DOI: 10.15514/ISPRAS-2025-37(2)-6

TSAR: инструмент для оценки статических анализаторов

[1] *К.А. Чибисов, ORCID: 0009-0008-4371-5475 <chibisov@ispras.ru>*
[1] *Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1,2] *А.Д. Тимонин, ORCID: 0009-0006-7453-0209 <timonin.a@phystech.edu>*
[2] *В.И. Лазарь, ORCID: 0009-0003-1958-3621 <lazar.v@phystech.edu>*
[1,3] *Д.М. Журихин, ORCID: 0009-0003-0239-2220 <zhur@ispras.ru>*
[1,3] *А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский физико-технический институт, 141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.*
[3] *Московский государственный университет имени М.В. Ломоносова, Россия, 119991, Москва, Ленинские горы, д. 1.*

Аннотация. В статье представляется новый инструмент TSAR, предназначенный для оценки эффективности статических анализаторов. TSAR включает в себя три основных компонента: систему анализа статических анализаторов, генератор тестов, базирующийся на Common Weakness Enumeration (CWE), и механизмы трансформации кода (мутатора) для усложнения работы анализаторов. Система анализа позволяет выявлять слабые места в инструментах статического анализа, в то время как генератор тестов создает специфические случаи на основе известных уязвимостей. Трансформации кода позволяют создавать сложные структуры, затрудняющие анализ и призванные проверить устойчивость анализаторов к обнаружению реальных уязвимостей. Данный инструмент предоставляет исследователям и разработчикам возможность для более глубокой оценки качества статических анализаторов программного обеспечения для их дальнейшего улучшения.

Ключевые слова: статический анализ; тесты; генератор тестов; трансформации; классификации уязвимостей CWE; компиляторная инфраструктура MLIR; проект LLVM; компилятор clang; анализатор cppcheck; тестовый набор Juliet; статический анализатор clang-tidy.

Для цитирования: Чибисов К.А., Бучацкий Р.А., Тимонин А.Д., Лазарь В.И., Журихин Д.М., Белеванцев А.А. TSAR: инструмент для оценки статических анализаторов. Труды ИСП РАН, том 37, вып. 2, 2025 г., стр. 79–96. DOI: 10.15514/ISPRAS–2025–37(2)–6.

TSAR: Tool for Static Analyzers Ranking

[1] *K.A. Chibisov, ORCID: 0009-0008-4371-5475 <chibisov@ispras.ru>*
[1] *R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>*
[1,2] *A.D. Timonin, ORCID: 0009-0006-7453-0209 <timonin.a@phystech.edu>*
[2] *V.I. Lazar, ORCID: 0009-0003-1958-3621 <lazar.v@phystech.edu>*
[1,3] *D.M. Zhurikhin, ORCID: 0009-0003-0239-2220 <zhur@ispras.ru>*
[1,3] *A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the Russian Academy of Sciences,*
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] *Moscow Institute of Physics and Technology,*
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.
[3] *Lomonosov Moscow State University,*
GSP-1, Leninskie Gory, Moscow, 119991, Russia.

Abstract. The article presents a new tool, TSAR, designed for evaluating the effectiveness of static analyzers. TSAR includes three main components: a static analyzer assessment system, a test generator based on the Common Weakness Enumeration (CWE), and code transformation mechanisms (mutators) to challenge the analyzers. The assessment system identifies weaknesses in static analysis tools, while the test generator creates specific cases based on known vulnerabilities. Code transformations create complex structures that complicate analysis and intended to test the analyzers' ability in detecting real vulnerabilities. This tool provides researchers and developers with an opportunity for a deeper assessment of the quality of software static analyzers for their further improvement.

Keywords: static analysis; tests; test generator; transformations; CWE; MLIR; LLVM; clang; cppcheck; Juliet; clang static analyzer; clang-tidy.

For citation: Chibisov K.A., Buchatskiy R.A., Timonin A.D., Lazar V.I., Zhurikhin D.M., Belevantsev A.A. TSAR: Tool for Static Analyzers Ranking. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 2, 2025, pp. 79-96 (in Russian). DOI: 10.15514/ISPRAS-2025-37(2)-6.

1. Введение

Современное программное обеспечение становится всё более сложным и многогранным, что делает его устойчивость к уязвимостям критически важной задачей для разработчиков и специалистов в области безопасности. Статический анализ кода зарекомендовал себя как один из ключевых методов обнаружения потенциальных уязвимостей на ранних этапах разработки. С увеличением сложности программных систем и разнообразия языков программирования возрастает необходимость в эффективных инструментах анализа кода. Однако множество статических анализаторов и их различные подходы к выявлению уязвимостей затрудняют выбор наилучшего инструмента для конкретной задачи.

Для обеспечения качественной защиты программного обеспечения недостаточно только использовать статические анализаторы, необходимо также регулярно оценивать их эффективность и точность. В связи с этим становится важным создание инструментов, позволяющих проводить объективную оценку различных статических анализаторов и выявлять их слабые места. Актуальными также остаются задачи генерации тестов, формирующих специфические случаи для анализа, и выявление возможностей для усовершенствования самих анализаторов.

В данной статье мы представляем TSAR (Tool for Static Analyzers Ranking) – новый инструмент, разработанный для комплексной оценки статических анализаторов. TSAR включает в себя три ключевых компонента: систему для анализа эффективности статических анализаторов, генератор тестов, основывающийся на классификации уязвимостей CWE

(Common Weakness Enumeration) [1], и механизмы трансформации кода, направленные на усложнение работы анализаторов.

TSAR может помочь в выявлении недостатков статических анализаторов, связанных с отсутствием различных методов анализа, таких как внутрипроцедурный анализ потоков данных и управления, межпроцедурный контекстно-чувствительный анализ, чувствительный к путям выполнения анализ и анализ помеченных данных. Эти усилия направлены на усовершенствование статических анализаторов и, как следствие, на повышение общей безопасности программного обеспечения.

В работе продемонстрированы возможности TSAR как средства для оценки и улучшения статических анализаторов, а также представлены результаты его применения на реальных примерах.

2. Обзор существующих решений

2.1 Методы оценки статических анализаторов

В данном разделе рассматриваются современные методы оценки статических анализаторов, основанные на обзоре ряда научных исследований.

В работе [2] предложен подход к тестированию статических анализаторов путем генерации случайных программ. Авторы аргументируют, что традиционные тестовые наборы могут быть предвзятыми и не отражать полноту возможных сценариев, с которыми может столкнуться анализатор.

