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 технологической практике

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

Санкт-Петербург
2022

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 [20].

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 состояния программы анализируются и часть недостижимых. Отсюда появляются ложноположительные срабатывания статического анализа.

Также можно выделить символьное исполнение. Это метод моделирования работы программы с использованием символьных значений вместо конкретных. Во время символьного исполнения, встретив оператор ветвления, анализатор добавит условие (ограничение) к себе в список 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 бесконечным.

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: избавляемся от ложноположительных срабатываний статического анализа и сокращаем пространство поиска символьного анализа.

Целью работы является разработка и реализация приложения устраняющего ложные срабатывания статических анализаторов кода символьным исполнением для платформы .NET.

1. Постановка задачи

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

1.1. Осенний семестр

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.

1.2. Весенний семестр

1. разработать и реализовать способ передачи целей в движок символьного исполнения;

2. разработать и реализовать алгоритм ранжирования результатов статического анализа на основе результатов символьного исполнения

3. 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 [3].

Создание анализаторов для языка C# заметно упростилось в 2015 году после того, как компания Microsoft выложила в открытый доступ исходный код проекта Roslyn [26]. 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 [15], ReSharper [14], PVS-Studio [12], InferSharp [7], sonar-dotnet [25], CHECKMARX SAST [4].

2.2. Символьное выполнение

Символьное выполнение [18] — 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 [6]. 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# [22] и Pex [13]. 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 [5] описывается комбинирование символьного исполнения, статического анализа и фаззинга для увеличения тестового покрытия бинарных приложений. А в работе [21] авторы проводят статический анализ потока данных android-приложений и удаляют ложноположительные срабатывания символьным движком TASMAN.

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

3. Метод

3.1. Общая схема решения

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

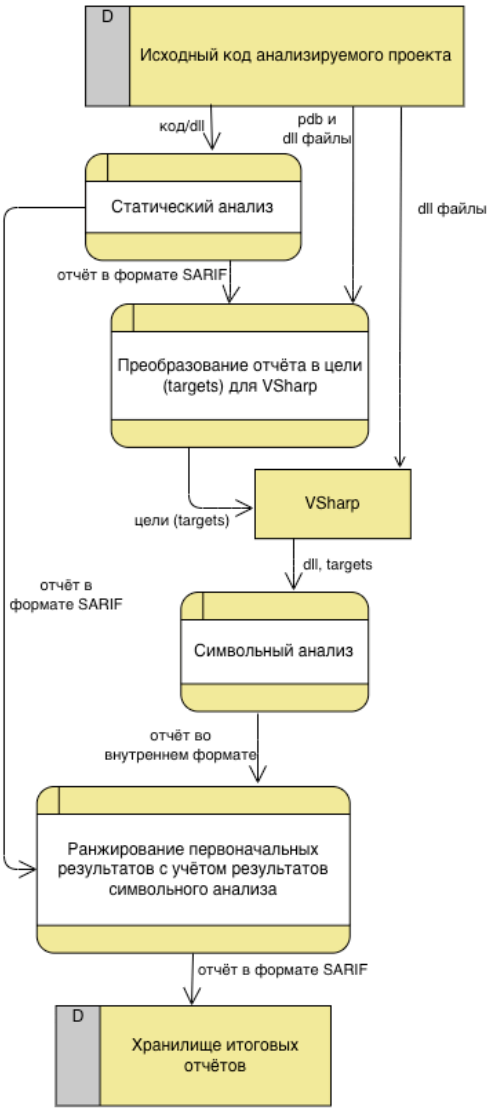

Рис. 1: Общая DFD-схема. Предлагаемое решение по своей сути является обёрткой вокруг символьной машины VSharp. На схеме не указан какой-либо конкретный статический анализатор т.к. Veritas может работать с целым перечнем анализаторов

Принцип действия следующий:

1. Пользователь с помощью статического анализатора кода получает отчёт об исследуемом проекте. Стандартом индустрии представления отчётов является json-based формат SARIF (Static Analysis Results Interchange Format) [17].

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.

На рисунке 2 изображена примерная схема развёртывания.

