Санкт-Петербургский государственный университет

Кафедра системного программирования

Группа 21.М07-мм

Устранение ложных срабатываний статических анализаторов кода символьным исполнением для платформы .NET

Седлярский Михаил Андреевич

Отчёт по производственной практике

Научный руководитель:
доцент кафедры системного программирования, к.ф.-м.н., Д. А. Мордвинов

Санкт-Петербург
2023

Оглавление

Введение

С каждым днём программное обеспечение становится всё сложнее и сложнее. Тестов, написанных людьми, уже недостаточно, чтобы исчерпывающе проверять программы на корректность. Пропущенные программные ошибки в различных сферах человеческой деятельности могут приводить к большим потерям: начиная от задержек во времени и заканчивая человеческими жизнями. В 2002 году Грегори Тассей оценивал годовой ущерб экономике США от ошибок программного обеспечения в 59.5 миллиардов долларов США [24].

Среди методов поиска ошибок можно выделить статический анализ исходного кода программы, который позволяет найти ошибки на редко выполняющихся путях, для которых сложно составить тесты либо выявить их динамическим анализом. Особенностью большинства методов статического анализа является то, что они исследуют не исходную программу, а её аппроксимацию сверху. Это означает то, что помимо всех возможных состояний программы анализируются и те состояния, которые на самом деле недостижимы. Отсюда появляются ложноположительные срабатывания статического анализа.

Также можно выделить другой метод анализа программ – символьное исполнение. Это метод моделирования работы программы с использованием символьных значений вместо конкретных. Во время символьного исполнения, встретив оператор ветвления, анализатор добавит условие (ограничение) в список так называемых ограничений пути и продолжит анализ в обеих ветвях. Затем используются SMT решатели, которые по заданным условиям находят входные данные, приводящие программу в интересующие ветви. Такими ветвями могут быть, например, те, в которых выбрасываются исключения.

Одной из главных проблем символьного исполнения является, так называемый, комбинаторный взрыв путей. Количество путей программы растёт экспоненциально от её размера. Более того, если программа содержит циклы с заранее неизвестным количеством итераций или рекурсию, то количество возможных путей исполнения будет бесконеч-

ным.

Можно заметить, что т.к. символьное исполнение исследует только существующие пути выполнения программы, то оно лишено возможности выносить ложноположительные вердикты.

Идея данной работы заключается в том, что можно подкрепить статический анализ кода символьным исполнением. Т.е. перепроверять результаты статического анализа в символьной виртуальной машине. Тем самым мы можем сгладить недостатки каждого из подходов: подтверждаем верноположительные срабатывания статического анализа и сокращаем пространство поиска символьного анализа. Такой подход поможет ранжировать результаты статических анализаторов кода, опуская возможные ложноположительные срабатывания в отчёте анализатора.

Целью работы является разработка и реализация приложения для платформы .NET, которое будет перепроверять срабатывания статических анализаторов кода символьным исполнением и дополнять отчёт стектрейсами и новыми срабатываниями.

1. Постановка задачи

Целью работы является разработка и реализация приложения для платформы .NET, подтверждающее срабатывания статических анализаторов кода символьным исполнением. Для выполнения цели были поставлены следующие задачи:

1. провести сравнительный анализ статических анализаторов кода для платформы .NET;
2. запустить статические анализаторы на настоящих проектах. На основании полученных результатов выбрать анализатор кода, с которым будут проводиться дальнейшие работы;
3. разработать и реализовать алгоритм преобразования результатов статического анализатора кода в цели для движка символьного исполнения V#;
4. разработать и реализовать способ передачи целей в движок символьного исполнения V#;
5. разработать и реализовать алгоритм ранжирования результатов статического анализа на основе результатов символьного исполнения;
6. провести эксперименты и оценить качество полученного инструмента.

2. Обзор

2.1. Статический анализ кода

Статический анализатор кода работает с помощью алгоритмов, которые не запускают исследуемую программу. А следовательно, не подготавливают никаких входных данных. Результатом работы анализатора является список мест в исходном коде, в которых были найдены ошибки. Для каждой ошибки также указывается её вид, некоторое пояснение и, возможно, дополнительная информация: потенциальный стек вызова или входные данные, вызывающие поломку программы. Подробный анализ существующих подходов статического анализа приведён в [4].

Создание анализаторов для языка C# заметно упростилось в 2015 году после того, как компания Microsoft выложила в открытый доступ исходный код проекта Roslyn [37]. Данный проект представляет из себя промышленный компилятор языка C# и интерфейсы работы с АСД.

На данный момент выделим следующие известные анализаторы для платформы .NET: Roslyn analysers [19], ReSharper [18], PVS-Studio [16], InferSharp [9], sonar-dotnet [34], CHECKMARX SAST [5].

2.2. Символьное выполнение

Символьное выполнение [22] — распространённый метод анализа программ, введенный в середине 70-х годов для доказательства свойств программного обеспечения. Данный подход заключается в том, что программа выполняется не на конкретных входных данных, а на так называемых символьных переменных. Для каждой ветви программы поддерживается условие пути (path condition) — формула логики первого порядка, содержащая символьные переменные. Если можно подобрать конкретные значения переменных таким образом, что формула выполняется, то тогда путь считается достижимым. Для получения такого набора конкретных значений, удовлетворяющих условию пути символьные виртуальные машины используют SMT-решатели [7]. Также SMT-

решатели могут доказать, что формула невыполнима. Другими словами, символьное выполнение может сгенерировать входные данные, которые приводят выполнение в конкретную точку программы или же доказать, что искомая точка недостижима.