Для создания синтаксически корректных, но семантически непредсказуемых программ на языке C, авторы использовали генератор Csmith [3]. Этим проверяли возможности статических анализаторов обрабатывать неконтролируемое множество программных конструкций и выявление потенциальных ошибок в их работе. Для тестирования подхода авторы использовали фреймворк для анализа и трансформации программ Frama-C. В процессе тестирования были обнаружены уязвимости и неточности в работе анализатора, что подчеркивает важность использования случайной генерации в процессе их оценки.

С помощью предложенного авторами подхода можно проверять анализаторы на устойчивость к нетривиальным и редко встречающимся конструкциям для обнаружения скрытых дефектов, которые могут не проявиться при использовании стандартных тестовых наборов. Однако сгенерированные программы могут быть слишком синтетическими и не отражать реальные сценарии использования. Также отсутствие контекста и семантического значения может затруднять интерпретацию результатов.

В исследовании [4] проведен сравнительный анализ открытых статических анализаторов для обнаружения уязвимостей в коде на C/C++. Авторами были рассмотрены наиболее популярные открытые анализаторы, такие как Cppcheck [5], Infer [6], Clang Static Analyzer [7] и OCLint [8].

Для оценки анализаторов был использован тестовый набор Toyota ITC [9], охватывающий значительное количество различных типов уязвимостей из списка CWE, включая переполнения буфера, утечки памяти и другие. Авторы разработали фреймворк, который выполняет запуск набора статических анализаторов по всем тестовым случаям. Для каждого тестового случая автоматически анализируется, были ли обнаружены соответствующие уязвимости в файле манифеста с использованием инструментов статического анализа. По результатам анализа генерируется отчет, содержащий различные статистические данные, в том числе точность, полнота и время выполнения для каждого инструмента, а также для каждой категории уязвимостей. Соответствие между дефектами, выявленными инструментом, и дефектами в тестовом наборе ITC устанавливалось на основе номера строки в исходном коде. Этот подход позволяет эффективно обрабатывать инструменты

статического анализа, которые используют различные таксономии слабых мест, минимизируя риск неправильной классификации отдельных дефектов.

По итогам исследования были сделаны выводы, что ни один из анализаторов не продемонстрировал абсолютного превосходства по всем показателям; существуют значительные различия в специализации и эффективности инструментов в зависимости от типа уязвимостей.

К ограничениям исследования можно отнести, что оценка анализаторов ограничена выбранным набором тестовых случаев и может не охватывать все возможные уязвимости. Также некоторые инструменты могут быть настроены для улучшения результатов, что не всегда учитывалось.

В работе [10] предложен метод дифференциального тестирования для оценки корректности и точности статических анализаторов. Метод заключается в сравнении выходных данных различных анализаторов при обработке одного и того же исходного кода для выявления расхождений в результатах, которые могут указывать на проблемы с корректностью или точностью одного, или нескольких инструментов.

Процесс тестирования заключается в генерации исходных кодов на языке C с использованием различных техник, включая мутацию существующих программ с дальнейшим анализом эталонных результатов и автоматическим обнаружением противоречий.

Метод позволил обнаружить ранее неизвестные дефекты в ряде популярных анализаторов и показал эффективность в автоматизированном выявлении проблем без необходимости в предварительно размеченных данных, что минимизирует человеческий фактор.

Метод требует наличия нескольких анализаторов для сравнения, которые должны поддерживать одинаковые языки и конструкции, а также не гарантирует выявление всех возможных дефектов, особенно если ошибки присутствуют во всех сравниваемых инструментах.

В исследовании [11] авторами была представлена методика оценки статических анализаторов с использованием дифференциального мутационного анализа.

Мутационный анализ позволяет сделать целенаправленные изменения (мутации) в исходном коде для создания новых вариантов программ. А дифференциальный подход позволяет сравнивать результаты анализа исходной и мутированной программы для выявления несоответствий. Это позволяет оценивать чувствительность статических анализаторов к определенным изменениям в коде.

Авторы использовали различные типы мутаций, имитирующих потенциальные ошибки или уязвимости. Экспериментальные результаты представляют собой набор сравнений инструментов для трех языков: Solidity (самый популярный язык для смарт-контрактов), Java и Python. Для оценки анализаторов выполнялись автоматический запуск инструментов статического анализа на оригинальном и мутированном коде и сравнение результатов.

Результаты мутационного анализа подтвердили целесообразность использования нескольких инструментов: за исключением одного из анализаторов для Java, все исследуемые инструменты уникально обнаружили более 1000 мутаций, при этом для Java и Python не было выявлено ни одной мутации, обнаруженной всеми инструментами.

Метод позволил идентифицировать области, в которых анализаторы недостаточно чувствительны к изменениям или, наоборот, дают ложные срабатывания. Стоит отметить, что процесс анализа может быть ресурсозатратным из-за большого числа вариантов мутированного кода, а также требующим тщательной проработки типов мутаций для обеспечения релевантности результатов.

В работе [12] предлагается автоматизированный и воспроизводимый подход для оценки эффективности пяти открытых и одного коммерческого статического анализатора кода на языке C, используя эталонный набор данных, состоящий из 27 открытых проектов с 192 известными уязвимостями безопасности (CVE). Исследование было проведено с целью

определить, можно ли использовать выбранный набор данных для оценки таких анализаторов, и какая степень детализации кода (относительно выбранного набора данных) является подходящей для приближенного обнаружения уязвимостей.

Результаты исследования показали, что ни один из анализируемых инструментов не обеспечивает полного покрытия всех типов уязвимостей. Эффективность обнаружения существенно варьируется в зависимости от категории уязвимости и особенностей кода. Некоторые инструменты продемонстрировали высокую точность в определенных областях, но оказались менее эффективными в других. Авторы отмечают, что сложные конструкции языка C, такие как указатели и динамическое выделение памяти, представляют значительные трудности для статического анализа. Они также подчеркивают важность контекстной информации и межпроцедурного анализа для повышения точности обнаружения уязвимостей.

2.2 Открытые инструменты оценки статических анализаторов

В этом разделе обсуждаются доступные тестовые наборы и инструменты для оценки статических анализаторов кода, которые применяют различные CWE для оценки качества анализа.

