ՀՀ ԳԱԱ ԻՆՖՈՐՄԱՏԻԿԱՅԻ ԵՎ ԱՎՏՈՄԱՏԱՑՄԱՆ ՊՐՈԲԼԵՄՆԵՐԻ ԻՆՍՏԻՏՈՒՏ

Ասրյան Սերյոժա Արմենի

ԿԱՏԱՐՎՈՂ ԿՈԴՈՒՄ ՍԽԱԼՆԵՐԻ ՀԱՅՏՆԱԲԵՐՄԱՆ ԴԻՆԱՄԻԿ ՎԵՐԼՈՒԾՈՒԹՅԱՆ ՄԵԹՈԴՆԵՐ

Ե.13.04 – «Հաշվողական մեքենաների, համալիրների, համակարգերի և ցանցերի մաթեմատիկական և ծրագրային ապահովում» մասնագիտությամբ տեխնիկական գիտությունների թեկնածուի գիտական աստիճանի հայցման ատենախոսության

ՍԵՂՄԱԳԻՐ

Երևան - 2019

ИНСТИТУТ ПРОБЛЕМ ИНФОРМАТИКИ И АВТОМАТИЗАЦИИ НАН РА

Асрян Серёжа Арменович

МЕТОДЫ ОБНАРУЖЕНИЯ ДЕФЕКТОВ В ИСПОЛНЯЕМОМ КОДЕ НА ОСНОВЕ ДИНАМИЧЕСКОГО АНАЛИЗА

АВТОРЕФЕРАТ

диссертации на соискание ученой степени кандидата технических наук по специальности 05.13.04 – «Математическое и программное обеспечение вычислительных машин, комплексов, систем и сетей»

Ереван - 2019

Ատենախոսության թեման հաստատվել է ՀՀ ԳԱԱ Ինֆորմատիկայի և ավտոմատացման պրոբլեմների ինստիտուտում:

Գիտական ղեկավար՝ ֆիզ. մաթ. գիտ. դոկտոր Հ. Ի. Ավետիսյան

Պաշտոնական ընդդիմախոսներ՝ ֆիզ.մաթ. գիտ. դոկտոր Ս.Կ. Շուքուրյան
տեխ. գիտ. թեկնածու Մ.Ղ. Գյուրջյան

Առաջատար կազմակերպություն՝ Հայաստանի Ազգային Պոլիտեխնիկական Համալսարան

Պաշտպանությունը կայանալու է 2019 թվականի հունիսի 18-ին, ժ. 16:00-ին ՀՀ ԳԱԱ Ինֆորմատիկայի և ավտոմատացման պրոբլեմների ինստիտուտում գործող 037 «Ինֆորմատիկա» մասնագիտական խորհրդի նիստում հետևյալ հասցեով՝ Երևան, 0014, Պ. Սևակի 1:

Ատենախոսությանը կարելի է ծանոթանալ ՀՀ ԳԱԱ ԻԱՊԻ գրադարանում:
Սեղմագիրն առաքված է 2019թ. մայիսի 8-ին:

Մասնագիտական խորհրդի
գիտական քարտուղար, ֆ.մ.գ.դ. Հ.Գ. Սարուխանյան

Тема диссертации утверждена в Институте проблем информатики и автоматизации НАН РА

Научный руководитель: доктор физико-математических наук А. И. Аветисян

Официальные оппоненты: доктор физ.-мат. наук С. К. Шукурян
кандидат тех. наук М. Г. Гюрджян

Ведущая организация: Национальный Политехнический Университет Армении

Защита состоится 18-ого июня 2019г. в 16:00, на заседании специализированного совета 037 “Информатика” в Институте проблем информатики и автоматизации НАН РА по адресу: 0014, г. Ереван, ул. П. Севака 1.

С диссертацией можно ознакомится в библиотеке ИПИА НАН РА.
Автореферат разослан 8-ого мая 2019г.

Ученый секретарь специализированного совета
доктор физ. мат. наук: А. Г. Саруханян

Общая характеристика работы

Актуальность работы. В современном мире разработка надежного программного обеспечения (ПО) является одной из важнейших задач в области информационных технологий. С ростом сложности и объёма программных комплексов все труднее становится разработка эффективных методов анализа качества и надежности ПО. Наличие всего одной ошибки в коде может привести к серьёзным последствиям. Например, дефект в коде библиотеки OpenSSL под названием *HeartBleed* привел к утечке шифрованных данных на многочисленных серверах. Если раньше поиск ошибок выполнялся вручную или с учетом предупреждений компиляторов, то на сегодняшний день применяются различные методы, которые позволяют частично или полностью автоматизировать процесс анализа ПО. Выделяются два основных подхода к исследованию программ – статический и динамический анализ. При проведении статического анализа не производится запуск исследуемой программы. Инструменты статического анализа используют промежуточные представления программ *(абстрактное синтаксическое дерево, граф потока управления и т.д.)* и осуществляют их обработку. На основе исследования промежуточного представления выполняется поиск ошибок. Как правило, статические анализаторы хорошо масштабируются, но имеют множество ложных срабатываний.

Динамический анализ представляет собой исследование программы во время ее выполнения. Проверяются непосредственно те фрагменты кода, которые были выполнены. Анализ проводится в среде выполнения самой программы с использованием конкретных входных данных. Это позволяет в основном устранять ложные срабатывания и находить конкретные входные данные, воспроизводящие найденные дефекты. К данному направлению относятся фаззинг *(fuzzing)*, динамическое символьное выполнение, методы бинарной трансляции, отладка и профилирование ПО, применение средств компиляторов, таких как санитайзеры и т.д.

Фаззинг[1][2] считается одним из распространённых методов динамического анализа. Принцип работы фаззинга заключается в запуске программ на случайных входных данных и отслеживании их поведения. Входные данные генерируются на основе применения различных алгоритмов мутаций. Некоторые продвинутые инструменты фаззинга учитывают покрытие кода с помощью предварительной или динамической инструментации. На основе этого можно определить, какие входные данные приводят к открытию нового пути выполнения. Это позволяет фаззеру приоритизировать входные данные.

Другим методом динамического анализа является динамическое символьное выполнение[3]. Сам метод основан на замене значений входных данных программы

[1] A. Roychoudhury, V.-T. Pham, M. Böhme, “Coverage-based Greybox Fuzzing as Markov Chain”, CCS '16 Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, pages. 1032-1043, 2016

[2] M. Böhme, V. Pham, M. Nguyen, A. Roychoudhury, “Directed Greybox Fuzzing”, Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, USA, pages 2329-2344, Nov. 2017

[3] S. K. Cha, T. Avgerinos, A. Rebert, D. Brumley, “Unleashing Mayhem on Binary Code”, в SP '12 Proceedings of the 2012 IEEE Symposium on Security and Privacy, стр. 380-394, 2012

символьными переменными. Используя полученные символьные переменные, составляются ограничения на входных данных на основе предикатов в операторах ветвления. Каждое из этих ограничений описывает условие для выполнения конкретного пути в программе. Применение SMT *(Satisfiability modulo theories)*[4] решателей для полученных ограничений позволяет генерировать входные данные для исследования новых путей программы.