3.2. Выбор статического анализатора

Т.к. формат SARIF де-факто является стандартом индустрии, Veritas теоретически может работать с разными анализаторами. Но для последующих реализации и эксперимента имеет смысл выбрать такой анализатор, который:

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

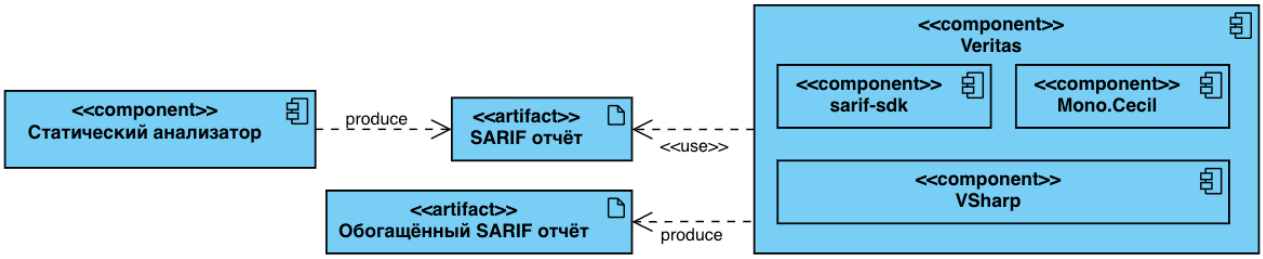

Рис. 2: Схема развёртывания. Veritas использует VSharp как библиотеку наряду с библиотеками Mono.Cecil для работы с pdb файлами и sarif-sdk для работы с отчётами стат. анализаторов

2. Поддерживает .net core.

3. Находит в проектах значительное количество ошибок, которые можно проверить символьным исполнением: разыменование нулевого указателя, выход за границы массива, постоянное равенство выражения какому-либо значению и т. д.

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

5. Обладает поддержкой авторов. Возможность быстро исправить найденную ошибку является не решающим, но ощутимым фактором.

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 сведена базовая информация о самых распространённых анализаторах для платформы .NET.

Стоит отметить, что из-за политической обстановки 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 этих 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. В таблице 2 видно, что лидерами по количеству срабатываний для большинства проектов стали Roslyn analyzers. Но большинство этих проверок относятся к стилю кода и их невозможно провалидировать символьным исполнением. То же замечание применимо и к sonar-dotnet. В процесс тестирования Infer# от Microsoft была обнаружена неисправность анализатора. Некорректно работала загрузка сборок. По этому поводу было сделано обращение к авторам проекта и совместными усилиями ошибка устранена [8]. 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). С помощью

поддержки анализатора 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	Infer#				sonar-scanner-msbuild			pvs studio				
					roslyn	sonar-dotnet	прочее					
	NULLPTR_DEREFERENCE	PULSE_RESOURCE_LEAK	THREAD_SAFETY_VIOLATION	STACK_VARIABLE_ADDRESS_ESCAPE	Всего	Всего	Всего	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	107	2	8	0	264	2893	487	анализ решения съедает все ресурсы на сервере				
litedb	1	7	6	0	0	470	9	15	19	1	0	110
moq4	0	1	0	0	214	441	5	4	2	0	0	61
NLog	30	75	44	0	17	266	12	34	9	0	1	163
nunit	26	192	7	0	218	856	4	28	8	0	0	203
xunit	0	0	0	0	292	1092	0	19	11	0	0	127
btcpayserver	4	6	4	0	267	1062	1	37	99	13	0	374
AutoMapper	0	0	0	0	88	167	0	2	1	0	0	38
spbu-homework	0	1	4	0	33	105	0	0	0	0	2	5
parallel-program	0	0	0	0	0	2	0	0	0	0	0	0
parallel-program	0	0	0	0	11	3	6	0	0	0	0	0
parallel-program	0	0	0	0	8	10	0	0	0	0	0	2
BenchmarkDotN	2	11	0	0	18	19	0	14	0	0	0	82
ILSpy (ILSpy.XP	48	4	0	2	873	1829	0	98	128	0	17	792
OpenRA	34	49	1	0	0	2990	0	13	39	0	2	342
Newtonsoft.Json	14	271	5	0	1478	740	0	30	10	0	34	265
RestSharp	0	53	0	0	28	47	9	0	1	0	0	16

Таблица 2: Результаты запуска анализаторов на 17 проектах

3.3. Внутренняя структура Veritas

Построение целей для символьного исполнения из результатов статического анализатора основано на следующих принципах:

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

2. Одним из артефактов сборки .net приложений являются pdb файлы. Помимо прочей метаинформации, в них сопоставлены некоторые места в исходном коде на номера инструкций в конкретных сборках (dll файлах). Такие места называются точками следования.