Инструмент для сравнения статических анализаторов OSSF CVE Benchmark [13] использует проекты с открытым исходным кодом с известными уязвимостями CVE. Анализатор тестируется сначала на коде до применения исправления, устраняющего уязвимость, и затем после. Тестирование на коде реальных проектов позволяет понять практическую ценность анализаторов, но в то же время не дает информации о том, какие техники использует анализатор. Кроме того, разметка данных на основе баз уязвимостей CVE может быть выбрана так, что покрывается только один вид анализа, например, анализ на уровне синтаксического дерева разбора.

Тестовый набор Juliet [14] включает программы на языках C/C++/Java, сгенерированные на основе шаблонов, а также написанные вручную. Этот набор покрывает суммарно 116 различных CWE. Каждый тест имеет градацию сложности, начиная от простой функции с линейным потоком управления и заканчивая межпроцедурными программами и программами, использующими несколько единиц трансляции. Juliet активно используется индустрией и хорошо позволяет оценить различные анализаторы исходного кода. Из недостатков можно выделить то, что не всегда понятно, как конкретный тест был получен, и какие возможности анализатора он пытается проверить в конкретном тесте.

Тестовый набор C/C++ программ Toyota ITC [9], в отличие от Juliet, имеет значительно меньшее покрытие CWE, но каждый тест объясняет, почему он так построен и какой аспект анализатора он проверяет. Это позволяет более точно говорить о покрытии различных аспектов анализа. Основным недостатком данного набора является то, что он полностью написан вручную и преимущественно проверяет возможности внутрипроцедурного анализа статических анализаторов.

Проект saveourtool [15] предоставляет утилиту для запуска произвольных анализаторов, хранения результатов анализа, сравнения результатов, а также представления статистики за все запуски анализаторов на веб-сервере. Данная утилита не предоставляет своих тестовых наборов, но может быть использована для ведения статистики анализаторов на наборах тестов, таких как Juliet и Toyota ITC. Утилита требует добавления анализатора в свою инфраструктуру для запуска его на каждом тесте, что не всегда удобно для анализаторов, которые анализируют один большой проект за раз, а не отдельные файлы в этом проекте.

Рассмотренные инструменты и тестовые наборы предоставляют различные подходы к оценке качества анализаторов кода. OSSF CVE Benchmark позволяет оценить практическую ценность анализаторов на реальных проектах, но не дает информации о техниках анализа. Набор тестов Juliet активно используется в индустрии и хорошо покрывает различные CWE,

но не всегда понятно, как конкретные тесты были получены. Тестовый набор Toyota ITC предоставляет более детальное объяснение каждого теста, но имеет меньшее покрытие CWE. Проект saveourtool предоставляет утилиту для ведения статистики анализаторов, но требует добавления анализатора в свою инфраструктуру.

3. Инструмент для оценки статических анализаторов

В данном разделе описана общая схема предложенного инструмента для оценки статических анализаторов TSAR. Детальное описание реализации каждого компонента приведено в последующих разделах 4 и 5.

Из проведенного обзора существующих решений становится очевидным, что при сравнении статических анализаторов необходимо учитывать несколько ключевых факторов. Во-первых, важна метаинформация о тестах, которая позволяет лучше понять контекст и природу уязвимостей, что, в свою очередь, способствует более глубокому анализу и точному выявлению проблем. Во-вторых, необходимо обеспечивать поддержку нескольких языков программирования, поскольку многие современные разработки используют различные технологии в своих проектах, включая C, C++ и Java. В-третьих, внедрение контекстной информации, в частности межпроцедурного анализа, становятся очевидными для повышения точности выявления уязвимостей.

На основе идей использования мутационных техник для оценки статических анализаторов, предложенных в работах [10] и [11], а также подходов, акцентирующих внимание на многообразии тестов на основе CWE из работы [4], и успеха тестового набора Juliet [14], был разработан инструмент TSAR. Этот инструмент включает в себя три основных компонента:

- Шаблоны, основанные на CWE: система шаблонов, которая обеспечивает глубокое покрытие различных категорий уязвимостей. Эта структура позволяет легко адаптировать и расширять наборы тестов, что делает их более релевантными для анализа современных программных решений.
- Мутатор шаблонов (tsar): этот компонент отвечает за трансформацию шаблонов и генерацию тестовых программ на языках C, C++ и Java. Использование мутационных техник позволяет не только выявить слабые места статических анализаторов, но и улучшить качество тестов, увеличивая их сложность и разнообразие.
- Утилита для сбора результатов анализа (metalyzer): этот инструмент не только агрегирует результаты анализа, но и предоставляет возможности для их сравнения между разными анализаторами. Метаинформация, собранная в процессе, позволяет лучше понять эффективность анализаторов в различных контекстах.

Идея шаблонов исходит из концепции, что в каждом тесте есть структура, специфичная для определенной ошибки, и варианты потока, создаваемые с целью проверки ключевых особенностей анализа, таких как межпроцедурность, чувствительность к путям, а также к полям структур и другим аспектам. Это сходство с подходом Juliet и ГОСТ Р 71207-2024 [16] обеспечивает более гибкую и целенаправленную проверку функциональности анализаторов. Именно поэтому в TSAR шаблоны разрабатываются вручную экспертами, которые лучше понимают особенности различных уязвимостей, в то время как мутации предоставляют поточные варианты тестов. Эта структура позволяет инструменту адаптироваться к специфическим требованиям анализа кода.

На рис. 1 представлена общая схема работы инструмента TSAR. Мутатор шаблонов (tsar) – это ядро инфраструктуры инструмента TSAR. Он использует шаблоны, написанные с использованием инфраструктуры MLIR [17] и построенные вокруг различных CWE, для генерации тестовых программы на языках C, C++ и Java. Эти программы применяются для проверки и анализа инструментов статического анализа кода. Результаты анализа сохраняются в формате, подходящем для экспорта в утилиту metalyzer: SARIF [18] или

простой текстовый вывод анализатора. metalyzer импортирует информацию о тестовом наборе, результатах анализа и выполняет сравнение анализаторов, что позволяет оценить эффективность и точность различных анализаторов кода. Стоит отметить, что metalyzer не запускает анализаторы самостоятельно, а работает с уже готовыми результатами анализа.