Методы динамического анализа пытаются анализировать все выполняемые пути в программе, что в общем случае является невыполнимой задачей из-за экспоненциального роста путей от количества условных переходов. Результативность анализа ПО может быть существенно увеличена за счет объединения методов динамического и статического анализа. С помощью статического анализа можно выполнять поиск дефектов в ПО *(переполнение буфера, повторное освобождение памяти и т.д.)*. Имея точки потенциальных дефектов, фаззер *(или динамическое символьное выполнение)* сможет сгенерировать входные данные, которые могут привести к их воспроизведению. Для этого строятся пути в программе, приводящие к полученным точкам. Каждый из этих путей представляет собой последовательность базовых блоков, достигающих точек потенциальных дефектов. Далее определяются функции соответствия и применяются генетические алгоритмы мутаций, чтобы выполнить оценку входных данных на основе сравнения трасс выполнения программы и построенных путей. При этом используются разные метрики сравнения трассы выполнения программы с построенными путями для выбора наиболее подходящих входных данных, на основе которых будет выполняться дальнейший анализ.

Анализ приложений, таких как компиляторы и интерпретаторы, является еще одной чрезвычайно важной задачей. Ошибки в компиляторах могут привести к генерации некорректного бинарного кода. Чтобы осуществить анализ глубоких стадий компиляции *(семантический анализ, генерация промежуточного кода и др.)*, необходимо обеспечить структурную целостность входных данных. Это позволит пройти дальше стадий лексического и синтаксического анализа. Существующие инструменты выполняют генерацию входных данных на основе БНФ. Однако из-за тесной связи между логикой генерации данных и спецификациями языков программирования данные методы ограничены с точки зрения расширяемости для новых языков программирования.

Целью диссертационной работы является исследование и разработка методов и программных средств динамического анализа исполняемого кода для обнаружения дефектов. Методы должны обеспечить взаимодействие со статическим анализом, позволяя итеративным образом увеличить качество каждого из анализов, а также иметь возможность выполнять анализ приложений, обрабатывающих структурированные данные.

Для достижения поставленных целей были сформулированы и решены следующие задачи:

1. Разработать платформу, которая позволяет объединить методы динамического и

[4] Leonardo de Moura, Nikolaj Bjørner, “Z3: an efficient SMT solver”, Proceedings of the Theory and practice of software,Budapest, Hungary, 2008, pages 337-340

статического анализа для взаимного улучшения каждого из методов.

2. Разработать методы направленного фаззинга для анализа конкретных участков кода программы.
3. Повысить эффективность динамического символьного выполнения путем взаимодействия с фаззингом для проведения направленного анализа.
4. Разработать алгоритм анализа глубоких стадий работы компиляторов, интерпретаторов и трансляторов, направляя процесс генерации входных данных.

Научная новизна. В рамках данной работы получены следующие основные результаты, обладающие научной новизной:

- Разработана и реализована платформа динамического анализа, обеспечивающая взаимодействие со статическими методами анализа и позволяющая итеративным образом улучшить результаты обоих методов.
- Разработан и реализован метод направленного динамического анализа, основанный на выполнении частичной инструментации и ограничении анализируемой части исполняемого кода.
- Разработан и реализован метод на основе интеграции динамического символьного выполнения с фаззером, позволяющий применить динамическое символьное выполнение на отдельных путях исполнения ПО.
- Разработан и реализован алгоритм для анализа компиляторов и интерпретаторов на основе генерации синтаксически корректных входных данных с целью исследования глубоких стадий работы этих программ. Данный алгоритм использует обратную связь с фаззером для итеративного улучшения качества сгенерированных данных.

Практическая значимость. Все разработанные методы динамического анализа используются в процессе разработки безопасного ПО в Институте системного программирования им. В.П. Иванникова РАН. Эффективность реализованных методов подтверждается полученными результатами анализа как на наборе тестовых программ, так и на реальных проектах.

Апробация работы и публикации. По теме диссертации опубликовано 7 работ в изданиях из перечня рецензируемых научных изданий ВАК. Две из них были индексированы в журнале Scopus. Список работ представлен в конце автореферата. Основные результаты диссертационной работы докладывались на следующих конференциях:

- 60-ая научная конференция МФТИ, 20-25 ноября 2017 г., Долгопрудный, Россия.
- Международная конференция Иванниковские чтения (Ivannikov Memorial Workshop), РАУ, 3-4 мая 2018 г., Ереван, Армения.
- Открытая конференция Института системного программирования им. В.П. Иванникова РАН, 22-23 ноября 2018 г., Москва, Россия.

Структура и объем работы. Диссертация состоит из введения, 4 глав и заключения. Работа изложена на 103 страницах. Список источников насчитывает 89 наименований. Диссертация содержит 6 таблиц и 18 рисунков.

КРАТКОЕ СОДЕРЖАНИЕ РАБОТЫ

Во введении представляются методы автоматического анализа программного обеспечения и рассматриваются комбинированные решения. Формулируются цели и задачи работы, обосновывается их актуальность, обсуждаются вопросы практического применения разработанных методов и инструментов, проводится краткий обзор работы.

Глава 1 содержит описание методов, имеющих непосредственное отношение к теме диссертации.

Раздел 1.1 описывает общую архитектуру инструментов фаззинга (*Рисунок 1*). Метод фаззинга представляет собой генерацию случайных наборов входных данных с последующей передачей их тестируемой программе. Цель этого метода – получить набор входных данных, выявляющих дефекты в ПО. Генерация входных данных осуществляется на основе начальных тестовых примеров с применением набора мутаций. Существуют различные алгоритмы мутаций, такие как изменение значений битов во входных данных, прибавление или вычитание определенных целых чисел, объединение нескольких тестовых примеров и т.д. Каждая итерация фаззинга приводит к рассмотрению единственного пути выполнения программы на основе использования одного из сгенерированных входных данных. Следовательно, чтобы достичь большого покрытия кода, необходимо генерировать входные данные, обеспечивающие выполнение отличающихся путей программы. Для достижения этой цели можно использовать информацию о поведении программы при каждом ее запуске. Это можно осуществить двумя основными способами: инструментация программы – встраивание дополнительных инструкций в коде программы, для получения информации о пройденных путях, сохраняя при этом логику самой программы. Применяется как статическая, так и динамическая инструментация. В первом случае инструментация выполняется один раз, перед запуском программы. В случае динамической инструментации, встраивание дополнительных инструкций выполняется на тех участках программы, которые были непосредственно выполнены *(англ. runtime instrumentation)*. Второй способ — это применение динамической бинарной трансляции *(примером такой системы является QEMU)*. В этом случае необходимая информация получается во время эмуляции целевой программы. Одними из наиболее распространённых инструментов фаззинга являются AFL[5], libFuzzer[6], honggfuzz[7]. AFL поддерживает рёберное покрытие[2] программы и использует полученную информацию для генерации входных данных на основе генетического алгоритма. Информация о покрытии кода получается с помощью статической инструментации или бинарной трансляции тестируемой программы. Еще один инструмент libFuzzer является частью компиляторной инфраструктуры LLVM и позволяет анализировать отдельные функции программ. Для осуществления анализа функций необходимо вручную дописывать вызовы библиотечных функций с описанием данных, которые принимают исследуемые функции.