На данный момент существует не так много символьных виртуальных машин для платформы .NET. Можно выделить V# [26] и Pex [17]. Второе решение не поддерживает .net core и практически не развивается. Поэтому вся дальнейшая работа будет проводиться с символьной виртуальной машиной V# (VSharp).

2.3. Подкрепление символьным исполнением

Уже известны работы, в которых совмещаются статический анализ и символьное выполнение. Например, в статье [6] описывается комбинирование символьного исполнения, статического анализа и фаззинга для увеличения тестового покрытия приложений. А в работе [25] авторы проводят статический анализ потока данных android-приложений и удаляют ложноположительные срабатывания символьным движком TASMAN.

Текущая работа преследует схожую цель. А именно: реализация приложения, подтверждающего верноположительные срабатывания статических анализаторов кода с помощью символьной виртуальной машины V#.

3. Метод

3.1. Общая схема решения

Разрабатываемое приложение для удобства носит название Veritas. На рисунке 1 изображена общая схема потока данных в системе статический анализатор — Veritas — V#.

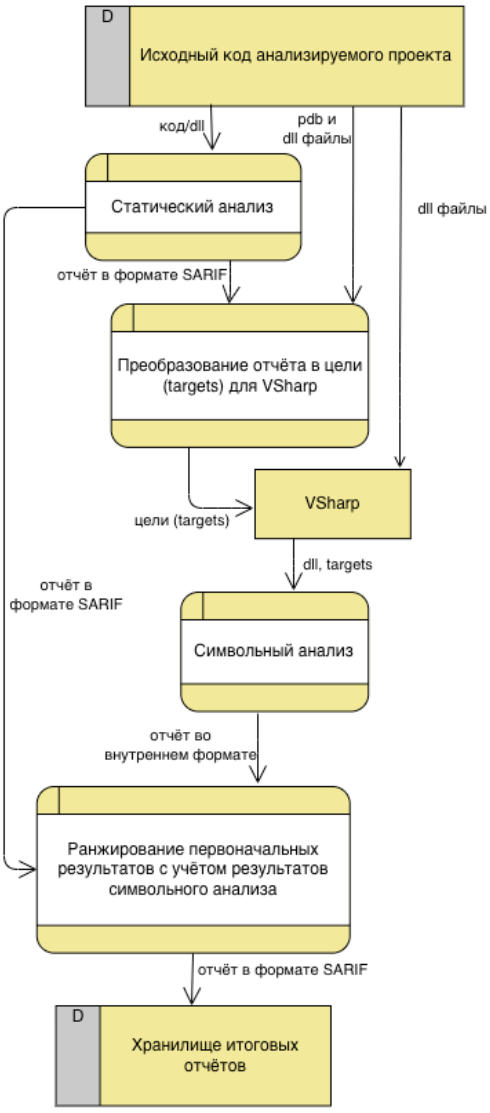

Рис. 1: Общая DFD-схема

Принцип действия следующий:

1. Пользователь с помощью статического анализатора кода получает отчёт об исследуемом проекте. Стандартом индустрии представ-

ления отчётов является json-based формат SARIF (Static Analysis Results Interchange Format) [21].

2. Полученный отчёт и результаты сборки проекта передаются в Veritas.

3. Veritas выбирает из отчёта ошибки, которые можно проверить символьным исполнением, и преобразует в цели (targets) для V#.

4. Затем Veritas в рамках своего процесса запускает символьный анализ. VSharp используется как библиотечная зависимость.

5. Полученные результаты используются для ранжирования и обогащения исходного отчёта статического анализатора: удаляются (или понижаются в важности) ложноположительные срабатывания, добавляются примеры входных данных для верноположительных срабатываний, добавляются сигналы об ошибках, которые не были найдены стат. анализатором (ложноотрицательные срабатывания).

6. На выходе получается обогащённый отчёт, который можно использовать в дальнейших CI процессах.

Предлагаемое решение по своей сути является обёрткой вокруг символьной машины VSharp. На схеме не указан какой-либо конкретный статический анализатор т.к. Veritas может работать с целым перечнем анализаторов, поддерживающих выходной формат SARIF.

На рисунке 2 изображена схема развёртывания. Veritas использует VSharp как библиотеку наряду с библиотеками Mono.Cecil для работы с pdb файлами и sarif-sdk для работы с отчётами статических анализаторов.

3.2. Выбор статического анализатора

Т.к. формат SARIF де-факто является стандартом индустрии, Veritas теоретически может работать с разными анализаторами. Но для после-

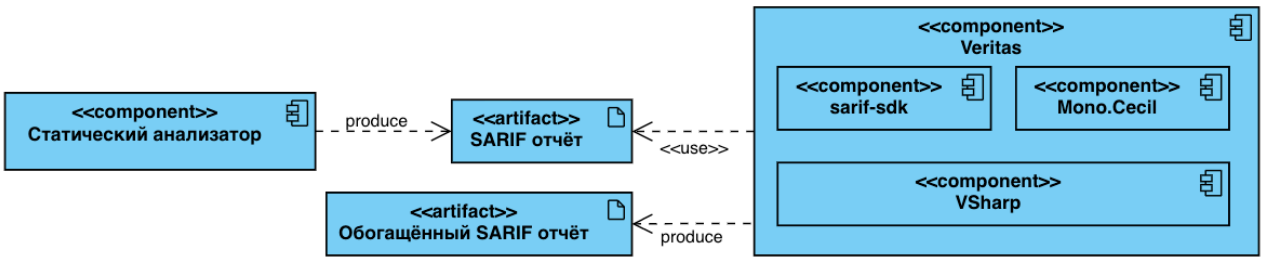

Рис. 2: Схема развёртывания

дующих реализации и эксперимента имеет смысл выбрать такой анализатор, который:

1. Бесплатный. Анализатор можно использовать бесплатно в исследовательских целях.