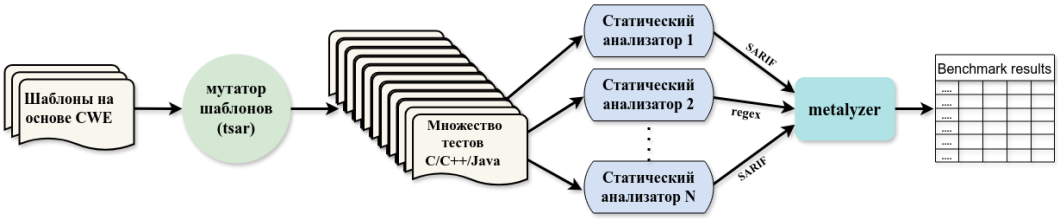

Рис. 1. Схема инструмента для оценки статических анализаторов TSAR.
Fig. 1. Diagram of TSAR tool for ranking static analyzers.

4. Генератор тестов

В данном разделе рассмотрены устройства мутатора шаблонов и генератора тестов tsar.

tsar – это утилита на основе инфраструктуры MLIR, предназначенная для мутации шаблонов и генерации программ на языках C, C++ и Java.

MLIR (Multi-Level Intermediate Representation) [17] – компиляторная инфраструктура для разработки промежуточных представлений для дальнейшего анализа, оптимизации и генерации кода. Является частью проекта LLVM.

Инфраструктура MLIR предлагает абстракции для разработки произвольных промежуточных представлений с иерархической структурой, аналогичной LLVM IR. Основные компоненты MLIR включают: операции, задающие семантику выполнения; регионы как контейнеры для операций; блоки, представляющие линейные последовательности операций. Взаимосвязь между этими элементами создает рекурсивную структуру: операция включает регион, регион состоит из блоков, каждый блок содержит операции, и так далее. Объединение различных операций, типов и атрибутов формирует диалект, который может быть представлен как библиотека для конкретного промежуточного представления. MLIR поддерживает более сорока встроенных диалектов для различных сценариев. Одной из ключевых особенностей MLIR является возможность определения пользовательских диалектов с использованием специализированного предметно-ориентированного языка.

Применение MLIR обусловлено стремлением иметь свой высокоуровневый язык описания шаблонов, который будет более выраженным по сравнению с C и C++, а также позволит эффективно разрабатывать трансляторы в другие языки программирования, такие как Java или Go.

Инфраструктура MLIR предоставляет диалект EmitC, который соответствует указанным требованиям. EmitC [19] – это строго типизированный диалект, схожий с языком C, предназначенный для трансляции операций из более абстрактных диалектов в операторы самого EmitC с последующей генерацией программ на языках C и C++. Этот диалект служит конечной точкой в дереве диалектов MLIR, что означает невозможность выполнения трансляции из него в другие диалекты выше по иерархии. Трансляция возможна только напрямую в языки C/C++ с использованием инструмента CppEmitter. Каждая операция, атрибут и другие сущности в EmitC должны иметь аналоги в диалектах выше по иерархии. В противном случае такие элементы будут считаться "мертвыми", так как их невозможно будет транслировать через механизм преобразования диалектов. EmitC предоставляет возможность конвертации множества различных операций из других диалектов MLIR в операции своего диалекта, что делает его важным инструментом для генерации программ на языках C и C++.

В утилите tsar каждая трансформация кода работает с операциями из диалекта EmitC или собственного диалекта `tsar`. Диалект `tsar` был создан для преодоления ограничений архитектуры EmitC и предназначен для управления процессом генерации программ, а также для описания сгенерированных программ. В нем используется специальная разметка, которая обеспечивает детерминированное изменение шаблонов, что необходимо для проверки конкретных видов анализа. Каждая трансформация добавляет в шаблон тег, описывающий тип анализа, что позволяет эффективно находить ошибки в новых мутациях.

Создание диалекта `tsar` позволило реализовать операции и атрибуты, которые не требуют аналогов в других диалектах, избегая при этом изменений в архитектуре EmitC. EmitC имеет свои задачи и направления развития, тогда как `tsar` сосредоточен на предоставлении механизмов для реализации запутываний, усложняющих внутреннее представление программ. Это достигается благодаря созданию уникальных атрибутов, операций и пассов, выполняющих соответствующие трансформации.

Трансляция MLIR шаблонов осуществляется передачей аргументов командной строки утилите tsar, отвечающих за включение конкретных мутаций. Мутации в свою очередь выполняют преобразования на основе разметки или известных паттернов в шаблоне. Затем результат мутации транслируется в запрашиваемый язык, например, C.

Для наглядности рассмотрим процесс генерации программы на языке C, используя утилиту tsar и шаблон на диалектах MLIR, представленного в листинге 1, в процессе генерации которого не включались мутации.

```
tsar.tags ["flow-sensitive"]
emitc.func @cwe369_div_zero_simple() -> () {
  %dividend = emitc.literal "42" : i32
  // Источник нуля
  %zero = tsar.transform_region() : () -> i32 {
    %zero = emitc.literal "0" : i32
    tsar.transform_out %zero : i32
  }
  // Блок, в котором фактически выполняется деление
  tsar.transform_region(%dividend, %zero) : (i32, i32) -> () {
    %data = "emitc.variable"() { value = 2 : i32 } : () -> !emitc.lvalue<i32>

    "emitc.assign"(%data, %zero) {
      tsar.source = #tsar.source<"CWE-369">,
      tsar.cfwrap = #tsar.cfwrap<"useless_if"> } : (!emitc.lvalue<i32>, i32) -> ()
    %data_load = emitc.load %data : !emitc.lvalue<i32>

    tsar.transform_region(%dividend, %data_load) {
      tsar.cfwrap = #tsar.cfwrap<"useless_if"> } : (i32, i32) -> () {
      // Операция деления
      %div_res = emitc.expression { tsar.sink = #tsar.sink<"CWE-369"> } : i32 {
        %div_res = emitc.div %dividend, %data_load : (i32, i32) -> i32
        emitc.yield %div_res: i32
      }

      %mask = emitc.literal "\"%d\\n\"" : !emitc.ptr<i8>
      emitc.call_opaque "printf"(%mask, %div_res) : (!emitc.ptr<i8>, i32) -> ()
    }
  }
  emitc.return
}
```

Листинг 1. Шаблон на диалектах MLIR на примере CWE-369 (Деление на ноль).
Listing 1. Template using MLIR dialects for CWE-369 (Divide by zero).

В листинге 1 представлен пример шаблона с описанием ошибки деления на ноль (CWE-369) на диалекте MLIR EmitC (префикс `emitc`) и разработанном диалекте `tsar`.

