avenue, New Yourk, NY 10016-5997, USA, 7 January 2010, ISBN 978-0-7381-6114-3*

[4]. Gerald J. Holzmann. The Power of 10: Roles for Developing Safety-Critical Code. *Computer/ 2006, vol. 39, no. 6, pp 95-97*

[5]. MISRA C: 2004 Guidelines for the use of the C language in critical systems. *First published October 2004, by MIRA Limited, Watling Street, Nuneaton, Warkwickshire CV10 0TU UK, ISBN 978-0-9524156-4-0*

[6]. E. J. Weyuker, T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on software engineering, 6(3):236-246.* May 1980.

[7]. E. W. Dijkstra. On the reliability of the programs. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF, дата обращения 05.05.2017

[8]. Dennis M. Ritchie. The development of the C language. *Proceedings of HOPL-II The second ACM SIGPLAN conference on History of programming languages. Cambridge, MA, USA – April 20-23, 1993, pp. 201-208*

[9]. S. C. Johnson. A Portable Compiler: Theory and Practice. *Proceedings of 5th ACM POPL Symposium*, January 1978

[10]. S. C. Johnson. Lint, a Program Checker. *Unix Programmer's manual*, Seventh Edition, Vol. 2B, M.D. McIlroy and B.W. Kernigan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[11]. Benjamine Chelf, Andy Chou. The next generation of Static Analysis. *Coverity*, March 18, 2008. http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf, , дата обращения 05.05.2017

[12]. Pär Emanuelsson, Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools. *Technical report. Department of Computer and Information Science, Linköping University.* Linköping, Sweden, 2008.

[13]. Dawson Engler, Benjamin Chelf, Andy Chou, Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, Volume 4, Article No. 1*. San Diego, California – October 22-25, 2000

[14]. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, Robert Bowdidge. Why don't software developers use static analysis tools to find bugs?. *ICSE'13 Proceedings of the 2013 International conference on Software Engineering.* San Francisco, CA, USE, May 18-26, 2013

[15]. John Franco, John Martin. A history of Satisfiability. *Handbook of Satisfiability.* IOS Press, 2009 doi:10.3233/978-1-58603-929-5-3

[16]. Coverity Scan: 2012 Open Source Report. http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf, дата обращения 05.05.2017

[17]. Coverity Scan. Project Spotlight: Python. http://wpcme.coverity.com/wp-content/uploads/2013-Coverity-Scan-Spotlight-Python.pdf, дата обращения 05.05.2017

[18]. Tukaram Muske, Alexander Serebrenik. Survey of Approaches for Handling Static Analysis Alarms. *Proceedings of IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM).* Raleigh, NC, USA. October 2-3, 2016

[19]. Woosuk Lee, Wonchan Lee, Kwengkeun Yi. Sound non-statistical clustering of static analysis alarms. *VMCAI'12 Proceedings of the 13th international conference on*

verification, model checking and abstract verification interpretation. Philadelphia, PA, USA. January 22-24, 2012.

[20]. Zachary P. Fry, Westley Weimer. Clustering static analysis defect reports to Reduce maintenance costs. *WCRE'13 Proceeding of 30th working conference on reverse engineering.* Koblenz, Germany. October 14-17, 2013.

[21]. Ted Kremenek, Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. *SAS'03 Proceedings of the 10th International conference on static analysis.* San Diego, CA, USA. June 11, 2003.

[22]. Sunghun Kim, Michael D. Ernst. Prioritizing Warning Categories by Analyzing Software History. *MSR'07 Proceedings of the Fourth International Workshop on Mining Software Repositories.* Minneapolis, MN, USA. May 20-26, 2007.

[23]. Sunghun Kim, Michael D. Ernst. Which warnings should I fix first? *ESEC-FSE'07 Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering.* Dubrovnik, Croatia. September 03-07, 2007

[24]. Haihao Shen, Jianhong Fang, Jianjun Zhao. EFindBugs: Effective Error Ranking for FindBugs. *ICST'11 Proceedings of the 2011 Fourth IEEE International conference on Software Testing, Verification and Validation.* Berlin, Germany. March 21-25, 2011

[25]. Sunghun Kim, Michael D. Ernst. Prioritizing Software Inspection Results using Static Profiling. *SCAM'06 Source code analysis and manipulation.* Philadelphia, PA, USA. December 11, 2006.