2. Поддерживает .net core.

3. Находит в проектах значительное количество ошибок, которые можно проверить символьным исполнением. Например, такие как: разыменование нулевого указателя и выход за границы массива.

4. Работает на разных платформах: windows, linux, macos.

5. Поддерживается авторами. Разработка Veritas будет идти быстрее, если авторы анализатора будут помогать консультациями и оперативно исправлять собственные ошибки.

6. Поддерживает SARIF v2. Новый формат более удобен в использовании в то время, как формат первой версии постепенно лишается поддержки в библиотеках.

В таблице 1 сведены данные о самых распространённых анализаторах для платформы .NET.

На момент написания работы поддержку предоставляют только open-source решения и PVS-studio. Также стоит указать, что Sonnar Scanner

это не один анализатор а совокупность Roslyn analyzers и sonar-dotnet анализатора.

Исходя из вышеперечисленных ограничений для дальнейшего сравнения были выбраны InferSharp, PVS-Studio и Sonnar Scanner (Roslyn analyzers + sonar-dotnet).

Название	Поддерживаемые платформы			Поддержка SARIF	Open source
	win	linux	macos		
Roslyn analysers	+	+	+	+	+
ReSharper	+	+	+	+	- (есть бесплатные лицензии)
PVS-Studio	+	+	+	+ (через plog-converter)	- (есть бесплатные лицензии)
InferSharp	+ (через wsl)	+	± (работает через docker, но теоретически можно собрать нативно)	+	+
Sonar Scanner for .NET + sonar-dotnet	+	+	+	+	+
CHECKMARX SAST	web решение				- (можно получить демо версию)

Таблица 1: Сводная информация о статических анализаторах .NET

Для дальнейшего сравнения была подготовлена тестовая база из 13 open-source и 4 студенческих проектов. Open-source проекты отбирались на платформе GitHub по количеству ”звёзд”. Стоит отметить, что в тестовой базе все проекты поддерживают .net core платформу. В таблицах 2, 3, 4 видно, что лидерами по количеству срабатываний для большинства проектов стали Roslyn analyzers. Но большинство этих проверок относятся к стилистике кода и их невозможно проверить символьным исполнением. Такое же замечание применимо и к анализатору sonar-dotnet. В процесс тестирования Infer# от Microsoft была обнаружена неисправность анализатора. Анализатор некорректно загружал сборки анализируемого приложения. По этому поводу было сделано обращение к авторам проекта и совместными усилиями ошибка устранена [10]. В последствии было выяснено, что Infer# в большинстве случаев

находит только один тип ошибок — разыменование нулевого указателя. К тому же, согласно выводу анализатора, во время анализа пропускалось до 60% инструкций. При работе с PVS-Studio также была обнаружена небольшая неисправность (неверно определялся .net sdk). Было создано обращение в поддержку анализатора и данная поломка была быстро устранена. Анализатор показывает умеренные результаты и довольно разнообразный перечень ошибок. Таким образом, наиболее удобным и репрезентативным анализатором для текущего проекта является PVS-Studio. Результаты запусков и вспомогательное ПО можно найти здесь [1].

Проект	NULLPTR_ DEREFERENCE	PULSE_ RESOURCE_LEAK	THREAD_ SAFETY_VIOLATION	STACK_VARIABLE _ADDRESS_ESCAPE
efcore	107	2	8	0
litedb	1	7	6	0
moq4	0	1	0	0
NLog	30	75	44	0
nunit	26	192	7	0
xunit	0	0	0	0
btcpayserver	4	6	4	0
AutoMapper	0	0	0	0
spbu-homeworks-1	0	1	4	0
parallel-programming-1-sync	0	0	0	0
parallel-programming-1-thread-pool	0	0	0	0
parallel-programming-3-thread-pool	0	0	0	0
BenchmarkDotNet	2	11	0	0
ILSpy (ILSpy.XPlat.slnf)	48	4	0	2
OpenRA	34	49	1	0
Newtonsoft.Json	14	271	5	0
RestSharp	0	53	0	0

Таблица 2: Результаты запуска Infer#

3.3. Внутренняя структура Veritas

Фильтрация результатов статического анализатора. Как уже упоминалось ранее, далеко не все срабатывания статических анализаторов можно проверить с помощью символьного исполнения. Например, замечания по поводу стилистики кода. Для текущей работы были выбраны три вида замечаний PVS-Studio:

1. V3080 (Possible null dereference) – разыменование нулевого указателя. В случае такой ошибки Veritas будет ожидать исключение типа NullReferenceException.

Проект	roslyn	sonar-dotnet	прочее
efcore	264	2893	487
litedb	0	470	9
moq4	214	441	5
NLog	17	266	12
nunit	218	856	4
xunit	292	1092	0
btcpayserver	267	1062	1
AutoMapper	88	167	0
spbu-homeworks-1	33	105	0
parallel-programming-1-sync	0	2	0
parallel-programming-1-thread-pool	11	3	6
parallel-programming-3-thread-pool	8	10	0
BenchmarkDotNet	18	19	0
ILSpy (ILSpy.XPlat.slnf)	873	1829	0
OpenRA	0	2990	0
Newtonsoft.Json	1478	740	0
RestSharp	28	47	9

Таблица 3: Результаты запуска sonar-scanner-msbuild (roslyn + sonar-dotnet)