Для контроля над трансформациями, а также для описания сгенерированного кода в диалекте `tsar` реализованы следующие инструкции и атрибуты:

- `tsar.transform_region` – определяет регион, который необходимо рассматривать как единое целое при выполнении преобразований.
- `tsar.tags` – теги, которые представляют метаданные о конечном шаблоне.
- `#tsar.cfwrap<"useless_if">` – атрибуты такого вида указывают, какая оптимизация может быть применена к конкретной инструкции. В данном случае присваивание на 13 строке будет обернуто в `if(true)`.
- `#tsar.sink/source<"CWE-369">` – атрибуты, определяющие сток и исток ошибки. `"CWE-369"` является идентификатором ошибки.

Трансляция шаблона, представленного в листинге 1, в код на C приведена в листинге 2.

```
1   // TSAR_TAGS: flow-sensitive
2   void cwe369_div_zero_simple() {
3     int32_t v1 = 2;
4     // TSAR_SOURCE;CWE-369
5     v1 = 0;
6     int32_t v2 = v1;
7     // TSAR_SINK;CWE-369
8     int32_t v3 = 42 / v2;
9     printf("%d\n", v3);
10    return;
11  }
```

Листинг 2. Сгенерированная тестовая программа на языке C для CWE-369 (Деление на ноль).
Listing 2. Generated C test program for CWE-369 (Divide by zero).

Здесь комментарии описывают данные, необходимые для формирования информации об ошибке, а также о тестируемых методах анализа. `TSAR_SOURCE` и `TSAR_SINK` были получены в результате раскрытия атрибутов `#tsar.source` и `#tsar.sink` соответственно, а `TSAR_TAGS` из `tsar.tags`.

К ограничениям утилиты tsar можно отнести неполную поддержку конструкций языков. Так, например, нельзя объявить структуру и классы в языках C/C++/Java, но можно ссылаться на поля в известных структурах, а также нет возможности называть переменные более осмысленно.

Отметим, что в утилите tsar реализована трансляция в язык Java, применение которой к листингу 1 представлено в листинге 3. Однако в рамках данной работы сравнительное тестирование статических анализаторов, применимых к программам на языке Java, не будет рассмотрено, так как это тема для дальнейшего исследования.

Для расширения поддержки языков в инструменте TSAR мы рассмотрели три подхода, различающихся по сложности:

1. Создание диалектов, аналогичных EmitC, со своими операторами, атрибутами и транслятором. Этот метод предоставляет максимальную гибкость и расширяемость, позволяя учитывать все особенности конструкций языков, их типов и других аспектов. Однако он требует значительных усилий, так как для каждого нового языка необходимо создавать отдельный диалект.
2. Адаптация транслятора EmitC для операций языков Java, Go, Kotlin и другие. Этот подход проще, так как не требует создания новых диалектов. Однако он

ограничивается поддержкой лишь базовых конструкций, соответствующих подмножеству операций EmitC.

3. Добавление новых операций и атрибутов в диалект `tsar`. Этот вариант предполагает модификацию общей концепции, однако такой подход делает диалект более гибким и функциональным.

```
1   public class defaultWrapperClassName {
2   // TSAR_TAGS: flow-sensitive
3   public static void cwe369_div_zero_simple() {
4       int v1 = 2;
5       // TSAR_SOURCE;CWE-369
6       v1 = 0;
7       int v2 = v1;
8       // TSAR_SINK;CWE-369
9       int v3 = 42 / v2;
10      StdSupport.printf("%d\n", v3);
11      return;
12    }
13  }
```

Листинг 3. Сгенерированная тестовая программа на языке Java для CWE-369 (Деление на ноль).
Listing 3. Generated Java test program for CWE-369 (Divide by zero).

Каждый из перечисленных методов имеет свои преимущества и недостатки. Наиболее перспективным решением является использование гибридного подхода, объединяющего лучшие практики всех трёх методов. Это позволит разработать универсальную и эффективную инфраструктуру в рамках инструмента TSAR.

5. Трансформации кода

В данном разделе приведены примеры трансформаций, реализованных в инструменте TSAR.

В работах [20-22] рассматриваются различные техники обфускации (запутывания) исходного кода программ. Обфускация имеет значительное преимущество в том, что она сохраняет семантику исходной программы, одновременно усложняя анализ для статических и динамических анализаторов. Данное свойство является особенно важным при разработке трансформаций, которые должны будут сохранять семантику исходного шаблона.

В утилите tsar в настоящее время реализованы несколько техник трансформации исходного кода на основе предложений из вышеупомянутых работ:

- Замена литерала на переменную, содержащую значение литерала.
- Аутлайнинг – вынос участков кода в отдельную функцию. Для контроля над аутлайнингом используется механизм `tsar.transform_region`.
- Добавление явного приведения типа при присвоении данных переменной.
- Добавление бесполезного присваивания.
- Обертка инструкций в бесполезные инструкции потока управления: `if(true)`; `switch`, который всегда переходит на одну и ту же метку; `for`, выполняющий всегда одну итерацию цикла и т.д.
- Перестановка веток `then` и `else` в инструкции `if`.
- Вставка операций `goto`/`try-catch`.
- Замена переменной на сумму двух других переменных.

Трансформации реализованы в виде пассов в инфраструктуре MLIR. Реализация пассов в MLIR является ключевым аспектом для выполнения трансформаций и оптимизаций кода. Каждый пасс определяется как класс, наследуемый от базового класса `mlir::OperationPass`, с переопределением метода `runOnOperation`, который отвечает за выполнение логики трансформации на операциях. MLIR поддерживает множество диалектов, что позволяет создавать как универсальные, так и специализированные пассы, а возможность параметризации пассов обеспечивает гибкость в их настройке. Кроме того, MLIR предлагает инструменты для тестирования, что упрощает создание тестов для проверки корректности работы пассов и отладки. Также MLIR позволяет комбинировать несколько трансформаций в одном проходе компиляции.

На примере из листинга 1 проиллюстрирована работа трансформаций аутлайнинга, перестановки веток `if`, а также оборачивания инструкций в конструкции `if(true)`.

Для оборачивания инструкций в `if(true)` и аналогичные структуры применяется трансформация под названием `tsar-wrap-in-useless-cf`, которая оборачивает инструкции с атрибутом `#tsar.cfwrap<useless_if>`. Результаты применения указанных трансформаций к шаблону из листинга 1 представлены в листинге 4.

```
1  // TSAR_TAGS: flow-sensitive;path-sensitive
2  void cwe369_div_zero_simple() {
3    int32_t v1 = 2;
4    if (1) {
5      // TSAR_SOURCE;CWE-369;
6      v1 = 0;
7    }
8    int32_t v2 = v1;
9    if (1) {
10     // TSAR_SINK;CWE-369;
11     int32_t v3 = 42 / v2;
12     printf("%d\n", v3);
13   }
14   return;
15 }
```

Листинг 4. Шаблон из листинга 1 после применения tsar-wrap-in-useless-cf.
Listing 4. Template from listing 1 after applying tsar-wrap-in-useless-cf.