[26]. Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. *InfoScale'07 Proceedings of the 2nd international conference on Scalable information systems.* Suzhou, China. June 06-08, 2007

[27]. Na Meng, Qianxiang Wang, Qian Wu, Hong Mei. An approach to merge results of multiple static analysis tools. *QSIC'08 Proceedings of the 2008 the eighth international conference on quality software.* Oxford, UK. August 12-13, 2008

[28]. Quinn Hanam, Lin Tan, Reid Holmes, Patrick Lam. Finding patters in static analysis alerts. Improving actionable alert ranking. *MSR 2014 Proceedings of the 11th working conference on mining software repositories.* Hyderabad, India. May 31 – June 01, 2014

[29]. Ulas Yüskel, Hasan Sözer. Automated classification of static code analysis alerts: a case study. *ICSM'13 Proceedings of the 2013 IEEE international conference on software maintenance.* Eindhoven, Netherlands. September 24-26, 2013

[30]. Jaime Spacco, David Hovermeyer, William Pugh. Tracking defect warnings across versions. *MSR'06 Proceedings of the 2006 international workshop on mining software repositories.* Shanghai, China. May 22-23, 2006

[31]. Brahti Chimdyalwar, Shrawan Kumar. Effective false positive filtering for evolving software. *ISEC'11 Proceedings of the 4th India software engineering conference.* Thiruvananthapuram, Kerala, India. February 24-27, 2011

[32]. J. R. Rithruff, J. Penix, J. D. Morgenthaler, S. Elbaum, G. Rothermel. Predicting accurate and actionable static analysis warnings: and experimental approach. *ICSE'08 Proceedings of the 30th international conference on software engineering.* Leipzig, Germany. May 10-18, 2008

[33]. H. Post, C. Sinz, A. Kaiser, T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. *ASE'08 Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering.* L'Aquila, Italy. September 15-19, 2008

[34]. Tukaram Muske, Advaita Datar, Mayur Khanzode, Kumar Madhukar. Efficient elimination of false positives using bounded model checking. *ISSRE'15 Proceedings of the 2015 IEEE 26th international symposium on software reliability engineering.* Gaithersburg, MD, USA. November 2-5, 2015

[35]. M. Junker, R. Huuck, A. Fehnker, A. Knapp. SMT-based false positive elimination in static program analysis". *ICFEM'12 Proceedings of the 14th international conference on formal engineering methods: formal mehods and software engineering.* Kyoto, Japan. November 12-16, 2012

[36]. G. Brat, W. Visser. Combining static analysis and model checking for software analysis tools. *ACE'01 Proceedings of the 16th IEEE international conference on automated software engineering.* San Diego, CA, USA. November 26-29, 2001

[37]. A. Fenker, R. Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Journal Innovations in systems and software engineering. Volume 9, Issue 1, March 2013*

[38]. D. Hovemeyer, W. Pugh. Finding bugs easily. ACM *SIGPLAN notices. Volume 39, issue 12, December 2004*

[39]. D. Evans, D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software. Volume 19, Issue 1, 2002*

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

[41]. C. Flanagan, K Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. *PLDI'02 Proceedings of the ACM SIGPLAN 2002 conference on programming language design and implementation.* Berlin, Germany. June 17-19, 2002

[42]. C. Csallner, Y. Smaragdakis. JCrasher: an automatic robustness testing tester for Java. *Software – Practice & Experience. Volume 34, Issue 11.* September 2004.