Проект	V3022 (Expression <exp> is always <value>)	V3080 (Possible null dereference)	V3146 (Possible null dereference. A method can return default null value)	V3106 (Possibly index is out of bound)	Всего
efcore	анализ решения съедает все ресурсы на сервере				
litedb	15	19	1	0	110
moq4	4	2	0	0	61
NLog	34	9	0	1	163
nunit	28	8	0	0	203
xunit	19	11	0	0	127
btcpayserver	37	99	13	0	374
AutoMapper	2	1	0	0	38
spbu-homeworks-1	0	0	0	2	5
parallel-programming-1-sync	0	0	0	0	0
parallel-programming-1-thread-pool	0	0	0	0	0
parallel-programming-3-thread-pool	0	0	0	0	2
BenchmarkDotNet	14	0	0	0	82
ILSpy (ILSpy.XPlat.slnf)	98	128	0	17	792
OpenRA	13	39	0	2	342
Newtonsoft.Json	30	10	0	34	265
RestSharp	0	1	0	0	16

Таблица 4: Результаты запуска PVS-Studio

2. V3146 (Possible null dereference. A method can return default null value) – другая вариация разыменования нулевого указателя, которая так же отображается на исключение типа NullReferenceException.

3. V3106 (Possibly index is out of bound) – выход индекса за границы коллекции. В этом случае Veritas ожидает исключение типа IndexOutOfRangeException.

Таким образом, в дальнейшем будут рассматриваться только результаты статического анализа перечисленных выше видов.

Построение целей для символьного исполнения. Цели для символьного исполнения строятся по следующим принципам:

1. Целями для символьного исполнения называются адреса инструкций в dll сборках, к исполнению которых пытается приблизиться движок символьного исполнения.

2. Каждому срабатыванию (результату) статического анализатора сопоставлено место в исходном коде: имя файла исходного кода, номер строки и номер столбца.

3. Одним из видов артефактов сборки .net приложений являются pdb файлы. Помимо прочей метаинформации, в них сопоставлены некоторые места в исходном коде с номерами инструкций в сборках (dll файлах). Такие места называются точками следования.

4. В большинстве случаев, сопоставленная результату локация в коде является точкой следования. Следовательно, мы можем для большинства результатов статического анализатора найти нужный адрес инструкции и делегировать его символьному движку.

5. Полученные точки следования не являются непосредственно теми инструкциями, которые вызывают искомую ошибку. Правильнее рассматривать их как приблизительные ориентиры, которые помогают символьному исполнению сократить пространство поиска.

Исходя из изложенных выше принципов, построение целей в Veritas делается в три этапа:

1. Поочерёдно загружаются точки следования из всех сборок исследуемого проекта, у которых есть соответствующие pdb файлы. Точки следования хранятся в хэш-таблице, где ключом выступает имя файла с исходным кодом, а значением список с точками следования.

2. Затем просматриваются заранее отфильтрованные результаты из отчёта статического анализатора. Если тип результата можно проверить с помощью символьного исполнения, то для локации результата из индекса достаются соответствующие точки следования.

3. Если для результата нашлось несколько точек следования, то для каждой из них вычисляется адрес базового блока, полученные адресса дедуплицируются. В последствии они будут переданы символьному исполнению в качестве целей.

4. Если для результата не нашлось ни одной точки следования, то он сохраняется в отдельный список и проверяется на корректность локализации: возможно, что номер строки, проставленный статическим анализатором, ошибочен и указывает на пустую строку в коде, комментарий или любую другую семантически пустую строку.

Передача целей в символьный движок и интерпретация результатов. Идея интеграции с V# состоит в следующих шагах:

1. Передать полученные цели в символьный движок и начать исполнение на множестве методов, которым принадлежат найденные точки следования.

2. Во время исполнения сохранять информацию о найденных исключениях, а именно: тип исключения и его стектрейс

3. Затем, сопоставить найденные стектрейсы исключений с результатами статического анализатора. Будем считать, что если для результата R существует такое исключение E, что тип исключения соответствует типу результата и локация (совокупность имени файла, номера строки и номера столбца) R присутствует в стектрейсе E, то результат R считается подтверждённым.

В текущей версии V# (0.0.1) существует несколько режимов исследования программы. В том числе targeted-режим, который умеет генерировать цели и направлять исполнение к ним, чтобы покрыть интересующие места программы. В ходе данной работы targeted-режим был расширен таким образом, чтобы его можно было инициализировать начальными целями и всё время исполнения он стремился только к ним.

Также было добавлено сохранение информации о выброшенных исключениях. Т.к. стектрейсы в символьной машине представлены списком инструкций (адресов в сборках), был реализован алгоритм декодирования таких списков в набор локаций.

Ранжирование результатов. Символьное исполнение ограничивается по времени т.к. может зависнуть из-за комбинаторного взрыва путей. Поэтому невозможно точно сказать: результат статического анализатора не подтверждён потому, что недостижим или потому, что отсечка по времени оказалось недостаточно большой. Поэтому вместо удаления ложноположительных срабатываний статического анализатора предлагается их ранжировать по следующей дискретной шкале:

1. Результаты подтверждённые символьным исполнением. Их можно обогатить стектрейсами и входными данными, приводящими к ошибке.

2. Неподтверждённые результаты (нет ни доказательств, ни опровержений).

3. Результаты с неправильной локализацией. Это те результаты, которые остались без целей и указывают на семантически пустые

строки в коде (см. рисунок 3).

4. Результаты с исключениями, которые были найдены V#, но не отражены ни на один из результатов статического анализатора.

4. Эксперимент

Проверим насколько полно Veritas может покрыть целями (targets) ошибки из отчётов PVS-Studio, долю подтверждённых срабатываний и долю ошибок, указывающих на некорректную локацию в коде.