[5] Страница инструмента American Fuzzy Lop (AFL), *http://lcamtuf.coredump.cx/afl*

[6] Страница инструмента libfuzzer (a library for coverage-guided fuzz testing), *https://llvm.org/docs/LibFuzzer.html*

[7] Страница инструмента honggfuzz, *http://google.github.io/honggfuzz*

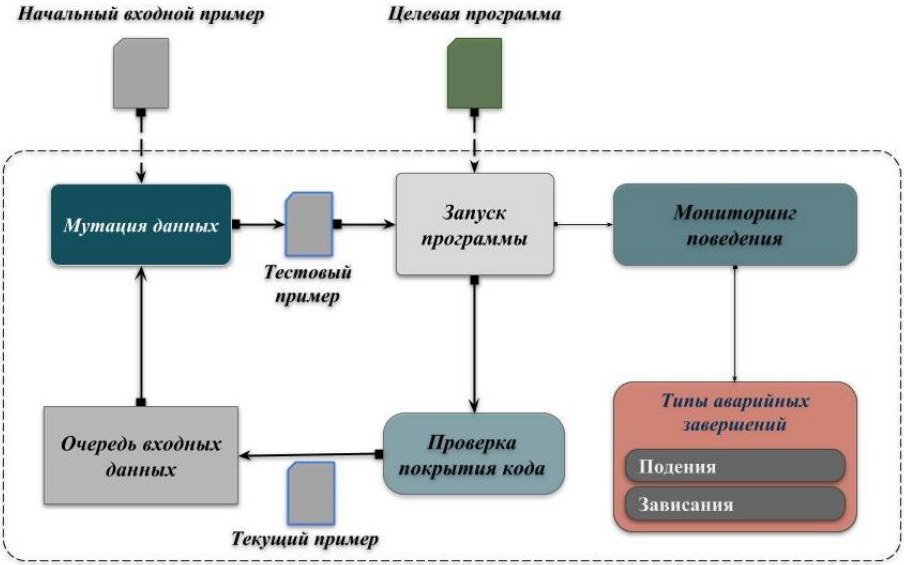

Рисунок 1. Общая структура фаззера

Раздел 1.2 описывает методы проведения динамического анализа с помощью объединения фаззинга и динамического символьного выполнения. Современные инструменты фаззинга успешно применяются на практике и позволяют обнаружить множество аварийных завершений. Однако в случае программ, содержащих сложные вычислительные проверки, фаззеру становится все труднее генерировать данные, приводящие к увеличению покрытия кода программы и, следовательно, к обнаружению новых ошибок. Одним из способов решения этой проблемы является комбинирование символьного выполнения с фаззером для генерации входных данных, удовлетворяющих сложным проверкам. В данном случае символьное выполнение *(Avalanche[8], Klee[9], S2E[10] и т.д.)* применяется для увеличения покрытия кода анализируемой программы. Во время символьного выполнения входные данные программы выступают в качестве символьных переменных, которые могут принимать произвольные значения. Когда выполняется инструкция условного перехода, содержащая символьные переменные, составляются функциональные зависимости между этими переменными, соответствующими операндами и результатами инструкции. На основе функциональных зависимостей строится трасса ограничений на пути выполнения (*или на ее отдельном фрагменте*). Свободными переменными в созданной трассе ограничений являются символьные переменные, соответствующие данным, получаемым из внешних источников. Все остальные данные в трассе ограничений (*результаты выполнения проверок в точках ветвления*), определены

[8] Страница Avalanche, http://ispras.ru/technologies/avalanche_dynamic_program_analysis_tool
[9] Страница KLEE, https://klee.github.io
[10] СтраницаS2E, https://s2e.systems

относительно внешних данных. Для построения входных данных, удовлетворяющих трассе ограничений, используются SMT решатели. Это позволяет составлять реальные входные данные для обхода путей выполнения. Основным недостатком этого метода является сильное замедление работы программы и экспоненциальный рост путей *(англ. path explosion)* выполнения от количества условных переходов, которые необходимо анализировать.

Объединение методов фаззинга и динамического символьного выполнения *(Driller*[11]*, Munch*[12]*)* предоставляет возможность компенсировать их недостатки. Если в течение некоторого времени фаззеру не удалось сгенерировать входные данные, приводящие к новому переходу в программе, запускается метод динамического символьного выполнения. Этот метод применяется для построения входных данных, позволяющих выполнить пути в программе, ранее не рассмотренные фаззером. Данный метод использует *интересные* тестовые примеры, полученные фаззером. Входные данные считаются интересными для фаззера, если их использование привело к открытию нового пути. Все примеры, полученные в результате динамического символьного выполнения, передаются фаззеру для дальнейшего исследования.

Раздел 1.3 посвящен рассмотрению методов, объединяющих динамический и статический анализ. С помощью совместного применения этих методов можно использовать интересующую информацию из статического анализа *(CppCheck*[13]*, Svace*[14]*, BinSide*[15]*)* для выполнения направленного динамического анализа. Один из существующих методов[16] использует статический анализ для вычисления последовательности зависимостей помеченных данных *(Taint Dependency Sequences - TDS)*, представляющей собой пути в программе, содержащие дефектные точки. После чего используется динамический анализ *(фаззинг)* для генерации конкретных данных, приводящих к выполнению путей, содержащих точки дефектов. Для каждого входного файла сохраняется трасса выполнения программы. Эти трассы используются для повышения эффективности фаззинга. Обработка входных данных производится на основе сравнения полученных трасс с путями из последовательности TDS: для последующей обработки выбираются те входные данные, которые привели к исследованию путей, наиболее совпадающих с путями из *TDS*. Другой инструмент[17] осуществляет комбинирование

[11] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, G. Vigna, “Driller: Augmenting Fuzzing Through Selective Symbolic Execution”, Network and Distributed System Security Symposium, 2016

[12] S. Ognawala, T. Hutzelmann, E. Psallida, A. Pretschner, “Improving function coverage with munch: a hybrid fuzzing and directed symbolic execution approach”, SAC '18 Proceedings of the 33rd Annual ACM Symposium on Applied Computing, Pau, France, pages 1475-1482, 2018

[13] M. Daniel, “Cppcheck design”. – 2010,
http://www.cs.kent.edu/~rothstei/spring_12/secprognotes/cppcheck-design.pdf

[14] В.П. Иванников, А.А. Белеванцев, А.Е. Бородин и др., “Статический анализатор Svace для поиска дефектов в исходном коде программ”, Труды Института системного программирования РАН (электронный журнал), 2014, Т. 26, № 1, стр. 231–250.

[15] Страница BINSIDE. Static binary code analysis tool, http://www.ispras.ru/en/technologies/binside/

[16] S. Rawat, Combining, “Static and Dynamic Analysis for Vulnerability Detection”

[17] J. Feist, L. Mounier, S. Bardin, R. David, M. “PotetFinding the needle in the heap: combining static analysis

статического анализа с динамическим символьным выполнением. Статической анализ используется для поиска в программе ошибок «*использования памяти после обнаружения*» (*англ. Use-After-Free*). Далее для каждой обнаруженной ошибки инструмент статического анализа возвращает разрезы графа потока управления. Для всех построенных разрезов вычисляются весовые значения, которые используются для вычисления расстояния (*англ. distance score*) до точек ошибок. После завершения статического анализа запускается инструмент динамического символьного выполнения, который применяет полученные разрезы для исследования полученных небольших частей программы.

В разделе 1.4 приводится описание методов автоматического тестирования компиляторов и интерпретаторов. Из-за сложной структуры входных данных тестирование подобных приложений с помощью фаззера связано с определенными трудностями (*входные данные должны соответствовать синтаксическим и семантическим правилам языка*). Для полного тестирования необходимо генерировать как валидные программы, так и программы, которые тем или иным способом не соответствуют спецификациям языка. Генерация тестов может быть выполнена двумя методами. В первом случае программы пишутся вручную, с учетом спецификаций определенного языка. Второй метод основан на использовании БНФ *(Форма Бэкуса - Наура)* представления входных данных для генерации тестовых программ. Последний из перечисленных может быть легко расширяемым в плане поддержки новых языков.