[43]. C. Csallner, Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. *ISSTA'06 Proceedings of the 2006 international symposium on software testing and analysis.* Portland, Maide, USA July 17-20, 2006

[44]. O. Chebaro, N Kosmatov, A. Giorgetti, J. Julliand. Programs slicing enhances a verification technique combining static and dynamic analysis. *SAC'12 Proceedings of the 27th annual ACM symposium of applied computing.* Trento, Italy. March 26-30, 2012

[45]. K. Li, C Reichenbach, C. Csallner, Y. Smaragdakis. Residual investigation: predictive and precise bug detection. *ISSTA'2012 Proceedings of the 2012 international symposium on software testing and analysis.* Minneanapolis, MN, USA. July 15-20, 2012

[46]. F. Elberzhager, J. Münch, V.T. Ngoc Nha. A systematic study on the combination of static and dynamic quality assurance techniques. *Infromation and siftware technology. Vol 54, Issue1. January, 2012*

[47]. A. Hanna, H. Z. Ling, X Yang, M. Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. *OTM'09 Proceedings of the confederated international congress, CoopIS, DOA, IS and ADBASE 2009 on on the move to meaningful internet systems: part II.* Vilamoura, Protugal. November 01-06, 2009

[48]. R. Hadjidj, X. Yang, S. Tlili, M. Debabi. Model-checking for software vulnerabilities detection with multi-language support. *PST'08 Proceedings of the 2008 sixth annual conference on privacy, security and trust.* Fredericton, NB, Canada. October 01-03, 2008.

[49]. D. Novillo. Tree SSA: a new optimization infrastructure for GCC. *Proceedings of the GCC developers summit.* Ottawa, ON, Canada. May 25-27, 2003

[50]. S. Schwoon. Model-checking pushdown systems. *PhD thesis.* Technischen Universität München. 2002

[51]. C. Artho, A. Biere. Combined static and dynamic analysis. Technical Report 466, ETH Zürich, Zürich, Switzerland, 2005.

[52]. O. Chebaro, N. Kostomarov, A. Giorgetti, J. Julliand. Combining static analysis and test generation for C program debugging. *TAP'10 Proceedings of the 4th international conference on tests and proofs.* Málaga, Spain. July 01-02, 2010

[53]. N. Williams, B. Marre, P. Mouy, M. Roger. PathCrawler: automatic generation of tests by combining static and dynamic analysis. *EDCC'05 Proceedings of the 5th European conference on dependable computing.* Budapest, Hungary. April 20-22, 2005

[54]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *SEFM'12 Proceedings of the 10th international conference on software engineering and formal methods.* Thesaloniki, Grece. October 01-05, 2012

[55]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *Formal aspects of computing. Volume 27, issue 3. May, 2015*

[56]. A. V. Nori, S. K. Rajamani, S. Tetali, A. V. Thakur. The Yogi ptoject: software property checking via static analysis and testing. *TACAS'09 Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems: held as part of the ETAPS'09 joint European conferences on theory and practice of software.* York, UK. March 22-29, 2009.

[57]. T. Ball, S. K. Rajamani. Slic: a specification language for interface checking (of C). *Technical report MSR-TR2001-21, Microsoft Research.* Redmond, WA, USA. January, 10. 2002

[58]. S. Rawat, D. Ceara, L. Mounier, M.-L. Potet. Combining static and dynamic analysis of vulnerability detection. *Cornell University Library arXiv:1305.3883.* May 16, 2013

[59]. T. Ball. The concept of dynamic analysis. *ESEC/FSE-7 Proceedings of the 7th European software engineering conference held jointly with 7th ACM SIGSOFT international symposium on foundations of software engineering.* Toulouse, France. September 06-10, 1999

[60]. J. Schütte, R. Fedler, D. Titze. ConDroid: targeted dynamic analysis of Android applications. *AINA'15 Proceedings of IEEE 29th international conference on advanced information networking and applications.* Gwangui, South Korea. March 24-27, 2015

[61]. X. Ge, K. Taneja, T. Xie, N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. *ICSE'11 Proceedings of the 33th international conference on software engineering.* Waikiki, Honolulu, HI, USA. May 21-28, 2011

[62]. N. Tillmann, J. de Hallex. Pex – white box test generation for .NET. *TAP'08 Proceedings of the 2nd international conference on tests and proofs.* Prato, Italy. April 09-11, 2008

[63]. M. Y. Wong, D. Lie. IntelliDroid: a targeted input generator for the dynamic analysis of android malware. *NDSS'16 The network and distributed system security symposium 2016.* San Diego, CA, USA. February 21-24, 2016