3. 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.

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

1. Поочердёно загружаются точки следования из всех сборки исследуемого проекта, у которых есть соответствующие pdb файлы. Точки следования хранятся в хэш-таблице, где ключом следования выступает имя файла исходного кода.

2. Затем просматриваются результаты из отчёта статического анализатора. Если тип результата можно проверить с помощью символьного исполнения, то для локации результата из индекса достаются соответствующие точки следования. Список поддерживаемых типов заранее занесён в Veritas на стадии разработки.

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

4. Эксперимент

Проверим насколько полно Veritas может покрыть целями (targets) ошибки из отчётов PVS-Studio.

4.1. Условия эксперимента

Генератор целей Veritas будет запускаться на 3-ёх проектах: BTCPayserver [2], NLog [11] и LiteDB [9]. Каждый из этих проектов был предварительно проанализирован с помощью PVS-Studio. Затем подсчитывается доля срабатываний, для которых получилось сгенерировать цели, от общего числа срабатываний подходящих типов: V3080 (Possible null dereference), V3146 (Possible null dereference. A method can return default null value), V3106 (Possibly index is out of bound).

4.2. Исследовательские вопросы

Узнать насколько полно Veritas покрывает целями ошибки из отчётов PVS-Studio.

4.3. Метрики

Хорошим покрытием будет считаться доля свыше 50% для каждого проекта.

4.4. Результаты

В таблице 3 видно, что доля покрытия результатов целями варьируется от 58,77% до 70%.

Проект	Срабатывания с целями, шт	Поддерживаемые срабатывания, шт	Доля
BTCPayServer	67	114	58,77%
NLog	7	10	70,00%
LiteDB	12	20	60,00%

Таблица 3: Результаты построения целей для трёх проектов

4.5. Обсуждение результатов

Чем можно объяснить, что от 30% до 40% результатов PVS-Studio не удалось отобразить на IL инструкцию? Неужели pdb информация настолько неполная? Отнюдь. Ручной анализ показал, что почти все результаты PVS-Studio, оставшиеся без целей, имеют некорректную привязку к местоположению в коде. Например, на рисунке 3 видно, что ошибки вида null dereference локализованы на строк с комментариями. Очевидно, что таким строкам не сопоставлены никакие точки следования в pdb файлах. В целом, такой итог работы построения целей в Veritas можно считать положительным т.к. ещё до символьного исполнения отбрасывается ряд некорректных срабатываний статического анализатора.