Поскольку стандартные мутации фаззера не учитывают структуры входных данных, исследуются только начальные этапы компиляции (*лексический и синтаксический анализы*). Для решения этой проблемы многие инструменты *(CSmith[18],GramFuzz[19], Superion[20], Ifuzzer[21])* используют описания языков программирования в форме БНФ. Основной идеей применения БНФ грамматик является генерация набора тестовых программ, отвечающих синтаксическим правилам соответствующих языков. Существующие инструменты имеют два основных недостатка. Во-первых, ограничено количество поддерживаемых грамматик. Во-вторых, генерация тестовых программ выполняется либо с помощью произвольного обхода грамматических правил, либо некоторым фиксированным способом. Чтобы увеличить точность сгенерированных тестов, можно использовать информацию о покрытии кода программы для разработки некоторой модели генерации программ (*например, используя весовые значения для обхода грамматик*).

В разделе 1.5 в качестве основной задачи диссертационной работы ставится разработка платформы, обеспечивающей итеративное взаимное улучшение эффективности динамических и статических методов анализа, позволяющей использовать

---

and dynamic symbolic execution to trigger use-after-free", California, 2016,

[18] Xuejun Yang, Yang Chen, Eric Eide, John Regehr, "Finding and understanding bugs in C compilers", PLDI '11 Proceedings of the 32nd ACM SIGPLAN, San Jose, California, USA, pp. 283-294, 2011

[19] Tao Guo, P. Zhang, X. Wang and Q. Wei, "GramFuzz: Fuzzing testing of web browsers based on grammar analysis and structural mutation", pp. 212-215, 2013

[20] J. Wang, B. Chen, L. Wei, Y. Liu, "Superion: Grammar-Aware Greybox Fuzzing", 2018

[21] S. Veggalam, S. Rawat, I. Haller, H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming", Proceedings of Computer Security - 21st European Symposium on Research in Computer Security, pp. 581-601, Sep. 2016

данные статического анализа для реализации направленного фаззинга приложений, а также разработка метода фаззинга компиляторов и интерпретаторов с учетом поведения программы.

Глава 2 содержит описание платформы взаимодействия динамических и статических методов анализа, которая позволяет итеративным образом улучшить качество обоих методов анализа.

Основная часть существующих методов статического анализа использует структуры данных, такие как граф патока управления (ГПУ) и граф вызовов (ГВ) функций, построенные на основе исходного или бинарного кода программы. Следовательно, качество проведенного анализа сильно зависит от полноты и точности построенных графов. Извлечение ГПУ и ГВ из бинарного кода связано с определенными сложностями: адрес целевой точки неявных переходов определяется только во время выполнения программы. Поэтому при статическом восстановлении ГПУ некоторые переходы могут отсутствовать. Аналогичная ситуация возникает и при восстановлении ГВ из-за наличия вызовов виртуальных функций. Для решения этих задач применяются методы как динамического, так и статического анализа.

Как было показано ранее, методы динамического анализа пытаются генерировать входные данные, которые увеличивают покрытие кода программы, на основе понимания того, что чем больше кода было выполнено, тем больше вероятность обнаружения ошибок. Однако все известные методы как фаззинга, так и динамического символьного выполнения сталкиваются с проблемой исследования всех путей выполнения программ. Одним из решений данной проблемы является проведение анализа только определенных участков программы. Имея информацию о точках дефектов в программе, можно выполнить локализацию исследуемой части программы. Этого можно достичь с помощью статического построения путей в программе, достигающих полученных точек дефектов, на основе которых будет выполнен динамический анализ.

В разделе 2.1 представляется платформа, обеспечивающая взаимодействие статических и динамических методов анализа. Описывается также разработанный инструмент фаззинга и ее компоненты. Архитектура платформы представлена на рисунке 2.

С помощью реализованной платформы можно повысить точность и результативность обоих методов следующим образом:

- выполнить восстановление неявных вызовов и неявных переходов в ГПУ и ГВ, используя трассу выполнения программы, сохраненную во время фаззинга *(компонент восстановления ГПУ и ГВ- рисунок 2)*.
- выполнить статическое построение путей в программе, приводящих к точкам потенциальных ошибок *(компонент статического построения путей - рисунок 2)*.
- выполнить направленный динамический анализ на основе построенных путей, проводящих к точкам дефектов, для исследования только интересующих участков программ.

Рисунок2. Схема разработанной платформы

Разработанный инструмент фаззинга является одним из основных компонентов платформы. Благодаря своей архитектуре инструмент является легко расширяемым, предоставляя возможность добавления различных компонентов, таких как компонент динамического символьного выполнения, модель мутаций и т.д.

В разделе 2.2 описывается метод восстановления ГПУ и ГВ. С помощью реализации двухстороннего взаимодействия между фаззером и методом статического анализа можно частично решить проблему восстановления вызовов виртуальных функций и неявных переходов без существенных накладных расходов на дополнительный анализ. Разработан и реализован метод, имеющий следующие этапы выполнения. Во время выполнения фаззинга сохраняется трасса выполнения программы, представляющая собой последовательность всех выполненных базовых блоков в виде двух последующих базовых блоков с их начальными и конечными адресами. Далее, используя полученную трассу, выполняется восстановление виртуальных вызовов и неявных переходов в ГВ и ГПУ.

Во время фаззинга выполняется неоднократный запуск компонента восстановления статической информации, который может привести к повышению точности результатов статического анализа. Каждый раз при выполнении нового пути во время фаззинга расширяется трасса выполнения программы, предоставляя возможность увеличивать количество данных, на основе которых выполняется восстановление ГПУ и ГВ.

Раздел 2.3 описывает работу компонента статического построения путей между заданными точками в программе. Данный метод работает на основе представления бинарного кода целевой программы в базе данных в виде таблиц, содержащих инструкции, графы вызова и потока управления и т.д.

Предлагаемый метод состоит из трех основных этапов. На первом этапе строится

подграф ГВ, который содержит все пути между точкой (*точками*) входа в программу и заданными точками. Алгоритм получения данного подграфа состоит в следующем. Выполняется прямой обход ГВ в ширину *(англ. Breadth First Search)* от начальных точек программы. После чего производится обратный обход в ширину от целевых точек к начальным и осуществляется пересечение получившихся множеств. Второй этап представляет собой построение путей в программе с помощью обхода в глубину *(англ. Depth First Search)* ГВ и ГПУ, учитывая результаты первого этапа. Сначала выполняется обход ГВ, после чего поиск путей выполняется на ГПУ каждой функции из ГВ. В результате получается множество путей, представляющих собой последовательность базовых блоков, достигающих заданных точек. Обновлённая информация переходов в ГПУ и ГВ позволяет увеличивать количество и точность путей, построенных компонентом статического построения путей (*Рисунок 2*).

Раздел 2.4 содержит выводы и основные результаты тестирования.

Была разработана и реализована платформа эффективного взаимодействия динамических и статических методов анализа. Платформа позволяет итеративным образом улучшить результаты обоих методов путем:

- обновления графа патока управления и графа вызовов с помощью восстановления неявных переходов и неявных вызовов функций в статической базе данных на основе информации о трассах выполнения программы, полученных во время фаззинга,
- улучшения результатов построения путей в программе, достигающих точек потенциальных дефектов,
- возможности выполнения направленного динамического анализа, используя построенные пути в программе для анализа ее отдельных фрагментов.