[64]. H. Gunadi. Formal certification of non-interferent Android bytecode (DEX bytecode). *ICECCS'15 Proceedings of the 2015 20th international conference on engineering and complex computer systems.* Gold Coast, Australia. December 9-12, 2015

[65]. L. De Moura, N. Bjørner. Z3: an efficient SMT solver. *TACAS'08/ETAPS'08 Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the constructions and analysis of systems.* Budapest, Hungary. March 29 – April 06, 2009

[66]. Chen N., Kim S. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering. 2015, volume 41, issue 2.*

[67]. T. Avgerinos, S. Kil Cha, A. Rebert, E. J. Schwartz, M. Woo, D. Brumley. Automatic exploit generation. *Communications of the ACM, volume 57, issue 2, February 2014.*

[68]. M. Li, Y. Chen, L. Wang, G. Xu. Dynamically validating static memory leak warnings. *ISSTA'13 Proceedings of the 2013 international symposium on software testing and analysis.* Lugano, Switzerland. July 15-20, 2013.

[69]. HP Fortify. https://saas.hpe.com/en-us/software/sca, дата обращения 05.05.2017:

[70]. CREST – automatic test generation tool for C. https://github.com/jburnim/crest, дата обращения 05.05.2017

[71]. D. Babić, L. Martignoni, S. McCamant, S. Song. Statically-directed dynamic automated test generation. *ISSTA'11 Proceedings of the 2011 international symposium on software testing and analysis.* Toronto, Ontario, Canada. July 17-21, 2011

[72]. N. Nethercote, J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *PLDI'07 Proceedings of the 28th ACM SIGPLAN Conference on programming languages design and implementation.* San Diego, CA, USA. June 10-13, 2007

[73]. F. Bellard. QEMU, a fast and portable dynamic translator. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[74]. J. Seward, N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[75]. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. ASPLOS XVI Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems. Newport Beach, CA, USA. March 05-11, 2011

Survey on static program analysis results refinement approaches

A.Y. Gerasimov <agerasimov@ispras.ru>
Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia

Abstract. In the present day, software development industry for different classes of computing devices grows at an extremely high speed. Continuously growing power of computational systems presents new opportunities to create powerful, often parallel, programs and software systems. This leads to the growth of complexity of software intended to manage these computational systems. The process of quality assurance calls for new approaches and methods both to check correctness and satisfiability of requirements for software as well as for check software for critical runtime defects and security vulnerabilities.

Program analysis is one of the methods intended to assure software quality. The static and dynamic analysis tool industry has been evolving aggressively since the first decade of 2000th. Nowadays there are many academic research and industrial tools for program analysis. But, due to fundamental limitations and engineering compromises for the sake of performance and scalability, static analysis tools cannot avoid false positive alarms. At the same time, reviewing static analysis tool alarms can take significant time of an experienced software engineer or a software quality assurance specialist. Hence, the task of automating refinement of static analysis tool results becomes more important. This survey covers approaches for static analysis tools result refinement. Approaches, which combine static and dynamic analysis of programs form the principal concern of this paper.

Keywords: static program analysis; dynamic program analysis; combined program analysis.

DOI: 10.15514/ISPRAS-2017-29(3)-6

For citation: Gerasimov A.Y. Survey on static program analysis results refinement approaches. *Trudy ISP RAN/Proc. ISP RAS*, vol. 29, issue 3, 2017, pp. 75-98 (in Russian). DOI: 10.15514/ISPRAS-2017-29(3)-6

References

[1]. Sketch of The Analytical Engine Invented by Charles Babbage by L. F. Menabea from the Bibliotheque Universalle de Geneve, October, 1942, No. 82. With notes upon the Memoir by the translator Ada Augusta, countess of Lovelace. https://www.fourmilab.ch/babbage/sketch.html, accessed 05.05.2017

[2]. Per Runeson, Carian Andersson, Thomas Thelin, Anneliese Andrews, Tomas Berling. What Do We Know about Defect Detection Methods? *IEEE Software May/June 2006*