4.1. Условия эксперимента

Veritas будет запускаться на следующих 14 проектах:

1. litedb [11]
2. NLog [13]
3. btcpayserver [3]
4. moq4 [29]
5. nunit [30]
6. xunit [36]
7. AutoMapper [2]
8. spbu-homeworks-1 [35]
9. ILSpy (ILSpy.XPlat.slnf) [8]
10. OpenRA [15]
11. Newtonsoft.Json [14]
12. parallel-programming-1-sync [31]
13. parallel-programming-1-thread-pool [32]
14. parallel-programming-3-thread-pool [33]

Для каждого из вышеперечисленных проектов был получен отчёт PVS-Studio (v.7.21.64848.1157) в формате SARIF.

Затем, Veritas для каждого проекта фильтрует срабатывания статического анализатора, формирует цели и передаёт их в V#.

Символьный движок в каждом запуске ограничивается по времени из расчёта 60 секунд на каждую цель. После завершения символьного анализа Veritas сопоставляет найденные исключения с срабатываниями PVS-Studio.

Срабатывания, для которых не найдены цели, дополнительно анализируются на корректность локализации. Если строка исходного кода, на которую указывает срабатывание, пустая, или состоит из пробельных символов или скобок, то такое срабатывание считается ошибочным.

Затем подсчитываются доли подтверждённых срабатываний и срабатываний с неправильной локализацией от общего числа срабатываний поддерживаемых типов (V3080, V3146, V3106).

4.2. Исследовательские вопросы

- Q1: узнать насколько полно Veritas покрывает целями срабатывания из отчётов PVS-Studio.
- Q2: узнать долю срабатываний статического анализатора с неправильной локализацией.
- Q3: узнать долю срабатываний, которые удаётся подтвердить символьным исполнением от числа срабатываний с целями.
- Q4: оценить количество новых исключений, которые обнаружит символьное исполнение.

4.3. Метрики

Для Q1 подходящей метрикой будет доля срабатываний с хотя бы одной целью от общего числа поддерживаемых срабатываний. В осталь-

ных исследовательских вопросах уже содержится определение подходящей метрики.

4.4. Результаты

Рассмотрим таблицу 5. Из неё видно, что отчёты 3 из 14 проектов не содержат интересных для Veritas срабатываний и поэтому последующие метрики для них бессмысленны. По остальным проектам видно, что доля покрытия срабатываний целями достигает 97,83%. При этом доля срабатываний с неправильной локализацией в среднем достаточно велико и может достигать 100%. Доля срабатываний, которые удалось подтвердить символьным исполнением невелика и не превышает 34,41%. В 8 из 11 проектах символьное исполнение обнаружило новые исключения.

Проект	Кол-во поддерживаемых срабатываний	Доля срабатываний с целями, %	Доля срабатываний с неправильной локализацией, %	Доля подтверждённых срабатываний с таргетами, %	Кол-во найденных исключений
litedb	20	60,00%	40,00%	33,33%	52
NLog	10	70,00%	30,00%	0,00%	4
btcpayserver	114	58,77%	27,19%	5,97%	28
moq4	2	0,00%	0,00%	-	0
nunit	8	12,50%	75,00%	0,00%	0
xunit	11	81,82%	0,00%	0,00%	1
AutoMapper	7	42,86%	57,14%	0,00%	5
spbu-homeworks-1	1	0,00%	100,00%	-	0
ILSpy (ILSpy.XPlat.slnf)	146	47,26%	41,10%	2,90%	123
OpenRA	97	95,88%	0,00%	34,41%	289
Newtonsoft.Json	46	97,83%	0,00%	0,00%	10
parallel-programming-1-sync	0				
parallel-programming-1-thread-pool	0				
parallel-programming-3-thread-pool	0				

Таблица 5: Результаты запуска Veritas

4.5. Обсуждение результатов

Чем можно объяснить, что в среднем 48,46% результатов PVS-Studio не удалось отобразить на IL инструкцию? Неужели pdb информация настолько неполная? Отнюдь. Дополнительный анализ показал, что в среднем 33,67% срабатываний имеют некорректную привязку к местоположению в коде. Например, на рисунке 3 видно, что ошибки вида null dereference локализованы на строки с комментариями. Очевидно,

что таким строкам не сопоставлены никакие точки следования в pdb файлах. Стоит отметить, что иногда доли срабатываний с целями и срабатываний с неправильной локализацией в сумме не равны 100%. Из этого следует, что помимо неправильной локализации существуют ещё причины, мешающие корректному построению целей. Поиск этих причин стоит сделать темой дальнейшего исследования. Доля подтверждённых срабатываний колеблется от 0% до 34,41%. Такие значения нельзя назвать плохими, но и, очевидно, что их недостаточно для промышленного использования. Двумя главными причинами таких результатов являются: относительно большое количество срабатываний с неверной локализацией (до 75-100%) и новизна символьного движка V#. Последний ещё находится на стадии активной разработки и имеет множество внутренних недочётов, которые мешают полностью исследовать все возможные пути анализируемых программ. Несмотря на это, V# нашёл заметное количество новых исключений. К сожалению,часть таких срабатываний хоть и не является ложноположительными в строгом смысле, но бесполезна т.к. V# для их выброса использовал приватные члены классов, которые недоступны обычному программисту. В целом, такой итог работы Veritas можно считать положительным т.к. он показывает перспективность гибридного анализа статическим и символьным подходами. Даже с учётом всех внутренних недочётов V#, получается подтвердить до трети срабатываний статического анализатора. Т.е. до трети срабатываний обогатятся стектрейсами. Более того, в случае с PVS-Studio ещё до запуска символьного исполнения Veritas опровергает до 75-100% срабатываний статического анализа с неправильной локализацией.