В таблице 1 представлены результаты компонента восстановления статической информации на различных приложениях операционных систем (ОС) как *Linux*, так и *Windows*.

Название программы	***Размер программы***	***Количество рёбер до восстановления***	***Количество рёбер после восстановления***
Mutool -1.12.0	35MB	35711	35944
Jasper-1.900.0	1.3MB	2717	2871
Faad2 - 2.7	432KB	2015	2087
FFmpeg– 4.1	18MB	73881	74468
Strings– 2.26.1	28KB	166	184
Latex2rtf – 2.3.8	392 KB	7194	7218
Optipng-0.6.4	228 KB	1802	1867
Tiff2pdf.exe - 2.2.1	1.75 MB	14770	14830

Jpeg2pdf.exe – 2.2.1	1.5 MB	11749	11808
Serenity.exe – 3.2.3	40 KB	538	545

Таблица 1. Результаты восстановления статической информации

Используя точки потенциальных дефектов программы, было выполнено построение путей, достигающих этих точек. Результаты проведенных тестирований (*Таблица 2*) показывают, что после восстановления данных в графе потока управления и графе вызовов выросло количество и точность разработанного компонента статического построения путей.

Название программы	***Количество путей до восстановления***	***Количество путей после восстановления***	***Достигнутые точки дефектов (до/после восстановления)***
JasPer-1.900.0	0	**255**	**0** / **2**
FFmpeg– 4.1	0	**220**	**0** / **2**
Optipng-0.6.4	0	**100**	**0** / **3**
Faad2 - 2.7	5	**250**	**2** / **4**
Strings– 2.26.1	0	**100**	**0** / **1**
Latex2rtf-2.3.8	35	**200**	**1** / **3**
Serenity.exe - 3.2.3	0	**140**	**1** / **3**
Jpeg2pdf.exe-2.2.1	45	**280**	**2** / **4**

Таблица 2. Результаты построения путей в программе

Глава 3 посвящена разработанным методам направленного динамического анализа, которые основаны на ограничении анализируемой части программ, с применением информации об интересующих участках кода, потенциально содержащих дефекты.

Раздел 3.1 описывает метод направленного фаззинга путем частичной инструментации исполняемого кода (*Рисунок 3.б*). Целью данного компонента является локализация той части исполняемого программного кода, которую необходимо анализировать. Имея в наличии точки потенциальных дефектов программы *(полученные методом статического анализа)*, можно построить пути от начальных точек до целевых. На этапе выполнения динамической инструментации программы, инструментируются только базовые блоки, принадлежащие путям, приводящим к точкам потенциальных дефектов. Вследствие этого все мутации над входными данными выполняются только по отношению к тем фрагментам входных данных, которые способствуют открытию новых переходов состояния программы по направлению к точкам потенциальных дефектов.

Кроме частичной инструментации ПО, можно выполнить направленный фаззинг, основанный на идее ограничения количества путей выполнения ПО (*Рисунок 3.в*). В данном случае добавляются вызовы системной функции *exit* в тех базовых блоках

программы, которые не принадлежат путям, приводящим к потенциальным дефектам. Подобные встраивания не должны влиять на поведение целевой программы *(например, привести к неожиданному завершению выполнения)*. Главной разницей, по сравнению с предыдущим методом, является то, что в данном подходе те участки кода, которые не входят в состав интересующих базовых блоков не выполняются, то есть увеличивается скорость выполнения фаззинга.

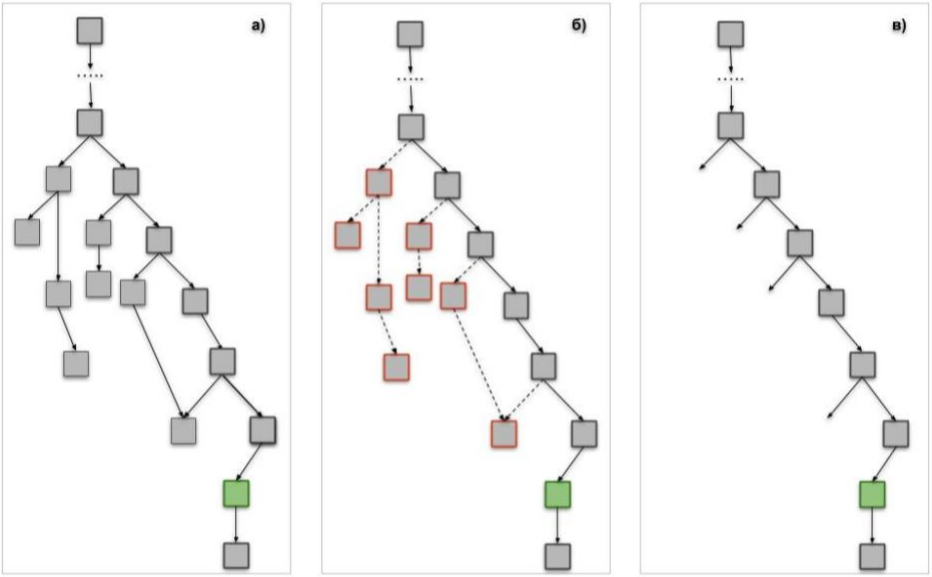

Рисунок 3. Пример ГПУ во время направленного фаззинга. На рисунке а) инструментируются все блоки программы; на рисунке б) блоки, соединенные штриховыми линиями, не инструментируются, но продолжают выполняться; в третьем случае эти блоки не выполняются

В разделе 3.2 рассматривается метод направленного динамического анализа, основанный на интеграции фаззера и динамического символьного выполнения. Имея точки дефектов в программе, можно использовать динамическое символьное выполнение для направленного анализа некоторого подмножества путей программы. Предлагаемый метод запускает динамическое символьное выполнение с учетом адресов базовых блоков условного перехода, приводящих к точкам дефектов. Список адресов соответствующих базовых блоков не задается вручную, а получается в результате работы инструмента статического построения путей *(см. раздел 2.3)*. Рассмотрение конкретных путей *(приводящих к точкам дефектов)* приводит к существенному уменьшению размера трасс ограничений. Это позволяет частично решить проблему экспоненциального роста путей от условных переходов при динамическом символьном выполнении. Работа предлагаемого метода представляет собой непрерывный процесс переключений между фаззером и инструментом динамического символьного выполнения в тех случаях, когда в результате фаззинга не наблюдается рост покрытия кода. Перед каждым запуском динамического

символьного выполнения обновляется список адресов базовых блоков, по которому выполняется фильтрация точек ветвления.

В данном разделе описывается также разработанный и реализованный метод динамического анализа, нацеленный на поиск ошибок типа «*использования памяти после освобождения*» (*UAF*) и «*повторного освобождения памяти*» *(англ. Double Free-DF)*. Метод состоит из двух основных этапов. На первом этапе осуществляется расширение покрытия кода программы на основе символьного выполнения, предоставляя входные данные для воспроизведения каждого открытого пути. На втором этапе производится проверка доступных путей выполнения программы на наличие ошибок UAF и DF, используя входные данные, сгенерированные на первом этапе.

В разделе 3.3 приводятся выводы и основные результаты.