Заметим, что исток и сток ошибки были обернуты в конструкцию `if(1)` (строки 4 и 9), а в `TSAR_TAGS` был добавлен новый тег `path-sensitive`, который указывает, что анализатор должен поддерживать анализ, чувствительный к путям исполнения, для выявления ошибки.

Далее перейдем к последовательному применению трансформации, которая меняет блоки `then` и `else` в инструкции `if`. Эта трансформация, обозначенная как `tsar-invert-if-else`, приводит к результатам, представленным в листинге 5. В данном случае в `TSAR_TAGS` не произошло значительных изменений, поскольку тег `path-sensitive` уже был установлен.

Далее рассмотрим применение аутлайнинга к исходному шаблону из листинга 1, за выполнение которого в утилите tsar отвечает трансформация, обозначенная как `tsar-outline-transform-regions`. Пример работы демонстрируется в листинге 6.

В результате применения аутлайнинга исток и сток ошибки были перенесены в функции *cwe369_div_zero_simple_1* (строка 6) и *cwe369_div_zero_simple_2* (строка 13) соответственно. Кроме того, в `TSAR_TAGS` был добавлен новый тег inter-procedure, который указывает на необходимость поддержки межпроцедурного анализа в анализаторе.

6. Тестирование

В данном разделе рассматривается автоматизация генерации тестов на основе заданных шаблонов, а также представляется пример тестирования и сравнения различных анализаторов.

```
1   // TSAR_TAGS: flow-sensitive;path-sensitive
2   void cwe369_div_zero_simple() {
3     int32_t v1 = 2;
4     if (!1) {
5     } else {
6       // TSAR_SOURCE;CWE-369;
7       v1 = 0;
8     }
9     int32_t v2 = v1;
10    if (!1) {
11    } else {
12      // TSAR_SINK;CWE-369;
13      int32_t v3 = 42 / v2;
14      printf("%d\n", v3);
15    }
16    return;
17  }
```

Листинг 5. Последовательное применение tsar-invert-if-else к листингу 4.
Listing 5. Consecutive application of tsar-invert-if-else to listing 4.

```
1   // TSAR_TAGS: flow-sensitive;inter-procedure
2   static int32_t cwe369_div_zero_simple_0() {
3     return 0;
4   }
5
6   static void cwe369_div_zero_simple_1(int32_t v1, int32_t v2) {
7    // TSAR_SINK;CWE-369
8     int32_t v3 = v1 / v2;
9     printf("%d\n", v3);
10    return;
11  }
12
13  static void cwe369_div_zero_simple_2(int32_t v1, int32_t v2) {
14    int32_t v3 = 2;
15    // TSAR_SOURCE;CWE-369
16    v3 = v2;
17    int32_t v4 = v3;
18    cwe369_div_zero_simple_1(v1, v4);
19    return;
20  }
21
22  void cwe369_div_zero_simple() {
23    int32_t v1 = cwe369_div_zero_simple_0();
24    cwe369_div_zero_simple_2(42, v1);
25    return;
26  }
```

Листинг 6. Применение tsar-outline-transform-regions к шаблону из листинга 1.
Listing 6. Application of tsar-outline-transform-regions on template from listing 1.

В предыдущем разделе была проанализирована структура шаблона и механизмы применения трансформаций к нему. Теперь мы рассмотрим последовательное усложнение шаблона и генерацию множества тестовых программ из одного шаблона за счет применения различных

трансформаций. Для этого в исходный шаблон вводятся специальные комментарии для MLIR следующего формата:

- `// target: c,cpp` – языки, которые поддерживаются в данном шаблоне.
  `// @base@: --tsar-wrap-in-useless-cf @out1@` – применить к `@base@` трансформацию и записать результат в `@out1@` для последующих трансформаций.

Каждая строка комментария интерпретируется таким образом, что она генерирует тесты для всех языков, указанных в разделе `target`.

В конечном шаблоне данная разметка может выглядеть как показано в листинге 7.

```
1  // target: c, cpp
2
3  // @base@: -- @base@
4  // @base@: --tsar-wrap-in-useless-cf @out1@
5  // @base@: --tsar-outline-transform-regions @out2@
6
7  // @out1@: --tsar-invert-if-else @out3@
8  // @out3@: --tsar-wrap-in-useless-cf @out4@
9  // @out4@: --tsar-outline-transform-regions @out5@
10
11 // @out1@: --tsar-wrap-in-useless-cf @out6@
12 // @out6@: --tsar-invert-if-else @out7@
13 // @out7@: --tsar-outline-transform-regions @out8@
14
15 // @out2@: --tsar-wrap-in-useless-cf @out9@
16 // @out9@: --tsar-invert-if-else @out10@
17 // @out10@: --tsar-wrap-in-useless-cf @out11@
18 // @out9@: --tsar-wrap-in-useless-cf @out12@
19 // @out12@: --tsar-invert-if-else @end@
20
21 <MLIR TEMPLATE>
```

Листинг 7. Пример разметки для генерации в MLIR шаблоне.
Listing 7. Example of markup for interpretation in MLIR template.

В результате интерпретации указанных комментариев будет сгенерировано 14 программ для языка C и 14 программ для языка C++.

На текущий момент в TSAR представлены шаблоны для множества классов уязвимостей, таких как CWE-125 (Out-of-bounds read), CWE-134 (Use of Externally-Controlled Format String), CWE-369 (Divide by zero), CWE-401 (Missing Release of Memory after Effective Lifetime), CWE-415 (Double free), CWE-416 (Use after free), CWE-476 (Null pointer dereference), CWE-563 (Assignment to Variable without Use), CWE-763 (Release of Invalid Pointer or Reference), CWE-788 (Access of memory location after the end of buffer) и CWE-824 (Access of Uninitialized Pointer) в сумме составляющие более тысячи тестовых программ на языках C и C++.