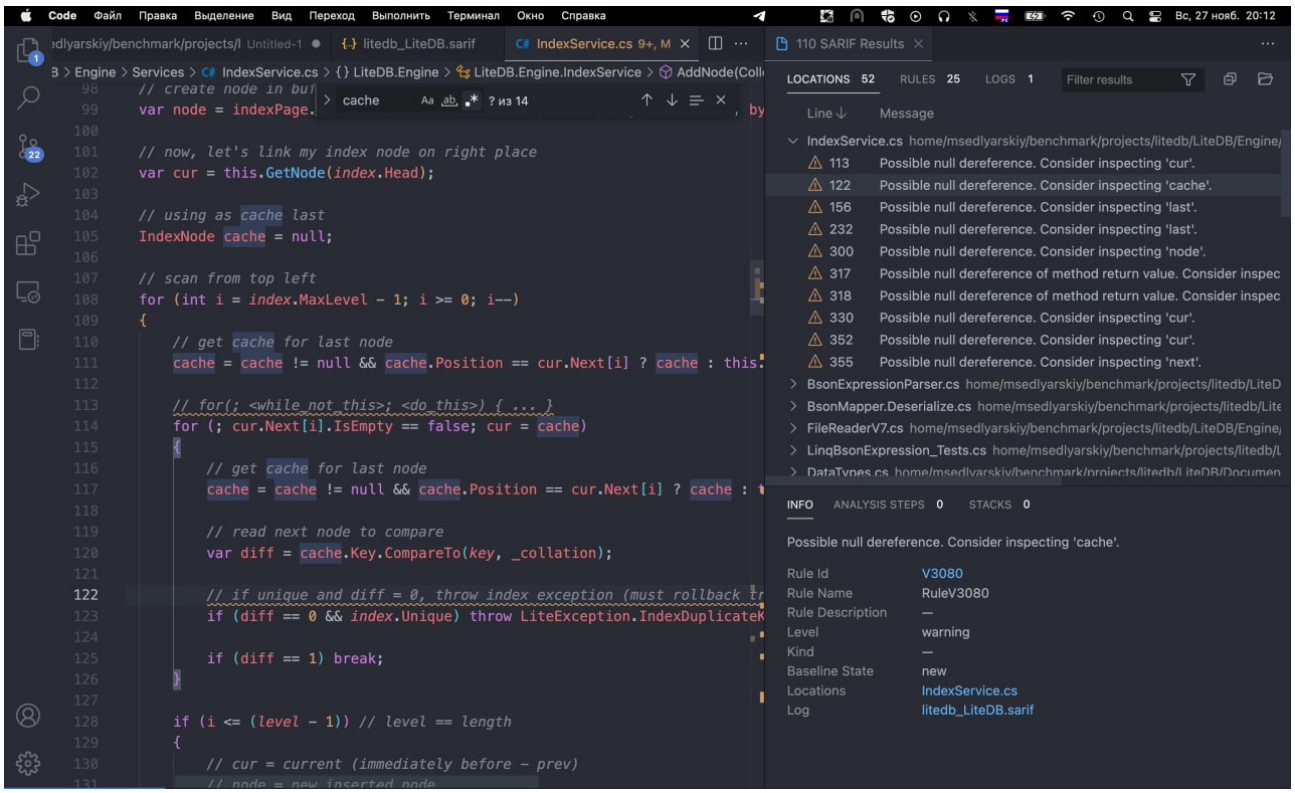

Рис. 3: Примеры некорректной локализации ошибок анализатором PVS-Studio

5. Применение

Полученный полученный программный продукт позволяет перепроверять и обогащать отчёты статического анализатора. Итоговый инструмент можно бесшовно встроить в CI процессы для экономии времени разработчиков. Ранжированные отчёты позволяют меньше времени тратить на ложноположительные срабатывания, а информация о стектрейсах быстрее находить ошибки в коде.

6. Реализация

В этом разделе описана реализация проекта Veritas. А именно: система преобразования результатов статического анализа в цели для символьного исполнения, дополнительные изменения в движке символьного исполнения в V#, и система сопоставления результатов символьного исполнения с результатами статического анализа. Код проекта можно посмотреть по ссылке на GitHub [27].

6.1. Загрузка сборок и их зависимостей

Как показала практика, платформа .net core не имеет полностью готового решения, позволяющего загружать dll сборки и их зависимости. Изначально предполагалось использовать собственное решение из проекта V#. Но выяснилось, что существующий подход ошибочен: он загружает сборки анализируемого приложения в общий контекст со сборками Veritas. Подобное смешение приводило к неопределённому поведению и трудно диагностируемым ошибкам. Поэтому был написан новый загрузчик сборок [28], который разделяет сборки в разные контексты. Были преложены новый способ загрузки сборок и зависимостей и алгоритм нормализации типов, который позволил правильно изолировать контекст сборок анализатора от контекста сборок анализируемого приложения.

Нормализация generic типов при загрузке сборок. Для краткости анализируемое приложение обозначим как aut (Application Under Testing). Рассмотрим случай, в котором V# используется как библиотека. Тогда сборки aut изначально присутствуют в контексте анализатора по умолчанию (DefaultAssemblyLoadContext). Чтобы избежать конфликтов, мы создаём отдельный контекст (VSharpAssemblyLoadContext) и загружаем в него aut сборки. И весь последующий анализ осуществляем именно с этими “копиями”.

Проблема при десериализации тестов. Допустим в vst файле закодирован объект типа *System.Collections.Generic.List‘1[MyType]* где *MyType* некий тип из анализируемого приложения. Далее стоит учесть

последующие положения:

1. Тип *System.Collections.Generic.List‘1* лежит в сборке *System.Private.CoreLib*

2. Сборка *System.Private.CoreLib* особенная и существует в единственном экземпляре только в *DefaultAssemblyLoadContext*

3. Отметим, что существует два контекста и в каждом из которых есть свой тип *MyType*. Рантайм дотнета считает их разными объектами, у которых просто совпадают имена.