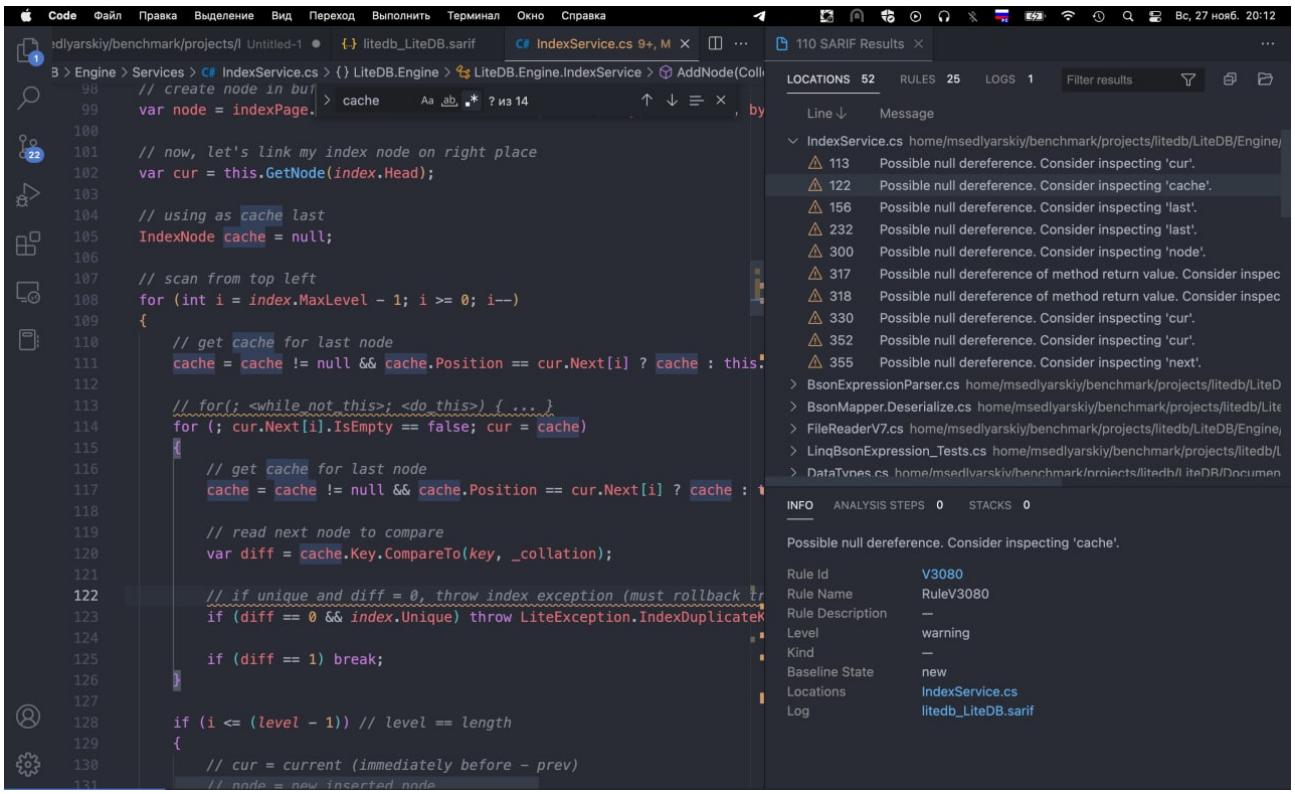

Рис. 3: Примеры некорректной локализации ошибок анализатором PVS-Studio

5. Применение

Полученный генератор целей в последствии будет применим для запуска символьного исполнения. Результат символьного исполнения будет использоваться для перепроверки и обогащения изначального отчёта статического анализатора. Итоговый инструмент можно будет бесшовно встроить в CI процессы для экономии времени разработчиков: чем меньше они реагируют на ложноположительные срабатывания, тем больше они занимаются полезной деятельностью.

6. Реализация

В этом разделе описана реализация проекта Veritas. А именно: система преобразования результатов статического анализа в цели для символьного исполнения. Код проекта можно посмотреть по ссылке на GitHub [23].

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 [24], который разделяет сборки в разные контексты. На данный момент такой подход хорошо справляется с поставленной задачей.

Построение индекса точек следования. Для получения точек следования из pdb файлов используется библиотека Mono.Cecil [10]. Задачу индексирования точек выполняет класс SequencePointsIndex [16]. Т.к. мы не можем узнать по имени файла исходного кода в какой сборке находится точка следования, то приходится жадно строить индекс по всем доступным сборкам.

Фабрика целей символьного исполнения. Далее результаты из SARIF отчёта преобразуются в цели символьного исполнения в экземпляре класса TargetsFactory [19], который использует индекс типа SequencePointsIndex.

Заключение

В осеннем семестре 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#.

В качестве побочных достижений можно выделить следующие:

1. была выявлена ошибка в анализаторе Infer# от Microsoft; оказано содействие разработчикам проекта в её устранении
2. была выявлена ошибка в анализаторе PVS-Studio; оказано содействие разработчикам проекта в её устранении;
3. разработан алгоритм загрузки сборок и зависимостей для платформы .net core.

В следующем семестре планируется:

1. разработать и реализовать способ передачи целей в движок символьного исполнения;
2. разработать и реализовать алгоритм ранжирования результатов статического анализа на основе результатов символьного исполнения
3. провести эксперименты и оценить качество полученного инструмента.

Список литературы

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

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

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

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

[5] 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.

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

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

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

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

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

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

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

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

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

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

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

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

[18] 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.

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

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

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

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

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

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

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

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