Для оценки эффективности предложенного подхода были протестированы следующие анализаторы: clang-tidy [23] с включенными статическими детекторами Clang Static Analyzer (CSA) [7] и cppcheck [5]. Анализаторы запускались с применением собственных инструментов для анализа проектов, после чего результаты импортировались в утилиту metalyzer для дальнейшего сравнения. Сравнение проводилось путем отображения ошибок, выявленных каждым анализатором, в соответствующие им классы уязвимостей CWE. Отображение ошибок для cppcheck и clang-tidy было основано на работе [24], описывающее сопоставление предупреждений анализаторов на соответствующие CWE, а также вручную.

Под истинным срабатыванием (TP, True Positive) будем понимать, когда ожидаемая ошибка в конкретной тестовой программе совпадает с выводами анализатора. Точность вычислялась

как отношение истинных срабатываний (TP) к количеству тестов. Сравнение осуществлялось с учетом классов CWE, допустимых номеров строк, где могла быть ошибка, а также соответствующих файлов. Результаты тестирования представлены в табл. 1.

Из анализа представленных данных в таблице следует, что сделать однозначный вывод о превосходстве одного анализатора над другими невозможно. Все анализаторы демонстрировали наличие ошибок в определенных классах уязвимостей (CWE), и ни один из них не смог полноценно покрыть все представленные CWE.

Табл. 1. Сравнение статических анализаторов на тестовом наборе tsar.
Table 1. Static analyzers performance on TSAR benchmark suite.

CWE\Анализатор	cppcheck		clang-tidy + CSA		Количество тестов
	TP	Точность	TP	Точность	
CWE-125	56	49%	90	79%	114
CWE-134	0	0%	42	100%	42
CWE-369	122	32%	140	36%	384
CWE-401	13	37%	23	66%	35
CWE-415	24	10%	84	37%	229
CWE-416	26	39%	26	39%	66
CWE-476	2	5%	40	100%	40
CWE-563	110	100%	110	100%	110
CWE-763	29	21%	98	70%	140
CWE-788	54	96%	56	100%	56
CWE-824	126	100%	126	100%	126

Одной из уникальных особенностей тестового набора TSAR является наличие тегов, которые описывают конкретные сгенерированные программы. Это позволяет более точно оценить возможности статических анализаторов. В частности, при анализе CWE-369 на основании тегов было выявлено, что CSA не обнаружил уязвимость во всех программах, содержащих тег `float`. Данный факт указывает на ограничения анализа clang-tidy при обработке данных типа `float`. Пример теста, в котором был использован тег `float`, представлен в листинге 8.

```
1  // TSAR_TAGS: float;flow-sensitive
2  void cwe369_div_zero_simple_float() {
3    float v1 = 2.0F;
4    // TSAR_SOURCE;CWE-369
5    v1 = 0.0F;
6    float v2 = v1;
7    // TSAR_SINK;CWE-369
8    int32_t v3 = (int32_t) 42.0F / (int32_t) v2;
9    printf("%d\n", v3);
10   return;
11 }
```

Листинг 8. Тест с тегом float, на котором clang-tidy не нашел ошибку деления на ноль.
Listing 8. Test with float tag where clang-tidy was unable to find divide by zero.

Этот тест был получен путем изменения типа переменной с `int` на `float` в идентичных тестах. В то время как CSA успешно выявил ошибку во всех вариациях теста с тегом `int`, в случае с тегом `float` анализатор не сумел обнаружить ту же уязвимость. Это свидетельствует о потенциальных слабостях в механизмах анализа CSA, связанных с определенными типами данных.

В ходе анализа CWE-788 было выявлено, что cppcheck не справился с двумя межпроцедурными тестами. Единственным отличительным тегом в этих тестах был `general-type-in-outline`, который применял более общие типы данных вместо изначально заданных.

```
1   // TSAR_TAGS: inter-procedure;general-type-in-outline
2   static void cwe788_out_of_bounds_write_0(int *v1) {
3     // TSAR_SINK;CWE-788
4     strcpy(v1, "This string is too long for the buffer");
5     return;
6   }
7
8   void cwe788_out_of_bounds_write() {
9     // TSAR_SOURCE;CWE-788
10    int8_t v1[10];
11    cwe788_out_of_bounds_write_0(v1);
12    printf("%s\n", v1);
13    return;
14  }
```

Листинг 9. Тест с межпроцедурным анализом и использованием указателя общего типа.
Listing 9. Test with interprocedure analysis and more general pointer type.

Например, в листинге 9 на строке 3 в качестве типа аргумента функции использовался `int8_t *v1` вместо `int8_t v1[10]`. Это указывает на то, что cppcheck полагался на тип данных, а не на межпроцедурный анализ. Согласно стандарту C, компилятор не учитывает различия в типах указателей при передаче аргументов функции.

Также при анализе CWE-415 (Double Free) было выявлено, что CSA не обнаружил уязвимость во всех тестовых программах, содержащих тег `try-catch` (листинг 10).

```
1   // TSAR_TAGS: class;try-catch
2   class MyTask {};
3   int32_t cwe415_double_free() {
4     MyTask *task = nullptr;
5     try {
6       task = new MyTask;
7       delete task;
8       throw(1);
9     } catch (...) {
10      // TSAR_SINK;CWE-415
11      delete task;
12    }
13    return 0;
14  }
```

Листинг 10. Тест с тегом try-catch, на котором CSA не нашел ошибку повторного освобождения.
Listing 10. Test with try-catch tag where CSA was unable to find double free.

Текущий анализ в CSA не может [25] корректно обработать поведение механизма исключений C++ и их влияние на потоки выполнения и деструкторы. `throw` просто останавливает анализ во избежание ошибок и некорректных результатов.

7. Заключение

В данной статье представлен новый инструмент TSAR, предназначенный для оценки эффективности статических анализаторов. TSAR включает три ключевых компонента: шаблоны на MLIR, построенные вокруг различных классов уязвимостей из CWE; мутатор шаблонов (tsar), который выполняет трансформации шаблонов и генерирует тестовые программы на языках C, C++ и Java; утилита для сбора результатов анализа и сравнения анализаторов (metalyzer).

Инструмент TSAR направлен на выявление недостатков статических анализаторов, особенно в тех случаях, когда анализаторы не охватывают определенные методы анализа, такие как внутрипроцедурный и межпроцедурный анализ, чувствительный к путям выполнения анализ и анализ помеченных данных.

Тестирование различных статических анализаторов на тестах, сгенерированных на основе шаблонов CWE, позволили выявить конкретные недостатки в методах анализа и указать области для улучшения.