[3]. IEEE 1044-2009 Standard Classification for Software Anomalies. *IEEE. 3 Park Avenue, New Yourk, NY 10016-5997, USA, 7 January 2010, ISBN 978-0-7381-6114-3*

[4]. Gerald J. Holzmann. The Power of 10: Roles for Developing Safety-Critical Code. *Computer/ 2006, vol. 39, no. 6, pp 95-97*

[5]. MISRA C: 2004 Guidelines for the use of the C language in critical systems. *First published October 2004, by MIRA Limited, Watling Street, Nuneaton, Warkwickshire CV10 0TU UK, ISBN 978-0-9524156-4-0*

[6]. E. J. Weyuker, T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on software engineering, 6(3):236-246.* May 1980.

[7]. E. W. Dijkstra. On the reliability of the programs. https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD303.PDF, accessed 05.05.2017

[8]. Dennis M. Ritchie. The development of the C language. *Proceedings of HOPL-II The second ACM SIGPLAN conference on History of programming languages. Cambridge, MA, USA – April 20-23, 1993, pp. 201-208*

[9]. S. C. Johnson. A Portable Compiler: Theory and Practice. *Proceedings of 5th ACM POPL Symposium*, January 1978

[10]. S. C. Johnson. Lint, a Program Checker. *Unix Programmer's manual*, Seventh Edition, Vol. 2B, M.D. McIlroy and B.W. Kernigan, eds. AT&T Bell Laboratories: Murray Hill, NJ, 1979.

[11]. Benjamine Chelf, Andy Chou. The next generation of Static Analysis. *Coverity*, March 18, 2008. http://www.coverity.com/library/pdf/Coverity_White_Paper-SAT-Next_Generation_Static_Analysis.pdf, accessed 05.05.2017

[12]. Pär Emanuelsson, Ulf Nilsson, A Comparative Study of Industrial Static Analysis Tools. *Technical report. Department of Computer and Information Science, Linköping University.* Linköping, Sweden, 2008.

[13]. Dawson Engler, Benjamin Chelf, Andy Chou, Seth Hallem. Checking system rules using system-specific, programmer-written compiler extensions. OSDI'00 *Proceedings of the 4th conference on Symposium on Operating System Design and Implementation, Volume 4, Article No. 1*. San Diego, California – October 22-25, 2000

[14]. Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, Robert Bowdidge. Why don't software developers use static analysis tools to find bugs?. *ICSE'13 Proceedings of the 2013 International conference on Software Engineering.* San Francisco, CA, USE, May 18-26, 2013

[15]. John Franco, John Martin. A history of Satisfiability. *Handbook of Satisfiability.* IOS Press, 2009. doi:10.3233/978-1-58603-929-5-3

[16]. Coverity Scan: 2012 Open Source Report. http://wpcme.coverity.com/wp-content/uploads/2012-Coverity-Scan-Report.pdf, accessed 05.05.2017

[17]. Coverity Scan. Project Spotlight: Python. http://wpcme.coverity.com/wp-content/uploads/2013-Coverity-Scan-Spotlight-Python.pdf, accessed 05.05.2017

[18]. Tukaram Muske, Alexander Serebrenik. Survey of Approaches for Handling Static Analysis Alarms. *Proceedings of IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM).* Raleigh, NC, USA. October 2-3, 2016

[19]. Woosuk Lee, Wonchan Lee, Kwengkeun Yi. Sound non-statistical clustering of static analysis alarms. *VMCAI'12 Proceedings of the 13th international conference on verification, model checking and abstract verification interpretation.* Philadelphia, PA, USA. January 22-24, 2012.

[20]. Zachary P. Fry, Westley Weimer. Clustering static analysis defect reports to Reduce maintenance costs. *WCRE'13 Proceeding of 30th working conference on reverse engineering.* Koblenz, Germany. October 14-17, 2013.

[21]. Ted Kremenek, Dawson Engler. Z-ranking: using statistical analysis to counter the impact of static analysis approximations. *SAS'03 Proceedings of the 10th International conference on static analysis.* San Diego, CA, USA. June 11, 2003.