Были разработаны и реализованы методы направленного динамического анализа программ, позволяющие анализировать отдельные участки программы на основе предварительной информации о точках дефектов в программах:

- метод *(direct-fuzz)* частичной инструментации исполняемого кода программы для проведения анализа в определенных фрагментах программы,
- метод *(direct-fast)* инструментации исполняемого кода программы путем добавления инструкций завершения в определенных участках программы для ограничения исполняемой части программы,
- метод применения динамического символьного выполнения с учетом информации о базовых блоках, на основе которых будут построены трассы ограничений.

Результаты тестирований подтверждают эффективность реализованных методов. В таблицах 3 и 4 представлены результаты найденных ошибок. Ограничение на анализ каждой программы составляло 24 часа. В таблице 3 показаны результаты комбинированной работы фаззера и направленного динамического символьного выполнения.

Название программы	***Операционная система***	***Количество найденных падений***	***Достигнутые точки***	***Время обнаружения подения***
bc	Ubuntu 16.04	3	1 / 1	4 ч
colcrt	Ubuntu 16.04	8	1 / 1	20 ч
bzip2recover	Ubuntu 18.04	1	1 / 2	15 ч
dc	Ubuntu 18.04	1	1 / 1	9 ч
blast2	Debian 6.0.10	1	1 / 1	2 ч
faad	Debian 6.0.10	2	1 / 1	21 ч
passwd	Debian 6.0.10	2	1 / 1	0.3 ч
uuenview	Debian 6.0.10	13	1 / 1	~ 1 ч

Таблица 3. Результаты объединения фаззера и динамического символьного выполнения

Для проведения тестирований направленного фаззинга (*Таблица 4*) были рассмотрены программы из набора DARPA Cyber grand challenge (CGC)[22]. Тестовые программы имеют размер от 4 килобайт до 10 мегабайт и реализуют различные функциональные возможности. Для этих программ удалось обнаружить дефекты в точках, найденных в результате статического анализа.

Название программы	***Количество ошибок с Direct-fuzz (раздел 3.1)***	***Количество ошибок с Direct-fast (раздел 3.2)***	***Достигнутые точки***
Personal_Fitness_Manager	2	0	1/2
Humaninterface	1	0	2/2
H20Flowlnc	0	1	1/3
One_Vote	1	0	1/1
Middleout	1	0	1/3
Particle_Simulator	0	1	2/3
Single-Sign-On	2	0	2/2
Stream_vm2	0	0	1/1
ASL6parse	1	0	2/4
Network_Queuing_Simulator	0	0	1/5
CGC_Board	2	1	1/1

Таблица 3. Результаты направленного фаззинга на программах из DARPA CGC

С помощью направленного фаззинга удалось также обнаружить ошибку CVE-2016-8693[23] в программе JasPer 1.900.1, которую не удалось обнаружить обычным фаззером за то же время проведения анализа.

Глава 4 содержит описание алгоритма для динамического анализа компиляторов, интерпретаторов и приложений, обрабатывающих БНФ-структурированные входные данные. Основной задачей является разработка и реализация эффективного алгоритма для анализа этих приложений, который будут независимыми от конкретных спецификаций языков. В рамках данной работы был разработан и реализован метод, позволяющий генерировать тесты на основе БНФ грамматик с учетом покрытия кода программы. Данный метод является расширяемым *(поддерживаются более 120 грамматик)* и предоставляет возможность итеративного улучшения качества сгенерированных тестов.

Раздел 4.1 описывает схему функционирования разработанного алгоритма. Для реализации данного алгоритма был использован набор грамматик *(более 120)* инструмента ANTLR, который использует представление грамматических правил в виде автомата с магазинной памятью. Алгоритм построения программ состоит в обходе *(в глубину)* автомата, начиная с одного из начальных правил. Каждый следующий выбор нетерминального символа выполняется в произвольном порядке. Инструмент

[22] Страница DARPA Cyber Grand Challenge, https://www.darpa.mil/program/cyber-grand-challenge.
[23] Страница CVE-2016-8693, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-8693

предоставляет возможность задавать максимальное значение допустимой глубины всех правил *(нетерминалов)*. В случае достижения максимальной глубины выполняется поиск кратчайшего пути от данного нетерминального символа до одного из соответствующих терминальных символов.

Предложенный метод предоставляет возможность построения произвольного количества синтаксически корректных входных данных для компиляторов и интерпретаторов.

В разделе 4.2 приводится описание разработанного и реализованного алгоритма построения БНФ примеров, учитывая обратную связь с фаззером и ее внедрение как стадии мутации фаззера. Некоторые из существующих систем анализа компиляторов используют наборы входных данных, сгенерированных до начала процесса тестирования. В данном случае единственная возможность изменения тестов во время фаззинга – это применение произвольных модификаций. Чтобы иметь возможность воздействия на построение входных данных в процессе тестирования, был разработан и реализован алгоритм, который применяет весовые значения *(stochastic grammars)* при обходе БНФ грамматик. Во время генерации тестовых примеров сохраняется информация о выполненных переходах, по которым был создан данный пример. Когда во время фаззинга обнаруживается интересный тестовый пример *(увеличивается покрытие кода)*, производится обновление весовых значений переходов автомата. Были внесены изменения и в генераторе тестовых примеров *(см. раздел 4.1)*, который, вместо произвольного выбора грамматических правил, использует весовые значения для измерения вероятности выбора следующего правила.

Реализация предлагаемого алгоритма как новой стадии мутации фаззера обеспечивает его итеративное выполнение и позволяет улучшить точность сгенерированных тестов, направляя процесс их генерации. Эмпирические результаты показали, что применение весовых значений позволяет увеличить покрытие анализируемых программ, по сравнению с рассмотренными методами *(Таблица 4)*.

Раздел 4.3 содержит выводы и основные результаты тестирования разработанного алгоритма.

Были разработаны и реализованы два алгоритма тестирования компиляторов на основе применения БНФ грамматик:

- алгоритм генерации набора тестовых примеров на основе произвольного обхода БНФ грамматик,
- алгоритм генерации тестов с использованием весовых значений, позволяющий изменить порядок обхода БНФ грамматик. Полученные таким образом тестовые примеры обеспечивают большее покрытие кода.

Для оценки эффективности реализованных алгоритмов были проведены тестирования на популярных компиляторах и интерпретаторах, таких как *gcc, g++, python, gfortran, php, luac* и т.д. Результаты проведенных тестирований приведены в таблице 4.

Название программы	***Покрытые базовые блоки (ПББ)с AFL***	***ПББ на основе метода в разделе 4.1***	***ПББ на основе метода в разделе 4.2***	***Количество итераций фаззинга***	***Прирост ББ (%)***

Gcc-7.1	24325	24959	26107	~9800	**+4.6**
G++-7.1	25895	26154	30103	~8400	**+15.1**
Python-2.7	7521	7561	7962	~41000	**+5.3**
Php-v7.1.7	1997	2036	2107	~325000	**+3.5**
Luac-5.3.4	12751	13395	16274	~9700	**+21.5**
Gfortran-7.1	23726	24060	24950	~5300	**+3.7**

Таблица 4. Результаты покрытия кода на основе алгоритма с весовыми значениями

В заключении формулируются основные выводы и предлагаются направления дальнейших исследований задач, рассмотренных в работе.