4. Когда у сборки *System.Private.CoreLib* вызывается метод *GetType(System.Collections.Generic.List‘1[MyType])*, то она берёт тип MyType из своего контекста – контекста по умолчанию. И это в последующем ломает декодирование полей и их значений т.к. в других местах *MyType* взят из контекста *VSharpAssemblyLoadContext*. Dotnet не может привести два одинаковых типа из разных контекстов друг к другу.

5. В целевой картине *MyType* должен взяться из контекста *VSharpAssemblyLoadContext*. См. рисунок 4.

Предложенное решение. Предлагается все generic типы не получать наивным образом из сборки с помощью *GetType()*, а самостоятельно составлять их по кусочкам с помощью рекурсии. Так будет возможность выбирать на каждом шаге из какого контекста получить тот или иной тип.

6.2. Построение целей в Veritas и интеграция с V#

Построение индекса точек следования. Для получения точек следования из pdb файлов используется библиотека Mono.Cecil [12]. Задачу индексирования точек выполняет класс SequencePointsIndex [20]. Т.к. мы не можем узнать по имени файла с исходным кодом в какой

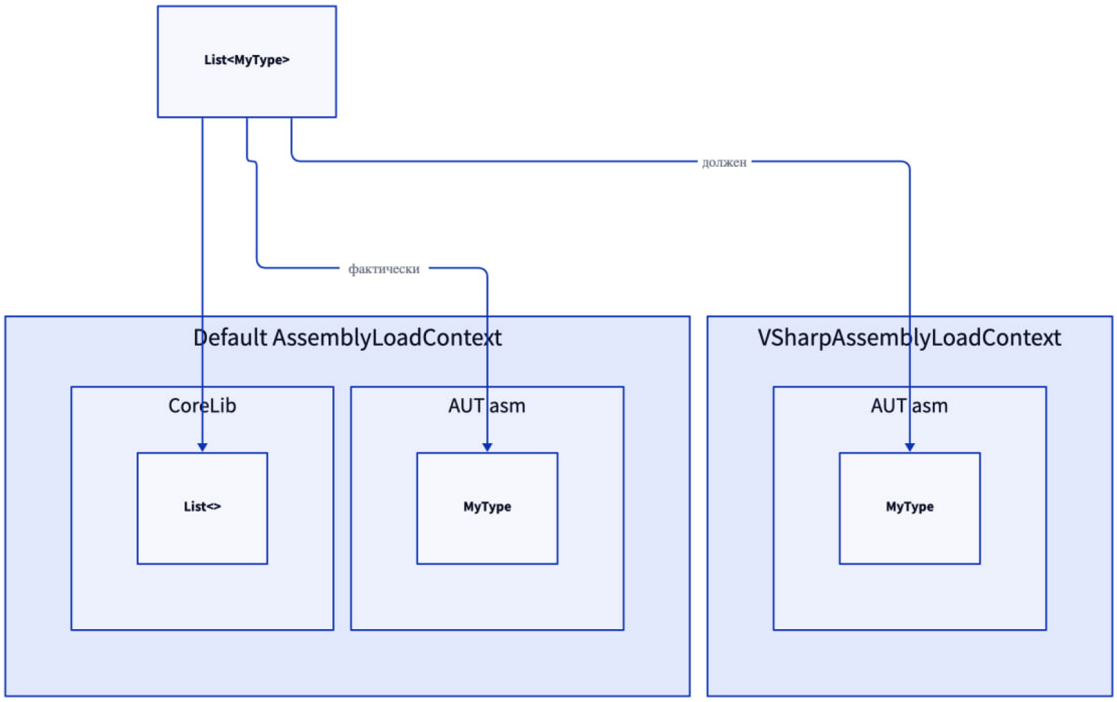

Рис. 4: Работа с несколькими контекстами сборок в .net core

сборке находится точка следования, то приходится жадно строить индекс по всем доступным сборкам. Одновременно строится обратный индекс, позволяющий по точке следования получить имя файла и номер строки. Обратный индекс используется при декодировании стектрейсов исключений, полученных после символьного анализа программы.

Фабрика целей символьного исполнения. Далее результаты из SARIF отчёта преобразуются в цели символьного исполнения в экземпляре класса TargetsFactory [23], который использует индекс типа SequencePointsIndex.

Передача целей в символьное исполнение и декодирование результатов. Полученные цели передаются в статический метод VSharp.TestGenerator.ProveHypotheses, созданный в рамках данной работы. Он запускает символьное исполнение в режиме HypothesisProveMode, который направляет символьный движок к заранее определённым целям. Режим HypothesisProveMode так же был разработан специаль-

но для интеграции Veritas с V#. В статистике, которую отдаёт метод ProveHypotheses, существует поле Exceptions, которое содержит список встреченных исключений с информацией об их типах и стектрейсах. Декодирование стектрейсов осуществляется через обратный индекс в экземпляре класса SequencePointsIndex.