[22]. Sunghun Kim, Michael D. Ernst. Prioritizing Warning Categories by Analyzing Software History. *MSR'07 Proceedings of the Fourth International Workshop on Mining Software Repositories.* Minneapolis, MN, USA. May 20-26, 2007.

[23]. Sunghun Kim, Michael D. Ernst. Which warnings should I fix first? *ESEC-FSE'07 Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering.* Dubrovnik, Croatia. September 03-07, 2007

[24]. Haihao Shen, Jianhong Fang, Jianjun Zhao. EFindBugs: Effective Error Ranking for FindBugs. *ICST'11 Proceedings of the 2011 Fourth IEEE International conference on Software Testing, Verification and Validation.* Berlin, Germany. March 21-25, 2011

[25]. Sunghun Kim, Michael D. Ernst. Prioritizing Software Inspection Results using Static Profiling. *SCAM'06 Source code analysis and manipulation.* Philadelphia, PA, USA. December 11, 2006.

[26]. Deguang Kong, Quan Zheng, Chao Chen, Jianmei Shuai, Ming Zhu. ISA: A source code static vulnerability detection system based on data fusion. *InfoScale'07 Proceedings of*

the 2nd international conference on Scalable information systems. Suzhou, China. June 06-08, 2007

[27]. Na Meng, Qianxiang Wang, Qian Wu, Hong Mei. An approach to merge results of multiple static analysis tools. *QSIC'08 Proceedings of the 2008 the eighth international conference on quality software.* Oxford, UK. August 12-13, 2008

[28]. Quinn Hanam, Lin Tan, Reid Holmes, Patrick Lam. Finding patters in static analysis alerts. Improving actionable alert ranking. *MSR 2014 Proceedings of the 11th working conference on mining software repositories.* Hyderabad, India. May 31 – June 01, 2014

[29]. Ulas Yüskel, Hasan Sözer. Automated classification of static code analysis alerts: a case study. *ICSM'13 Proceedings of the 2013 IEEE international conference on software maintenance.* Eindhoven, Netherlands. September 24-26, 2013

[30]. Jaime Spacco, David Hovermeyer, William Pugh. Tracking defect warnings across versions. *MSR'06 Proceedings of the 2006 international workshop on mining software repositories.* Shanghai, China. May 22-23, 2006

[31]. Brahti Chimdyalwar, Shrawan Kumar. Effective false positive filtering for evolving software. *ISEC'11 Proceedings of the 4th India software engineering conference.* Thiruvananthapuram, Kerala, India. February 24-27, 2011

[32]. J. R. Rithruff, J. Penix, J. D. Morgenthaler, S. Elbaum, G. Rothermel. Predicting accurate and actionable static analysis warnings: and experimental approach. *ICSE'08 Proceedings of the 30th international conference on software engineering.* Leipzig, Germany. May 10-18, 2008

[33]. H. Post, C. Sinz, A. Kaiser, T. Gorges. Reducing false positives by combining abstract interpretation and bounded model checking. *ASE'08 Proceedings of the 2008 23rd IEEE/ACM international conference on automated software engineering.* L'Aquila, Italy. September 15-19, 2008

[34]. Tukaram Muske, Advaita Datar, Mayur Khanzode, Kumar Madhukar. Efficient elimination of false positives using bounded model checking. *ISSRE'15 Proceedings of the 2015 IEEE 26th international symposium on software reliability engineering.* Gaithersburg, MD, USA. November 2-5, 2015

[35]. M. Junker, R. Huuck, A. Fehnker, A. Knapp. SMT-based false positive elimination in static program analysis". *ICFEM'12 Proceedings of the 14th international conference on formal engineering methods: formal mehods and software engineering.* Kyoto, Japan. November 12-16, 2012

[36]. G. Brat, W. Visser. Combining static analysis and model checking for software analysis tools. *ACE'01 Proceedings of the 16th IEEE international conference on automated software engineering.* San Diego, CA, USA. November 26-29, 2001

[37]. A. Fenker, R. Huuck. Model checking driven static analysis for the real world: designing and tuning large scale bug detection. *Journal Innovations in systems and software engineering. Volume 9, Issue 1, March 2013*