В рамках дальнейшей работы планируется увеличить в TSAR покрытие различных CWE, расширить количество доступных трансформаций, улучшить поддержку языка Java, а также сделать процесс написания шаблонов более доступным. Этого планируется достичь путем использования более высокоуровневых диалектов MLIR или даже языка программирования C, обеспечивая трансляцию до диалекта EmitC через ClangIR (промежуточное представление Clang, которое является диалектом MLIR для языков C/C++). Данные улучшения направлены на повышение точности и эффективности TSAR как инструмента для анализа и тестирования статических анализаторов, что, в свою очередь, должно способствовать улучшению качества тестируемого программного обеспечения.

Список литературы / References

[1]. CWE, Common Weakness Enumeration, https://cwe.mitre.org, accessed 01.12.2024.

[2]. Cuoq, P. et al. (2012). Testing Static Analyzers with Randomly Generated Programs. In: Goodloe, A.E., Person, S. (eds) NASA Formal Methods. NFM 2012. Lecture Notes in Computer Science, vol 7226. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-28891-3_12

[3]. Yang X. et al. Finding and understanding bugs in C compilers // Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation. – 2011. – С. 283-294.

[4]. Arusoaie A., Ciobâca S., Craciun V., Gavrilut D. and Lucanu D., "A Comparison of Open-Source Static Analysis Tools for Vulnerability Detection in C/C++ Code," 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, 2017, pp. 161-168, doi: 10.1109/SYNASC.2017.00035.

[5]. Marjamaki D., “Cppcheck - a tool for static c/c++ code analysis”, http://cppcheck.wiki.sourceforge.net/

[6]. Calcagno C. et al. Moving fast with software verification // NASA Formal Methods Symposium. – Cham : Springer International Publishing, 2015. – С. 3-11.

[7]. Clang, “Clang Static Analyzer”, https://clang-analyzer.llvm.org/

[8]. OCLint - A static source code analysis tool to improve quality and reduce defects for C, C++ and Objective-C, https://github.com/oclint/oclint, accessed 01.12.2024.

[9]. Shiraishi S., Mohan V., Marimuthu H. Test suites for benchmarks of static analysis tools // 2015 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW). – IEEE, 2015. – С. 12-15.

[10]. Christian Klinger, Maria Christakis, and Valentin Wüstholz. 2019. Differentially testing soundness and precision of program analyzers. In Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2019). Association for Computing Machinery, New York, NY, USA, 239–250. https://doi.org/10.1145/3293882.3330553

[11]. Groce A. et al., "Evaluating and Improving Static Analysis Tools Via Differential Mutation Analysis," 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), Hainan, China, 2021, pp. 207-218, doi: 10.1109/QRS54544.2021.00032.

[12]. Stephan Lipp, Sebastian Banescu, and Alexander Pretschner. 2022. An empirical study on the effectiveness of static C code analyzers for vulnerability detection. In Proceedings of the 31st ACM

SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2022). Association for Computing Machinery, New York, NY, USA, 544–555

[13]. The OpenSSF CVE Benchmark, https://github.com/ossf-cve-benchmark/ossf-cve-benchmark, accessed 01.12.2024.

[14]. Boland T., Black P. E. Juliet 1.1 C/C++ and java test suite // Computer. – 2012. – Т. 45. №. 10. С. 88-90.

[15]. saveourtool: Test framework for Static Analyzers and Compilers, https://github.com/saveourtool, accessed 01.12.2024.

[16]. ГОСТ Р 71207-2024 – Защита информации. Разработка безопасного программного обеспечения. Статический анализ программного обеспечения. Общие требования. https://protect.gost.ru/document1.aspx?control=31&baseC=6&page=3&month=2&year=2024&search=&id=257752

[17]. Lattner C. et al. MLIR: A compiler infrastructure for the end of Moore's law // arXiv preprint arXiv:2002.11054. – 2020.

[18]. Static Analysis Results Interchange Format (SARIF), https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html, accessed 01.12.2024.

[19]. 'emitc' Dialect to generate C/C++ from MLIR, https://mlir.llvm.org/docs/Dialects/EmitC/, accessed 01.12.2024.

[20]. Banescu S., Pretschner A. A tutorial on software obfuscation // Advances in Computers. – 2018. – Т. 108. – С. 283-353.

[21]. Jing D. Improvement of Vulnerable Code Dataset Based on Program Equivalence Transformation // Journal of Physics: Conference Series. – IOP Publishing, 2022. – Т. 2363. – №. 1. – С. 012010.

[22]. Li Y. et al. A closer look into transformer-based code intelligence through code transformation: Challenges and opportunities // arXiv preprint arXiv:2207.04285. – 2022.

[23]. Clang Tidy. https://clang.llvm.org/extra/clang-tidy/, accessed 01.12.2024.

[24]. Charoenwet W. et al. An empirical study of static analysis tools for secure code review // Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis. – 2024. – С. 691 - 703.

[25]. Treat C++ 'throw' as a sink jrose-apple // committed on Aug 18, 2012 https://github.com/llvm/llvm-project/commit/a4309c941c622f45f5fe58faaa0227a7f8b4da16, accessed 01.12.2024.

Информация об авторах / Information about authors

Кирилл Алексеевич ЧИБИСОВ – инженер отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Kirill Alekseevich CHIBISOV – engineer at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Рубен Артурович БУЧАЦКИЙ – кандидат технических наук, научный сотрудник отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Ruben Arturovich BUCHATSKIY – Cand. Sci. (Tech.), researcher at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Андрей Дмитриевич ТИМОНИН – лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Andrey Dmitrievich TIMONIN – laboratory assistant at Compiler Technology department of ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Владислав Игоревич ЛАЗАРЬ – студент МФТИ, Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Valdislav Igorevich LAZAR – a student at MIPT. Research interests: static analysis, compiler technologies, optimizations.

Дмитрий Михайлович ЖУРИХИН – старший научный сотрудник ИСП РАН, преподаватель МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, машинное обучение.

Dmitry Mikhailovich ZHURIKHIN – senior researcher at ISP RAS, lecturer at MSU. Research interests: static analysis, program optimization, machine learning.

Андрей Андреевич БЕЛЕВАНЦЕВ – доктор физико-математических наук, ведущий научный сотрудник ИСП РАН, профессор МГУ. Сфера научных интересов: статический анализ программ, оптимизация программ, параллельное программирование.

Andrey Andreevich BELEVANTSEV – Dr. Sci. (Phys.-Math.), Prof., leading researcher at ISP RAS, Professor at MSU. Research interests: static analysis, program optimization, parallel programming.