Заключение

Были выполнены следующие задачи:

1. проведён сравнительный анализ статических анализаторов кода для платформы .NET;
2. были запущены статические анализаторы на настоящих проектах. На основании полученных результатов был выбран анализатор кода (PVS-Studio), с которым были проведены дальнейшие работы;
3. разработан и реализован алгоритм преобразования результатов статического анализатора кода в цели для движка символьного исполнения V#;
4. разработан и реализовать способ передачи целей в движок символьного исполнения;
5. проведён эксперимент, оценивающий качество полученного инструмента;

В качестве побочных достижений можно выделить следующие:

1. была выявлена ошибка в анализаторе Infer# от Microsoft; оказано содействие авторам в её устранении
2. была выявлена ошибка в анализаторе PVS-Studio; оказано содействие разработчикам проекта в её устранении;
3. разработан новый алгоритм загрузки сборок и предложен подход нормализации типов и зависимостей для платформы .net core. Эти нововведения значительно повысили стабильность движка символьного анализа V#;

Подводя итог, можно сказать, что поставленная цель была достигнута. Результаты Veritas можно считать положительными т.к. он показывает перспективность гибридного анализа статическим и символьным подходами. На текущий момент получается подтвердить до трети

срабатываний статического анализатора. Т.е. до трети срабатываний обогащается стектрейсами и сгенерированными тестами. Более того, в случае с PVS-Studio ещё до запуска символьного исполнения Veritas опровергает ряд срабатываний статического анализа с неправильной локализацией. Помимо основных целей были достигнуты побоычные, но не менее важные: был внесён вклад в open-source проект компании Microsoft и предложен новый подход загрузки сборок для open-source проекта V#.

Список литературы

[1] Analyzers benchmark. — https://github.com/m-sedl/analyzers-benchmark.

[2] AutoMapper. — https://github.com/AutoMapper/AutoMapper.

[3] BTCPayserver. — https://github.com/btcpayserver/btcpayserver/tree/63620409a99828dd937f9a212e935a3e15d52dc6.

[4] Belevantsev Andrey. Multilevel static analysis for improving program quality // Programming and Computer Software. — 2017. — 11. — Vol. 43. — P. 321–336.

[5] CHECKMARX SAST. — https://checkmarx.com/product/cxsast-source-code-scanning/.

[6] Combining dynamic symbolic execution, code static analysis and fuzzing / Seryozha Asryan, Jivan Hakobyan, Sevak Sargsyan, Shamil Kurmangaleev // Proceedings of the Institute for System Programming of the RAS. — 2018. — 12. — Vol. 30. — P. 25–38.

[7] English Lyn, Sriraman Bharath. Problem Solving for the 21st Century. — 2010. — 01.

[8] ILSpy. — https://github.com/icsharpcode/ILSpy.

[9] InferSharp. — https://github.com/microsoft/infersharp.

[10] InferSharp Issue 152. — https://github.com/microsoft/infersharp/issues/152#issuecomment-1280058520.

[11] LiteDB. — https://github.com/mbdavid/litedb/tree/6d9ac6237ff8cae104a3c57a8de4ec55b4506e87.

[12] Mono.Cecil. — https://www.mono-project.com/docs/tools+libraries/libraries/Mono.Cecil/.

[13] NLog. — https://github.com/NLog/NLog/tree/ace23d89ecf3dc2675648414a1f01ee9c1b2383d.

[14] Newtonsoft.Json. — https://github.com/JamesNK/Newtonsoft.Json.

[15] OpenRA. — https://github.com/OpenRA/OpenRA.

[16] PVS-Studio. — https://pvs-studio.com/ru/.

[17] Pex – White Box Test Generation for .NET. — https://www.microsoft.com/en-us/research/publication/pex-white-box-test-generation-for-net/.

[18] ReSharper. — https://www.jetbrains.com/help/resharper/InspectCode.html.

[19] Roslyn analysers. — https://github.com/dotnet/roslyn-analyzers.

[20] SequencePointsIndex. — https://github.com/m-sedl/veritas/blob/main/Veritas/SequencePointsIndex.cs.

[21] Static Analysis Results Interchange Format (SARIF) Version 2.0. — https://docs.oasis-open.org/sarif/sarif/v2.0/sarif-v2.0.html.

[22] A Survey of Symbolic Execution Techniques / Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia et al. // ACM Computing Surveys (CSUR). — 2018. — Vol. 51. — P. 1 – 39.

[23] TargetsFactory. — https://github.com/m-sedl/veritas/blob/main/Veritas/TargetsFactory.cs.

[24] The Economic Impacts of Inadequate Infrastructure for Software Testing : Planning Report 02-3 / National Institute of Stan- dards and Technology, ; Executor: Gregory Tassey : 2002.

[25] Using targeted symbolic execution for reducing false-positives in dataflow analysis / Steven Arzt, Siegfried Rasthofer, Robert Hahn, Eric Bodden. — 2015. — 06. — P. 1–6.

[26] V# Symbolic execution engine for .NET Core. — https://github.com/VSharp-team/VSharp.

[27] Veritas. — https://github.com/m-sedl/veritas.

[28] VeritasAssemblyLoadContext. — https://github.com/m-sedl/veritas/blob/main/Veritas/VeritasAssemblyLoadContext.cs.

[29] moq4. — https://github.com/Moq/moq4.

[30] nunit. — https://github.com/nunit/nunit.

[31] parallel-programming-1-sync. — https://github.com/m-sedl/analyzers-benchmark/tree/main/projects/parallel-programming-1-sync.

[32] parallel-programming-1-thread-pool. — https://github.com/m-sedl/analyzers-benchmark/tree/main/projects/parallel-programming-1-thread-pool.

[33] parallel-programming-3-thread-pool. — https://github.com/m-sedl/analyzers-benchmark/tree/main/projects/parallel-programming-3-thread-pool.

[34] sonar-dotnet. — https://github.com/SonarSource/sonar-dotnet.

[35] spbu-homeworks-1. — https://github.com/ElenaBakova/Homeworks.

[36] xunit. — https://github.com/xunit/xunit.

[37] Платформа Roslyn. — https://github.com/dotnet/roslyn.