[38]. D. Hovemeyer, W. Pugh. Finding bugs easily. ACM *SIGPLAN notices. Volume 39, issue 12, December 2004*

[39]. D. Evans, D. Larochelle. Improving security using extensible lightweight static analysis. *IEEE Software. Volume 19, Issue 1, 2002*

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

[41]. C. Flanagan, K Rustan, M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, R. Stata. Extended static checking for Java. *PLDI'02 Proceedings of the ACM SIGPLAN 2002*

conference on programming language design and implementation. Berlin, Germany. June 17-19, 2002

[42]. C. Csallner, Y. Smaragdakis. JCrasher: an automatic robustness testing tester for Java. *Software – Practice & Experience. Volume 34, Issue 11*. September 2004.

[43]. C. Csallner, Y. Smaragdakis. DSD-Crasher: a hybrid analysis tool for bug finding. *ISSTA'06 Proceedings of the 2006 international symposium on software testing and analysis.* Portland, Maide, USA July 17-20, 2006

[44]. O. Chebaro, N Kosmatov, A. Giorgetti, J. Julliand. Programs slicing enhances a verification technique combining static and dynamic analysis. *SAC'12 Proceedings of the 27th annual ACM symposium of applied computing*. Trento, Italy. March 26-30, 2012

[45]. K. Li, C Reichenbach, C. Csallner, Y. Smaragdakis. Residual investigation: predictive and precise bug detection. *ISSTA'2012 Proceedings of the 2012 international symposium on software testing and analysis.* Minneanapolis, MN, USA. July 15-20, 2012

[46]. F. Elberzhager, J. Münch, V.T. Ngoc Nha. A systematic study on the combination of static and dynamic quality assurance techniques. *Infromation and siftware technology. Vol 54, Issue1. January, 2012*

[47]. A. Hanna, H. Z. Ling, X Yang, M. Debbabi. A synergy between static and dynamic analysis for the detection of software security vulnerabilities. *OTM'09 Proceedings of the confederated international congress, CoopIS, DOA, IS and ADBASE 2009 on on the move to meaningful internet systems: part II*. Vilamoura, Protugal. November 01-06, 2009

[48]. R. Hadjidj, X. Yang, S. Tlili, M. Debabi. Model-checking for software vulnerabilities detection with multi-language support. *PST'08 Proceedings of the 2008 sixth annual conference on privacy, security and trust.* Fredericton, NB, Canada. October 01-03, 2008.

[49]. D. Novillo. Tree SSA: a new optimization infrastructure for GCC. *Proceedings of the GCC developers summit.* Ottawa, ON, Canada. May 25-27, 2003

[50]. S. Schwoon. Model-checking pushdown systems. *PhD thesis.* Technischen Universität München. 2002

[51]. C. Artho, A. Biere. Combined static and dynamic analysis. Technical Report 466, ETH Zürich, Zürich, Switzerland, 2005.

[52]. O. Chebaro, N. Kostomarov, A. Giorgetti, J. Julliand. Combining static analysis and test generation for C program debugging. *TAP'10 Proceedings of the 4th international conference on tests and proofs.* Málaga, Spain. July 01-02, 2010

[53]. N. Williams, B. Marre, P. Mouy, M. Roger. PathCrawler: automatic generation of tests by combining static and dynamic analysis. *EDCC'05 Proceedings of the 5th European conference on dependable computing.* Budapest, Hungary. April 20-22, 2005

[54]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *SEFM'12 Proceedings of the 10th international conference on software engineering and formal methods.* Thesaloniki, Grece. October 01-05, 2012

[55]. P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, B. Yakobowski. Frama-C: a software analysis perspective. *Formal aspects of computing. Volume 27, issue 3. May, 2015*

[56]. A. V. Nori, S. K. Rajamani, S. Tetali, A. V. Thakur. The Yogi ptoject: software property checking via static analysis and testing. *TACAS'09 Proceedings of the 15th international conference on tools and algorithms for the construction and analysis of systems: held as*

part of the ETAPS'09 joint European conferences on theory and practice of software. York, UK. March 22-29, 2009.

[57]. T. Ball, S. K. Rajamani. Slic: a specification language for interface checking (of C). *Technical report MSR-TR2001-21, Microsoft Research.* Redmond, WA, USA. January, 10. 2002

[58]. S. Rawat, D. Ceara, L. Mounier, M.-L. Potet. Combining static and dynamic analysis of vulnerability detection. *Cornell University Library arXiv:1305.3883.* May 16, 2013

[59]. T. Ball. The concept of dynamic analysis. *ESEC/FSE-7 Proceedings of the 7th European software engineering conference held jointly with 7th ACM SIGSOFT international symposium on foundations of software engineering.* Toulouse, France. September 06-10, 1999

[60]. J. Schütte, R. Fedler, D. Titze. ConDroid: targeted dynamic analysis of Android applications. *AINA'15 Proceedings of IEEE 29th international conference on advanced information networking and applications.* Gwangui, South Korea. March 24-27, 2015

[61]. X. Ge, K. Taneja, T. Xie, N. Tillmann. DyTa: dynamic symbolic execution guided with static verification results. *ICSE'11 Proceedings of the 33th international conference on software engineering.* Waikiki, Honolulu, HI, USA. May 21-28, 2011

[62]. N. Tillmann, J. de Hallex. Pex – white box test generation for .NET. *TAP'08 Proceedings of the 2nd international conference on tests and proofs.* Prato, Italy. April 09-11, 2008

[63]. M. Y. Wong, D. Lie. IntelliDroid: a targeted input generator for the dynamic analysis of android malware. *NDSS'16 The network and distributed system security symposium 2016.* San Diego, CA, USA. February 21-24, 2016

[64]. H. Gunadi. Formal certification of non-interferent Android bytecode (DEX bytecode). *ICECCS'15 Proceedings of the 2015 20th international conference on engineering and complex computer systems.* Gold Coast, Australia. December 9-12, 2015

[65]. L. De Moura, N. Bjørner. Z3: an efficient SMT solver. *TACAS'08/ETAPS'08 Proceedings of the theory and practice of software, 14th international conference on tools and algorithms for the constructions and analysis of systems.* Budapest, Hungary. March 29 – April 06, 2009

[66]. Chen N., Kim S. STAR: stack trace based automatic crash reproduction via symbolic execution. *IEEE transactions on software engineering. 2015, volume 41, issue 2.*

[67]. T. Avgerinos, S. Kil Cha, A. Rebert, E. J. Schwartz, M. Woo, D. Brumley. Automatic exploit generation. *Communications of the ACM, volume 57, issue 2, February 2014.*

[68]. M. Li, Y. Chen, L. Wang, G. Xu. Dynamically validating static memory leak warnings. *ISSTA'13 Proceedings of the 2013 international symposium on software testing and analysis.* Lugano, Switzerland. July 15-20, 2013.

[69]. HP Fortify. https://saas.hpe.com/en-us/software/sca, accessed 05.05.2017

[70]. CREST – automatic test generation tool for C. https://github.com/jburnim/crest, accessed 05.05.2017

[71]. D. Babić, L. Martignoni, S. McCamant, S. Song. Statically-directed dynamic automated test generation. *ISSTA'11 Proceedings of the 2011 international symposium on software testing and analysis.* Toronto, Ontario, Canada. July 17-21, 2011

[72]. N. Nethercote, J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *PLDI'07 Proceedings of the 28th ACM SIGPLAN Conference on programming languages design and implementation.* San Diego, CA, USA. June 10-13, 2007

[73]. F. Bellard. QEMU, a fast and portable dynamic translator. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[74]. J. Seward, N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. *ATEC'05 Proceedings of the annual conference on USENIX annual technical conference.* Anaheim, CA, USA. April 10-15, 2005

[75]. V. Chipounov, V. Kuznetsov, G. Candea. S2E: a platform for in-vivo multi-path analysis of software systems. ASPLOS XVI Proceedings of the sixteenth international conference on architectural support for programming languages and operating systems. Newport Beach, CA, USA. March 05-11, 2011