Основные результаты диссертационной работы:

1. Разработана и реализована платформа динамического анализа, обеспечивающая взаимодействие со статическими методами анализа, позволяющая итеративным образом улучшить результаты обоих методов [2, 3] следующим образом:
   - восстановление неявных переходов и неявных вызовов функций для обновления графа патока управления и графа вызовов функций на основе трасс выполнения динамическим анализом,
   - построение путей в программе, приводящих к точкам ошибок, на основе которых будет выполнен направленный динамический анализ для исследования только определенных участков программы.
2. Разработаны и реализованы методы направленного динамического анализа программ, позволяющие анализировать отдельные участки программы на основе предварительной информации о точках дефектов в программах [1, 2, 3, 4, 5]:
   - метод частичной инструментации исполняемого кода программы для проведения анализа в определенных фрагментах программы,
   - метод инструментации исполняемого кода программы путем добавления инструкций завершения в определенных участках программы для ограничения исполняемой части программы,
   - метод применения динамического символьного выполнения на отдельных путях исполнения ПО с учетом информации о интересующих базовых блоках.
3. Разработан и реализован метод анализа на основе динамического символьного выполнения для обнаружения ошибок использования памяти после освобождения и двойного освобождения памяти [4, 5].
4. Разработан и реализован алгоритм анализа глубоких стадий работы компиляторов, интерпретаторов и трансляторов, направляя процесс генерации входных данных [6, 7]:
   - генерация тестов осуществляется во время проведения анализа. Для этого используется информация о покрытии кода программы, которая позволяет изменить порядок обхода БНФ грамматик для повышения эффективности построенных тестов.

Список опубликованных статей по теме диссертации:

1. A. Gerasimov, S. Vartanov, M. Ermakov, L. Kruglov, D. Kutz, A. Novikov, S. Asryan. Anxiety: A Dynamic Symbolic Execution Framework // 2017 Ivannikov ISPRAS Open Conference, Jan. Moscow, Russia, pages 16-21, 2018.
2. S. Sargsyan, Sh. Kurmangaleev, J. Hakobyan, M. Mehrabyan, S. Asryan, H.Movsisyan. Directed Fuzzing Based on Program Dynamic Instrumentation // Proceedings of the 2019 International Conference on Engineering Technologies and Computer Science: Innovation & Application, IEEE Computer Society, pages 30-33, Los Alamitos, USA, 2019.
3. A. Gerasimov, S. Sargsyan, S. Kurmangaleev, J. Hakobyan, S. Asryan, M. Ermakov. Combining dynamic symbolic execution, code static analysis and fuzzing // Proceedings of the Institute for System Programming, т. 30, № 6, Moscow, Russia, pp. 25-38, 2018
4. С. Асрян, С. Гайсарян, Ш. Курмангалеев, А. Агабалян, Н. Овсепян и С. Саргсян, «Обнаружение ошибок, возникающих при использовании динамической памяти после освобождения» Труды Института системного программирования РАН, т. 30, № 3, Москва, Россия pp. 7-20, 2018
5. S. Asryan, S. Gaissaryan, Sh.Kurmangaleev, A. Aghabalyan, N. Hovsepyan, S. Sargsyan, “Dynamic detection of Use-After-Free bugs” // Programming and Computer Software, vol. 45, № 7, Moscow, Russia, 2019.
6. S. Sargsyan, Sh. Kurmangaleev, M. Mehrabyan, M. Mishechkin, T. Ghukasyan, S. Asryan. Grammar-based Fuzzing // Proceedings of Ivannikov Memorial Workshop, pp. 32-36, Armenia, May 2018.
7. S. Asryan. Feedback driven grammar-based fuzzing // Mathematical Problems of Computer Science, vol. 50, 2018, Yerevan, Armenia, pages 67-75.

Resume

Seryozha Asryan

Dynamic analysis methods for detecting defects in binaries

In the modern world the development of reliable software is still an essential aspect in the field of information technologies (IT). With the increase in size and complexity of software systems it is hard to develop efficient methods of analysis to ensure their quality and security. Even a single defect in software can cause serious issues. For instance, because of *HeartBleed* defect in OpenSSL library, millions of users’ personal information was leaked. Initially defect detection was made manually either looking through a code or examining compiler warnings. However, nowadays various automatic tools are used. There are two main methods for software testing: static and dynamic analyses. The static analysis reasons about program without its execution. It uses intermediate representations of programs, like abstract syntax tree, control flow graph, call graph etc., to perform program analysis and find defects. Despite the scalability of static analysis methods, they have high rate of false positives.

Dynamic analysis performs program testing in its actual running environment during its execution, hence, it reasons only about paths being executed. This approach eliminates false positives and produces inputs that actually reproduce detected defects. This method includes fuzzing, dynamic symbolic execution, debugging and profiling programs, compiler-based instrumentation components (*e.g. sanitizers*) etc.

Fuzzing is one of the most popular and efficient methods of dynamic analysis. Its main workflow consists of running the target program with malformed *(random)* inputs and tracking its behavior. During fuzzing, inputs are generated using different mutation algorithms. More advanced fuzzing tools use code coverage information captured via static or dynamic instrumentation of a target binary. This information is used to determine which mutated inputs to keep for further fuzzing.

Another well-known method is dynamic symbolic execution. It is based on the replacement of the program inputs by symbolic values. During dynamic symbolic execution the predicates inside branch statements, that depend on symbolic values, are collected. The collection of predicates is used as constraints of inputs for current execution path. After that, a constraint solving techniques (*SMT solvers*) are used to solve gathered constraints in order to generate new inputs that would lead program execution towards alternative branches.

The dynamic analysis methods try to examine all feasible paths of the program, which in general is not solvable due to path explosion problem. The efficiency of software analysis can be significantly increased by combining dynamic and static analyses methods. We can detect defects in programs *(such as use after free, buffer overflow etc.)* with static analysis. Using found defects fuzzer *(or dynamic symbolic execution)* can generate inputs that lead execution towards target points and reproduce defects. For that purpose, paths between program entry points and target ones are constructed. Each path represents a sequence of basic blocks that reaches defect points. Afterwards, the fitness function is defined and genetic algorithm is used to evaluate inputs comparing programs execution trace with constructed paths. As a result of such comparison, which is based on different metrics, the most appropriate input for further processing is chosen.

Another important problem is the analysis of programs, such as compilers and interpreters. Defects in these programs can cause a generation of incorrect binaries. To pass the initial stages of compilation *(lexical and syntactic analyses)* and analyze its deep stages *(semantic analysis, code generation etc.)* we need to ensure structural completeness of inputs. Existing tools generate inputs based on input grammar's BNF representation. However, input generation model in those tools is closely connected to target language specifications, hence, they are limited in terms of extensibility to analyze new programs.

A work is dedicated to design dynamic analysis methods for finding defects in binary programs. Methods must provide interaction with static analysis to allow iterative improvement of both methods and be able to perform efficient analysis of programs that process structured inputs.

Main results of the work

- A platform for dynamic analysis that provides interaction with static analysis methods, which allows to iteratively improve efficiency of both analyses.
- The methods of directed dynamic analysis that allow to analyze distinguished paths of the program on the basis of prior information on defects points in programs.
- A method of detecting memory corruption defects *(use after free, double free)* based on dynamic symbolic execution.
- An effective algorithm of analyzing programs, that process structured inputs, such as compilers and interpreters. The method uses code coverage information to guide input

generation process.

Ամփոփում

Ասրյան Սերյոժա Արմենի

ԿԱՏԱՐՎՈՂ ԿՈԴԻՄ ՍԽԱԼՆԵՐԻ ՀԱՅՏՆԱԲԵՐՄԱՆ ԴԻՆԱՄԻԿ ՎԵՐԼՈՒԾՈՒԹՅԱՆ ՄԵԹՈԴՆԵՐ

Տեղեկատվական տեխնոլոգիաների (ՏՏ) ոլորտում կարևորագույն խնդիրներից մեկն է հուսալի ծրագրային ապահովման մշակումը: Ծրագրային համակարգերի չափի և բարդության աճին զուգընթաց դժվարանում է նրանց որակի և ապահովության վերլուծությունն իրականացնող արդյունավետ մեթոդների մշակումը: Նույնիսկ մեկ սխալի առկայությունը կոդի մեջ կարող է ծանր հետևանքներ ունենալ: Ինչպես օրինակ՝ *OpenSSL* գրադարանում եղած *HeartBleed* կոչվող ծրագրային սխալը, որը հանգեցրեց կոդավորված տվյալների արտահոսքի միլիոնավոր սերվերների վրա: Ներկայումս կիրառվում են տարբեր մեթոդներ, որոնք թույլ են տալիս մասամբ կամ ամբողջությամբ ավտոմատացնել վերլուծության պրոցեսը: Առանձնացվում են վերլուծության երկու հիմնական մոտեցումներ՝ ստատիկ ու դինամիկ: Ստատիկ վերլուծության ժամանակ իրականացվում է ծրագրի հետազոտություն առանց նրա կատարման: Ստատիկ վերլուծության գործիքները օգտագործում են ծրագրերի միջանկյալ ներկայացումը *(աբստրակտ շարահյուսական ծառ, դեկավարման հոսքի գրաֆ, կանչի գրաֆ և այլն)* և իրականացնում դրանց մշակումը: Սխալների որոնումը տեղի է ունենում միջանկյալ ներկայացման հետազոտման արդյունքում: Որպես կանոն, ստատիկ վերլուծություն կատարող մեթոդները հեշտ ընդլայնվում են, բայց ունեն սխալանքի բարձր ցուցանիշ:

Դինամիկ վերլուծության դեպքում ծրագրի հետազոտությունը տեղի է ունենում նրա կատարման միջավայրում՝ դիտարկելով ծրագրի միայն այն հատվածները, որոնք կատարվել են: Դինամիկ վերլուծության դեպքում օգտագործվում են իրական մուտքային տվյալներ, ինչը թույլ է տալիս գրեթե ամբողջությամբ վերացնել սխալանքը և ստանալ մուտքային տվյալներ, որոնք վերարտադրում են հայտնաբերված սխալները: Այս մոտեցման հիմնական ուղղություններից է ֆազինգը, դինամիկ սիմվոլիկ կատարումը, բինար թարգմանության մեթոդները, կոմպիլյատորային միջոցների օգտագործումը *(sanitizers)* և այլն:

Ֆազինգը համարվում է դինամիկ վերլուծության ամենատարածված մեթոդներից մեկը: Այս մեթոդի հիմքում ընկած է կամայական մուտքային տվյալների միջոցով ծրագրի կատարման վարքին հետևելը: Մուտքային տվյալները ստեղծվում են մուտացիայի տարբեր ալգորիթմեր կիրառելու միջոցով: Ֆազինգի որոշ արդի գործիքներ հաշվի են առնում կոդի ծածկույթը՝ կատարելով կոդի ստատիկ կամ դինամիկ ինստրումենտացիա, որը թույլ է տալիս առանձնացնել այն մուտքային տվյալները, որոնց միջոցով դիտարկվել են կատարման նոր ճանապարհներ:

Դինամիկ անալիզի մեկ այլ մեթոդ է դինամիկ սիմվոլիկ կատարումը: Այս դեպքում մուտքային տվյալների արժեքները փոխարինվում են սիմվոլիկ փոփոխականներով: Դիտարկելով սիմվոլիկ փոփոխականները և ճյուղավորման օպերատորներին պատկանող պրեդիկատները՝ մուտքային տվյալների վրա

ստեղծվում են սահմանափակումներ: Այդ սահմանափակումներից յուրաքանչյուրը նկարագրում է պայման, որը բավարարելու դեպքում կդիտարկվի ծրագրի կատարման կոնկրետ ճանապարհ: *SMT solver*-ի օգտագործումը թույլ է տալիս լուծել ստացված սահմանափակումները և ստանալ մուտքային տվյալներ ծրագրի կատարման նոր ճանապարհներ դիտարկելու համար:

Դինամիկ անալիզի մեթոդները փորձում են հետազոտել ծրագրի բոլոր հնարավոր ճանապարհները, որը գործնականում անիրագործելի է ճյուղավորման օպերատորների քանակից կախված ճանապարհների քանակի էքսպոնենցիալ աճի հետևանքով: Ծրագրային ապահովման վերլուծության արդյունավետությունը կարելի է զգալիորեն բարձրացնել ստատիկ և դինամիկ մեթոդնեի միավորման արդյունքում: Ստատիկ վերլուծության միջոցով հնարավոր է հայտնաբերել ծրագրերի հավանական սխալների կետերը և կիրառել ֆազերը *(կամ դինամիկ սիմվոլիկ կատարման մեթոդը)* կառուցելու մուտքային տվյալներ այդ կետերին հասնելու՝ հայտնաբերված սխալները վերարտադրելու համար:

Մեկ այլ կարևորագույն խնդիր է կոմպիլյատորների և ինտերպրետատորների վերլուծությունը: Կոմպիլյատորներում եղած սխալները կարող են հանգեցնել սխալ ծրագրերի ստեղծման: Կոմպիլյացիայի խորը փուլերը հետազոտելու համար *(սեմանտիկ վերլուծություն, կոդի գեներացիա և այլն)* անհրաժեշտ է ապահովել մուտքային տվյալների կառուցվածքային ամբողջականությունը: Ներկայումս գոյություն ունեցող գործիքները իրականացնում են տվյալների գեներացիա *BNF (Backus–Naur form)* քերականությունների միջոցով: Սակայն տվյալների գեներացիայի տրամաբանության և ծրագրավորման լեզուների սպեցիֆիկացիաների միջև սերտ կապի հետևանքով այդ գործիքները սահմանափակ են նոր լեզուներին համապատասխանող մուտքային տվյալներ ստեղծելու համար:

Ատենախոսության նպատակն է մշակել դինամիկ վերլուծության մեթոդներ և ծրագրային միջոցներ՝ կատարվող կոդում սխալներ գտնելու համար: Մեթոդները պետք է ապահովեն համագործակցություն ստատիկ վերլուծության հետ, ինչպես նաև ունենան հնարավորություն հետազոտելու ստրուկտուրիզացված տվյալներ մշակող ծրագրեր:

Ատենախոսության հիմնական արդյունքներն են՝

- Մշակվել և իրականացվել է դինամիկ վերլուծության համակարգ, որը ապահովում է համագործակցություն ստատիկ մեթոդի հետ՝ իտերատիվ կերպով նրանց արդյունքների լավացման համար:
- Մշակվել և իրականացվել են ուղղորդված դինամիկ վերլուծության մեթոդներ, որը թույլ է տալիս իրականացնել ծրագրի առանձնացված հատվածների հետազոտություն:
- Մշակվել և իրականացվել է դինամիկ սիմվոլիկ կատարման միջոցով դինամիկ հիշողության սխալ օգտագործման դեպքերի հայտաբերման մեթոդ:
- Մշակվել և իրականացվել է դինամիկ վերլուծության ալգորիթմ ստրուկտուրիզացված ծրագրերի հետազոտման նպատակով: Ալգորիթմը օգտագործում է կոդի ծածկույթը տվյալների գեներացիայի պրոցեսն ուղղորդելու համար: