DOI: 10.15514/ISPRAS-2023-35(5)-12

Проверка программ на соответствие стандарту MISRA C с использованием инфраструктуры Clang

[1] Р.А. Бучацкий, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
[1,2] Я.А. Чуркин, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>
[1] К.А. Чибисов, ORCID: 0009-0008-4371-5475 <chibisov@ispras.ru>
[1] М.В. Пантилимонов, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>
[1,3] Е.В. Долгодворов, ORCID: 0000-0001-6962-6448 <krym4s@ispras.ru>
[1,3] А.В. Вязовцев, ORCID: 0009-0007-0826-2186 <andrey.vyazovtsev@ispras.ru>
[1] А.Г. Волохов, ORCID: 0009-0003-7797-4508 <alexeyvoh@ispras.ru>
[1,3] В.В. Трунов, ORCID: 0009-0001-9457-2610 <vtrunov@ispras.ru>
[4] Г.О. Миракян, ORCID: 0009-0001-2028-3496 <mirakyan.gayane@student.rau.am>
[1,3] К.Н. Китаев, ORCID: 0009-0004-9469-8100 <kitaev.konstantin@ispras.ru>
[1,2] А.А. Белеванцев, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>

[1] Институт системного программирования им. В.П. Иванникова РАН,
109004, Россия, г. Москва, ул. А. Солженицына, д. 25.
[2] Московский государственный университет имени М.В. Ломоносова,
119991, Россия, Москва, Ленинские горы, д. 1.
[3] Московский физико-технический институт,
141701, Московская область, г. Долгопрудный, Институтский переулок, д. 9.
[4] Российско-Армянский (Славянский) университет,
РА, г. Ереван, 0051, ул. О.Эмина 123.

Аннотация. MISRA C – это сборник правил и рекомендаций по программированию на языке C, который является фактическим стандартом в отраслях, где безопасность играет ключевую роль. Стандарт разработан консорциумом MISRA (Motor Industry Software Reliability Association) и включает в себя набор рекомендаций, которые позволяют использовать язык C для разработки безопасного, надежного и переносимого программного обеспечения. MISRA широко применяется во многих отраслях с высокими требованиями к надежности, включая аэрокосмическую, оборонную, автомобильную и медицинскую.

Мы разработали статические детекторы для проверки кода на соответствие рекомендациям стандарта безопасного кодирования MISRA C 2012. Средство проверки кода основано на компиляторной инфраструктуре LLVM/clang. В данной статье описываются стратегии, лежащие в основе проектирования и реализации детекторов. На тестовых примерах MISRA C предложенные детекторы с высокой точностью определяют соответствие или нарушение рекомендациям. Также детекторы показывают большее покрытие и лучшую скорость работы, чем Cppcheck, популярный статический анализатор с открытым исходным кодом.

Ключевые слова: MISRA; статический анализ; символьное выполнение; LLVM; Clang; Clang-Tidy; статический анализатор Clang.

Для цитирования: Бучацкий Р.А., Чуркин Я.А., Чибисов К.А., Пантилимонов М.В., Долгодворов Е.В., Вязовцев А.В., Волохов А.Г., Трунов В.В., Миракян Г.О., Китаев К.Н., Белеванцев А.А. Проверка программ на соответствие стандарту MISRA C с использованием инфраструктуры Clang. Труды ИСП РАН, том 35, вып. 5, 2023 г., стр. 169–192. DOI: 10.15514/ISPRAS–2023–35(5)–12.

Checking programs for compliance with MISRA C standard using the Clang framework

[1] R.A. Buchatskiy, ORCID: 0000-0001-8522-1811 <ruben@ispras.ru>
[1,2] Y.A. Churkin, ORCID: 0009-0000-5044-4249 <yan@ispras.ru>
[1] K.A. Chibisov, ORCID: 0009-0008-4371-5475 <chibisov@ispras.ru>
[1] M.V. Pantilimonov, ORCID: 0000-0003-2277-7155 <pantlimon@ispras.ru>
[1,3] E.V. Dolgodvorov, ORCID: 0000-0001-6962-6448 <krym4s@ispras.ru>
[1,3] A.V. Vyazovtsev, ORCID: 0009-0007-0826-2186 <andrey.vyazovtsev@ispras.ru>
[1] A.G. Volokhov, ORCID: 0009-0003-7797-4508 <alexeyvoh@ispras.ru>
[1,3] V.V. Trunov, ORCID: 0009-0001-9457-2610 <vtrunov@ispras.ru>
[4] G.H. Mirakyan, ORCID: 0009-0001-2028-3496 <mirakyan.gayane@student.rau.am>
[1,3] K.N. Kitaev, ORCID: 0009-0004-9469-8100 <kitaev.konstantin@ispras.ru>
[1,2] A.A. Belevantsev, ORCID: 0000-0003-2817-0397 <abel@ispras.ru>

[1] Ivannikov Institute for System Programming of the RAS,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.
[2] Lomonosov Moscow State University,
GSP-1, Leninskie Gory, Moscow, 119991, Russia.
[3] Moscow Institute of Physics and Technology,
9 Institutskiy per., Dolgoprudny, Moscow Region, 141701, Russia.
[4] Russian-Armenian (Slavonic) University,
0051, Republic of Armenia, Yerevan, Hovsep Emin str. 123.

Abstract. MISRA C is a collection of rules and recommendations for C programming language that is the de facto standard in industries where security plays the key role. The standard was developed by the MISRA (Motor Industry Software Reliability Association) consortium and includes a set of recommendations that allow the C language to be used to develop safe, reliable and portable software. MISRA is widely used in many industries with high reliability requirements, including aerospace, defense, automotive and medical.

We have developed static checkers to check code for compliance with MISRA C 2012 secure coding standard. The developed checkers are based on the LLVM/clang compiler infrastructure. This paper describes the strategies underlying the design and implementation of checkers. Using MISRA C 2012 example suite, the proposed checkers determine compliance or violation of the recommendations with high accuracy. The checkers also show greater coverage and better performance than Cppcheck, a popular open-source static analyzer.

Keywords: MISRA; static analysis; symbolic execution; LLVM; Clang; Clang-Tidy; Clang static analyzer.

For citation: Buchatskiy R.A., Churkin Y.A., Chibisov K.A., Pantilimonov M.V., Dolgodvorov E.V., Vyazovtsev A.V., Volokhov A.G., Trunov V.V., Mirakyan G.H., Kitaev K.N., Belevantsev A.A. Checking programs for compliance with MISRA C standard using the Clang framework. *Trudy ISP RAN/Proc. ISP RAS*, vol. 35, issue 5, 2023. pp. 169-192 (in Russian). DOI: 10.15514/ISPRAS-2023-35(5)-12.

1. Введение

Программное обеспечение (ПО) давно прошло стадию применения исключительно в исследовательских целях и заняло ключевую роль в современном мире. По мере увеличения размера и сложности разрабатываемого кода, вопросы надежности и безопасности стали важными проблемами при разработке ПО. Как следствие, проверка программ на отсутствие ошибок и уязвимостей, которые могут привести к отказу систем и даже человеческим жертвам, стала более сложной.

Язык C часто применяется при разработке ПО в критических системах (например, в контроллерах реактивных двигателей и медицинских системах). Несмотря на популярность он имеет плохую репутацию в контексте безопасного кодирования:

- синтаксис языка C предрасположен к совершению ошибок программистом, например, использование оператора присваивания (=) вместо оператора проверки на равенство (==) в конструкциях `if` и `while`;
- философия языка C предполагает, что программисты знают, что они делают: любая совершенная ошибка в коде может остаться незамеченной. Область, в которой C особенно слаб в этом отношении – это проверка типов. Например, является допустимой запись числа с плавающей запятой в целое, которое используется для представления значений `true` или `false`.

Использование языка C в системах, связанных с безопасностью, требует значительной осторожности. С целью обеспечения контроля над языками C/C++ организации начали разрабатывать и принимать стандарты безопасного кодирования, например, "Motor Industry Software Reliability Association C" (MISRA C) [1], его дальнейшие редакции MISRA C 2004, MISRA C++ 2008, MISRA C 2012, "Computer Emergency Response Team C" (CERT C) [2], "AUTomotive Open System ARchitecture" (AUTOSAR) [3]. Эти стандарты содержат правила и рекомендации по разработке безопасного, защищенного и надежного программного обеспечения и стали обязательными в соответствующих областях.

Для проверки соответствия кода безопасным стандартам кодирования применяются различные техники, из них выделим:

- проверка кода вручную: опытные разработчики или специалисты по безопасности просматривают исходный код, чтобы выявить потенциальные проблемы безопасности или нарушения стандартов безопасного кодирования. Этот процесс может занять много времени, но он эффективен для выявления многих типов уязвимостей;
- автоматический анализ кода: анализ с применением средств статического и/или динамического анализа. Автоматический анализ признан более эффективным, чем ручной, хотя бы потому, что он легко интегрируется в процесс разработки;
- аудиты соответствия: проводятся сторонними аудиторами или внутренними группами безопасности, чтобы гарантировать, что программное обеспечение разработано в соответствии со стандартами безопасного кодирования. Аудиты включают проверку документации, политик, процедур и кода для выявления любых несоответствий.

Данная работа посвящена разработке статических детекторов кода для программ на языке C для проверки их соответствия безопасному стандарту кодирования MISRA C 2012. Детекторы реализованы на основе компиляторной инфраструктуры LLVM [4] с использованием инструментов Clang-Tidy [5] и Clang Static Analyzer (CSA) [6].

Целью данной работы является реализация статических детекторов, проверяющих код на языке C на соответствие стандарту MISRA C 2012. Структура работы включает в себя обзор существующих решений (раздел 2), анализ стандарта MISRA C 2012 (раздел 3), разбор функциональных возможностей Clang-Tidy и CSA (раздел 4), примеры реализованных детекторов и трудностей, с которыми пришлось столкнуться при их разработке (раздел 5), тестирование (раздел 6) и заключение.

2. Обзор существующих решений

В настоящее время для проверки программ на соответствие стандартам безопасного кодирования в большинстве своем используются статические анализаторы, предоставляемые по коммерческим лицензиям, но есть небольшая часть с открытым исходным кодом.

В данном разделе приведено краткое описание существующих популярных открытых и коммерческих решений, приведена сводная таблица, отражающая поддержку стандартов безопасного кодирования в рассмотренных анализаторах, а также приведено сравнение анализаторов с открытым исходным кодом.

2.1 Анализаторы с открытым исходным кодом

Clang-Tidy и **CSA** – статические анализаторы с открытым исходном кодом, разрабатываемые как часть проекта LLVM [4] для анализа языков C/C++/Objective-C.

Clang-Tidy делает анализ на уровне абстрактного синтаксического дерева (AST) [7] программы и на уровне препроцессора, используя для этого инфраструктуру компилятора Clang. Clang-Tidy имеет поддержку множества стандартов кодирования (cert, cppcoreguidelines, hicpp и т.п.) и активную поддержку сообщества, которая добавляет поддержку как общепринятых стандартов кодирования, так и популярные практики написания кода.

CSA в настоящее время используется как дополнение к Clang-Tidy и запускается с использованием его инфраструктуры. Для анализа кода CSA использует символьное исполнение, что позволяет обнаружить ошибки в программах классов: "*деление на ноль*", "*использование ресурсов после освобождения*", "*недостижимый код*" и т.д.

Cppcheck [8] – статический анализатор с открытым исходным кодом для языков C/C++. Cppcheck очень популярная утилита из-за своей простоты и доступности. Несмотря на это, безопасные стандарты кодирования поддерживаются в нем либо по коммерческой лицензии, либо как расширение для исходного анализатора на языке Python, что негативно сказывается на качестве анализа и производительности данного анализатора.

SonarQube [9] – открытая платформа для анализа исходного кода на языках C/C++, C#, JavaScript, Python, Ruby, PHP, Swift, Ruby. Данный анализатор написан на языке Java и в первую очередь интегрируется с экосистемой данного языка. Данный анализатор имеет частичную поддержку стандартов MISRA C/C++ которые доступны только по коммерческой лицензии.

2.2 Коммерческие анализаторы

Coverity [10] – коммерческий анализатор, используемый для поиска уязвимостей в коде на языках C, C++, C#, Java, JavaScript, TypeScript, и Objective-C. Данный анализатор имеет интеграцию с интегрированными средами разработки (IDE) такими как Visual Studio, Intellij IDEA и Eclipse. Он покрывает большинство стандартов безопасного кодирования.

Klocwork [11] – коммерческий анализатор, используемый в первую очередь для поиска уязвимостей в исходном коде на языках C, C++, C#, Java, JavaScript и Python. В нем реализовано большинство стандартов безопасного кодирования.

PVS-Studio [12] – коммерческий анализатор, используемый в первую очередь для поиска уязвимостей в исходном коде на языках C, C++, C# и Java. Данный анализатор может использоваться некоммерческими проектами с открытым исходным кодом без лицензионного сбора. Он полностью поддерживает основные стандарты безопасного кодирования такие как MISRA C 2012 и AUTOSAR C++14.

2.3 Сравнение анализаторов

В табл. 1 представлено сравнение статических анализаторов по критерию поддержки стандартов безопасного кодирования. Из таблицы видно, что большинство анализаторов, поддерживающих стандарты безопасного кодирования, предоставляются по коммерческим лицензиям, что, в свою очередь, накладывает ограничения на их использование, не предоставляет легкой возможности для оценки качества реализованных детекторов независимыми специалистами по безопасности, а также расширения функционала детекторов и настройки под конкретный проект.

*Табл. 1. Поддержка стандартов безопасного кодирования в анализаторах C/C++ кода (символом * отмечены анализаторы с открытым исходном кодом, символом $ поддержка по коммерческой лицензии)*
*Table 1. C/C++ coding standards support in code analyzers (the * symbol indicates open source analyzers, the $ symbol indicates support under a commercial license)*

Стандарт / Анализатор	MISRA C++ 2008	MISRA C 2012	CERT	AUTOSAR
Coverity	Да	Да	Да	Да
Klocwork	Да	Да	Нет	Да
PVS-Studio	Частичная	Да	Да	Да
SonarQube*	Частичная$	Частичная$	Нет	Нет
Cppcheck*	Нет	Частичная	Частичная	Частичная$
Clang-Tidy / CSA*	Нет	Нет	Да	Нет

Статические анализаторы с открытым исходным кодом как правило предоставляют поддержку стандартов безопасного кодирования по коммерческим лицензиям. Так, например, анализатор SonarQube не имеет поддержки никаких безопасных стандартов кодирования в своей открытой версии, что сильно замедляет написание стандартов кодирования с нуля, так как сокращается возможность переиспользования кода в нем.

Cppcheck, который имеет частичную поддержку рекомендаций MISRA C 2012 и CERT C, использует для анализа AST и лексический анализ, чего в свою очередь недостаточно для проверки правил, требующих моделирование выполнения программы и оценки значения выражений. Также данный анализатор использует расширения на языке Python, что негативно сказывается на производительности анализа при большом количестве детекторов.

Clang-Tidy и CSA несмотря на то, что не имеют поддержки MISRA в отличие от Cppcheck, имеют активную поддержку сообщества, более мощные инструменты анализа, символьное исполнение и анализ на уровне AST, также реализуют множество стандартов кодирования, которые в некоторых местах брали за основу стандарты MISRA C/C++.

В данной работе за основу реализации детекторов для рекомендаций MISRA C 2012 были выбраны инструменты Clang-Tidy и CSA из компиляторной инфраструктуры LLVM.

3. Стандарт безопасного кодирования MISRA

Первая редакция стандарта MISRA C была опубликована в 1998 году [13] с целью удовлетворить потребность, возникшую в автомобильной промышленности, и отраженную в рекомендациях MISRA по разработке программного обеспечения для транспортных средств [14]. Первая редакция имела большой успех и стала применяться за пределами автомобильного сектора. Более широкое использование стандарта в отрасли и сообщаемый опыт разработчиков ПО побудили к серьезной переработке MISRA C, в результате которой в 2004 году было опубликовано второе издание.

В 2013 году было опубликовано третье издание – MISRA C 2012. В MISRA C 2012 внесены многочисленные улучшения по сравнению с предыдущим изданием: расширена поддержка стандарта языка C; в дополнение к C90 добавлена поддержка C99; рекомендации и правила были определены более четко и точно, и, как следствие, значительно возросла вероятность того, что разные инструменты статического анализа могут дать одинаковые результаты.

В настоящее время стандарт MISRA C 2012 включает ***175*** *рекомендаций* (с учетом расширений Amendment 1 [15] и 2 [16]). Каждая рекомендация классифицируется как *директива* или *правило*.

Правило – это рекомендация, для которой предоставлено полное описание требования. Должна быть возможность проверить соответствие исходного кода правилу без необходимости использования какой-либо дополнительной информации. Инструменты статического анализа должны быть способны проверять соответствие правилам.

Директива – это рекомендация, для которой невозможно дать полное описание, необходимое для проведения проверки на соответствие. Для проведения проверки необходима дополнительная информация, которая может быть предоставлена в проектной документации или спецификациях требований к исходному коду. Инструменты статического анализа могут помочь в проверке соблюдения директив, но разные инструменты могут по-разному интерпретировать то, что представляет собой несоответствие кода директиве.

Каждому правилу в свою очередь присвоена своя категория:

- *Обязательное* – правило должно выполняться без каких-либо исключений в любом коде, заявляющий соответствие данным стандартам безопасного кодирования.
- *Необходимое* – правило должно выполняться, но код может иметь задокументированное формальное отклонение.
- *Рекомендательное* – данное правило не обязательно для реализации, чтобы заявить соответствие кода безопасному стандарту кодирования.

При реализации инструментов статического анализа приоритет расставляется согласно значимости категорий: обязательные, затем необходимые, и лишь потом рекомендательные.

Стандарт MISRA C 2012 также предоставляет информацию о том, как должна осуществляться проверка на соответствие кода рекомендации: автоматически или полуавтоматически. Из этого следует, что рекомендации, которые проверяются полуавтоматически, должны предоставлять настройки своих эвристик.

Стоит отметить, что стандарт MISRA активно применяется многими международными компаниями в автомобильной индустрии: General Motors, Ford Motor Company, Volvo Cars, Bosch, Renault, BMW Group.

4. Статический анализ в LLVM

LLVM [4] – компиляторная инфраструктура с открытым исходным кодом, предоставляющая большое число переиспользуемых компонентов для разработки компиляторов. LLVM предоставляет широкие возможности для анализа программ, написанных на языках C/C++/Objective-C посредством компилятора Clang.

Clang содержит в себе множество C/C++ библиотек, позволяющих работать с исходным кодом на разных стадиях компиляции, среди них можно выделить следующие:

- `libclangLex` – библиотека для работы с лексическим анализатором, которая в свою очередь включает работу с препроцессором;
- `libclangAST` – библиотека для работы с AST;
- `libclangASTMatchers` – библиотека, содержащая подготовленные, часто используемые шаблоны для поиска в AST;

- `libclangAnalysis` – библиотека с часто используемыми утилитами для анализа кода на базе AST. В ней содержатся абстракции для работы с графом вызовов, базовый анализ достижимости кода, изменяемости переменных.

Данные библиотеки используются как фундамент для статических анализаторов кода на базе LLVM: Clang-Tidy и CSA.

4.1 Clang-Tidy

Clang-Tidy [5] – инструмент статического анализа, разработанный для проверки C/C++/Objective-C кода на соответствие различным стандартам кодирования и общепринятым практикам, детекторы которых сгруппированы как модуль `ClangTidyModule`.

`ClangTidyModule` – набор детекторов, имеющих общий класс, например *cert*. Детекторы не изолированы внутри модуля и могут наследовать реализации из других `ClangTidyModule`. Такое переиспользование возможно, так как каждый детектор должен реализовать интерфейс `ClangTidyCheck`, который представлен на листинге 1.

```
class ClangTidyCheck {
public:
  virtual void registerMatchers (ast_matchers::MatchFinder *Finder) {}
  virtual void check (const ast_matchers::MatchFinder::MatchResult &Result) {}
  virtual void registerPPCallbacks (const SourceManager &SM,
                                    Preprocessor *PP,
                                    Preprocessor *ModuleExpanderPP) {}
}
```

Листинг 1. Интерфейс класса ClangTidyCheck.
Listing 1. ClangTidyCheck class interface.

- `registerMatchers` – отвечает за регистрацию шаблонов поиска в AST, которые записывает в `Finder`;
- `check` – это обратный вызов, как результат обработки шаблонов из `registerMatchers`. Данный метод вызывается для каждого совпадения в AST и передает интересующие узлы через параметр `Result`;
- `registerPPCallbacks` – функция, аналогичная `registerMatchers`, но для препроцессора. Обратные вызовы добавляются через методы *PP*.

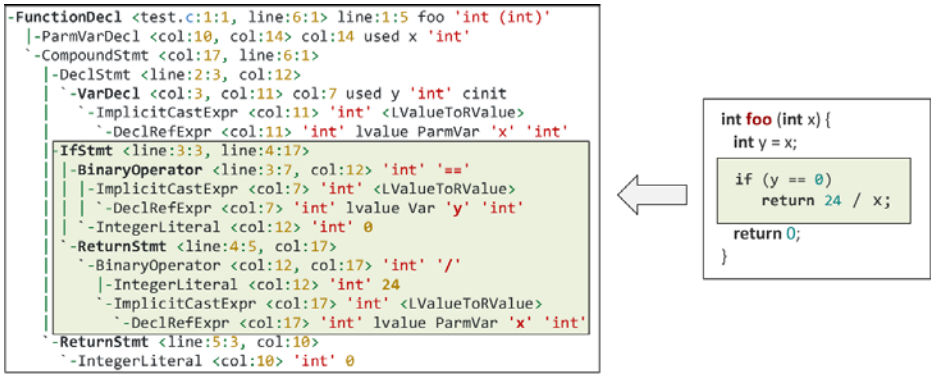
Детекторы для Clang-Tidy можно реализовать на уровне препроцессора с помощью `PPCallbacks` или на уровне AST с помощью шаблонов поиска (AST Matchers) [17].

Абстрактное синтаксическое дерево (AST) Clang [7] строится после обработки исходного кода препроцессором и представляет собой структуру, создаваемую фронтендом компилятора и служащую промежуточным представлением программы, используемым Clang. Генерация бинарного кода осуществляется на основе AST.

AST Clang содержит полную информацию об исходном коде программы: каждый узел дерева хранит местоположение в исходном коде до и после обработки препроцессором. Это делает AST пригодным для использования в качестве наиболее простой основы для анализа кода. На листинге 2 показан пример AST представления, полученного с помощью компилятора *clang*. Левая часть – это внутреннее представление дерева для соответствующего кода справа.

Шаблоны поиска AST в Clang-Tidy [17] – это интерфейс для декларативного определения участков кода, которые нужно найти, и выполнения определенных действий над каждым найденным шаблоном. В качестве примера шаблона поиска AST рассмотрим следующее определение: `ifStmt(has(binaryOperator(hasOperatorName("==")))).bind("if")`.

В данном примере выполняется поиск в AST оператора `"if"`, который содержит в условии бинарный оператор сравнения `"=="` (листинг 2). Команда сопоставления `bind(...)` присваивает имя узлу AST, к которому она применяется, для дальнейшего использования.



Листинг 2. Пример AST представления для кода на языке C.
Listing 2. AST representation of function written in C language.

В Clang-Tidy также доступен анализ на уровне графа потока управления (CFG), который построен «на уровне исходного кода» программы и состоит из указателей на операторы AST, упорядоченные в порядке потока управления (т. е. в порядке исполнения).

Существенным ограничением Clang-Tidy является невозможность одновременного анализа сразу нескольких единиц трансляции. В некоторых ситуациях информации только на уровне AST или только в рамках одной единицы трансляции бывает недостаточно.

4.2 Clang Static Analyzer

Clang Static Analyzer [6] – статический анализатор для проверки C/C++/Objective-C кода, использующий метод символьного исполнения. Этот метод предполагает присвоение символьных значений переменным программы и разбиение всех возможных состояний программы на классы. Для анализа используется структура данных под названием *разобранный граф* (далее РГ) (exploded graph) – это один из известных способов организации чувствительного к путям анализа. Анализатор интерпретирует все возможные пути исполнения в графе потока управления. Каждый путь в данном направленном графе – множество узлов вида `(ProgramPoint, ProgramState)`.

`ProgramPoint` – участок программы до и после оператора в потоке управления.

`ProgramState` – абстрактное состояние программы, содержащее текущие значения переменных, оценку значений для выражений, ограничения на значения неизвестных переменных, области памяти, отображающие содержимое переменных, а также нетипизированное хранилище данных – `GenericDataMap` (далее GDM), принадлежащее пользовательским детекторам, а также позволяющее хранить составные типы данных, необходимые для анализа, например, список указателей на одну и ту же память в детекторе, который анализирует алиасы и так далее. Состояния меняются между `ProgramPoint`.

Ребра в РГ между узлами означают, что `ProgramPoint` первого узла корректирует его `ProgramState` до `ProgramState` из второго узла и переводит программу в точку выполнения второго узла.

В CSA, также, как и Clang-Tidy анализ происходит в рамках одной единицы трансляции. В ней ищутся функции, являющиеся корневыми, то есть, теми, которые не вызываются в других

местах, в рамках этой единицы трансляции. Для каждой такой корневой функции инициализируется начальное состояние `ProgramState`, который, затем, в результате анализа операторов и выражений меняется. Анализатор симулирует выполнение условных выражений, таких как оператор `"if"`, путем создания новых узлов и ребер в разобранном графе для истинного и ложного результата с учетом текущего абстрактного состояния.

В CSA применяется кэширование узлов графа для снижения сложности его обхода. Если будет сгенерирован новый узел с тем же состоянием и программной точкой, что у существующего узла, путь кэшируется, и переиспользуется существующий узел.

Взаимодействие анализатора с конечными детекторами осуществляется посредством паттерна "Посетитель" (*Visitor*) и использования идиомы языка `C++` *Curiously Recurring Template Pattern* (CRTP) [18], часть интерфейса которого представлена на листинге 3.

```
class CSAChecker: public Checker<check::PreCall, check::PostCall,
                                 check::Location, check::Bind,
                                 check::BranchCondition, check::PointerEscape,
                                 check::EndAnalysis> {
public:
  void checkPreCall(const CallEvent &Call, CheckerContext &C) const {}
  void checkPostCall(const CallEvent &Call, CheckerContext &C) const {}
  void checkLocation(SVal Loc, bool IsLoad, const Stmt *S,
                     CheckerContext &) const {}
  void checkEndAnalysis(ExplodedGraph &G,
                        BugReporter &BR,
                        ExprEngine &Eng) const {}
}
```

Листинг 3. Пример интерфейса детектора CSA.
Listing 3. CSA checker interface example.

- `checkPreCall` – вызывается сразу перед входом в функцию;
- `checkPostCall` – вызывается сразу после выхода из функции;
- `checkLocation` – вызывается при каждом доступе области памяти `Loc`;
- `checkEndAnalysis` – вызывается при завершении анализа, результирующий граф находится в `G`.

Каждый детектор в процессе такого обхода может добавлять ребра в РГ посредством обновления состояния `ProgramState` в контексте анализатора (`CheckerContext`). Если детектор обнаруживает ошибку в коде, то сообщение о ней будет содержать путь анализатора до этой ошибки: какие ветки и условия он выбирал, ограничения на переменные, участвующие в пути, сколько итераций сделал в цикле, и т.д.

В анализаторе также реализована модель памяти, позволяющая запоминать конкретные и символьные значения определенных областей памяти (*memory regions*) и получать к ним доступ в любой момент анализа.

Частью CSA являются предсказатели (*assumers*), позволяющие накладывать ограничения на значения переменных в процессе символьного выполнения, например – `if ( i != 0 ) func(i);`, в данном примере из диапазона значений переменной `i` будет выколота точка `0`, в `ProgramPoint`, соответствующей точки выполнения после оператора `if ( i != 0 )`. Это позволяет реализовывать детекторы, например, проверяющие переполнение типа. Стоит отметить, что предсказатели действуют достаточно консервативно, например, если о символе ничего не известно, то диапазон его значений будет определен как полный диапазон значений типа. Анализатору можно подсказывать о диапазонах значений переменных с помощью определенных методов, например, `assumeSymNE` из модуля *RangeConstraintManager*, который предполагает неравенство символа определенному переданному значению. Методы

данного типа изменяют информацию о диапазонах значений символов в состоянии и возвращают обновленное состояние.

Также ядром CSA поддерживаются различные типы диагностик: упрощенная диагностика (`BasicBugReport` – только сама диагностика и подсказки); чувствительная к путям диагностика (`PathSensitiveBugReport` – сама диагностика, подсказки и путь от корневой функции к месту, в котором она была выдана).

CSA, используя анализ графа потока управления и символьное выполнение, эмулирует вызов вложенных функций. Это позволяет осуществлять межпроцедурный анализ, чего нельзя сказать про Clang-Tidy.

В заключение стоит отметить, что для анализа на уровне AST, не требующего межпроцедурность и символьное выполнение, Clang-Tidy является хорошим выбором, в силу простоты реализации детекторов и скорости анализа. В ином случае, стоит обратить внимание на CSA. Также стоит отметить, что помимо межпроцедурного, в CSA была добавлена поддержка межмодульного анализа, но на данный момент она плохо масштабируема на большие программы.

5 Разработка детекторов для MISRA C

За основу реализации детекторов для рекомендаций MISRA C 2012 была выбрана инфраструктура открытых анализаторов Clang-Tidy и CSA, работа каждого из которых была разобрана в разделе 4.

В рамках данной работы были реализованы детекторы для 139 рекомендаций MISRA C 2012 из которых 122 детектора для Clang-Tidy и 17 для CSA. Из оставшихся 36 рекомендаций 27 покрываются уже имеющимися в Clang детекторами или диагностиками (в том числе ошибками компиляции), 3 рекомендации явно требуют анализа в рамках множества единиц трансляций (что не возможно реализовать с помощью Clang), а 6 рекомендаций являются директивами и сильно зависят от структуры конкретного проекта. Совокупное покрытие стандарта MISRA C 2012 разработанными детекторами и уже имеющимися в Clang диагностиками составляет 95%.

В Clang-Tidy у каждого детектора есть уникальное имя. Детекторы для запуска можно выбрать с помощью опции `-checks=`, которая задает разделенный запятыми список включаемых и отключаемых (с префиксом `-`) детекторов. С помощью Clang-Tidy также можно запускать детекторы CSA. Ниже представлена команда для запуска реализованных детекторов Clang-Tidy и CSA для проверки кода на соответствие стандарту MISRA C 2012:

```
clang-tidy -checks="-*, misra-c-2012*, clang-analyzer-misra*" main.c
```

Здесь опция `-checks=` отключает все детекторы по умолчанию (`-*`), включает все детекторы Clang-Tidy (`misra-c-2012*`) и детекторы CSA (`clang-analyzer-misra*`).

В случае обнаружения детектором несоответствия кода стандарту MISRA C 2012 выдается предупреждение в виде диагностики с соответствующим сообщением и с указанием места в исходном коде.

5.1 Пример детекторов на Clang-Tidy

Рассмотрим примеры реализации детекторов для правил MISRA C 2012 с использованием инфраструктуры Clang-Tidy для которых будет достаточно поиска шаблонов на AST и лексического анализа. Большинство реализованных детекторов для правил MISRA C 2012 не встретило сложностей, пример такого детектора представлен в разделе 5.1.1. Из сложностей можно отметить аккуратную работу с шаблоном поиска `findAll`, которая может существенно снижать производительность на больших проектах. Для решения этой проблемы был предложен определенный подход, который описан в разделе 5.1.2.

5.1.1 Правило 15.2

Правило 15.2 из стандарта MISRA C 2012 звучит следующим образом:

> «Оператор *goto* должен переходить к метке, объявленной позже в той же функции»

На листинге 4 приведен пример кода на языке C с конструкциями как нарушающими данное правило, так и соответствующими ему:

```
void rule15_2() {
    int J = 0;
L1:
    ++J;
    if (10 == J) {
        goto L2; // Нарушения нет
    }
    goto L1;     // Нарушение - переход к метке, объявленной до оператора goto.
L2:
    ++J;

L3: goto L3;   // Нарушение - переход к метке, объявленной до оператора goto.
}
```

Листинг 4. Иллюстрация различных аспектов правила MISRA C 2012 15.2.
Listing 4. Code markup according to the MISRA C 2012 Rule 15.2.

Рассмотрим возможную реализацию детектора для данного правила с использованием Clang-Tidy. Для реализации данного правила достаточно анализа AST представления для поиска узлов оператора «goto».

В разделе 4.1 был описан интерфейс детекторов Clang-Tidy. Реализация методов `check` и `registerMatchers` детектора для данного правила представлена на листинге 5:

```
1   void LabelAfterGotoCheck::registerMatchers(MatchFinder *Finder) {
2     Finder->addMatcher(gotoStmt().bind("goto"), this);
3   }
4
5   void LabelAfterGotoCheck::check(const MatchFinder::MatchResult &Result) {
6     const auto *MatchedGoto = Result.Nodes.getNodeAs<GotoStmt>("goto");
7     const auto &SM = Result.SourceManager;
8
9     SourceLocation GotoLoc = MatchedGoto->getBeginLoc();
10    SourceLocation LabelLoc = MatchedGoto->getLabel()->getBeginLoc();
11    // Check that `goto` location is presumed before `label` location
12    if (SM->getFileLoc(GotoLoc) <= SM->getFileLoc(LabelLoc))
13      return;
14
15    diag(GotoLoc, "Оператор goto должен переходить к метке, "
16                  "объявленной позже в той же функции");
17    diag(LabelLoc, "Метка объявлена здесь", DiagnosticIDs::Note);
18  }
```

Листинг 5. Возможная реализация детектора для правила MISRA C 2012 15.2 в Clang Tidy.
Listing 5. One of the possible MISRA C 2012 Rule 15.2 checker implementations in Clang Tidy.

Функция `registerMatchers` регистрирует шаблон поиска `gotoStmt`. Функция `check` получает шаблон, найденный в AST дереве в виде экземпляра класса `GotoStmt`, находит смещение оператора `goto` в файле и смещение соответствующей метки (строки 9-10) и сравнивает их (строка 12). В случае, если смещение метки находится до смещения оператора `goto`, то выдается предупреждение (строки 15-17).

5.1.2 Правило 8.9

Далее приведем пример правила с более сложной реализацией детектора. Рассмотрим правило 8.9 из стандарта MISRA C 2012, которое звучит следующим образом:

> «Объект должен быть объявлен в теле функции, если его использование встречается только в одной функции»

Возможный вариант кода, нарушающего правило 8.9, представлен на листинге 6:

```
int X; // Нарушение - переменная используется только в функции 'foo'
extern char Y; // Нарушение - переменная используется только в функции 'foo'

static int Z; // Нарушения нет - переменная используется как в функциях 'foo' и 'bar'
char A; // Нарушения нет - переменная не используется ни в одной функции

void foo() {
  int I = 0; // Нарушения нет - переменная используется только в функции 'foo'
             // и определена в ней же
  for (I = 0; I < X; ++I, ++Z, ++Y) {
  ...
  }
}

void bar() {
  Z++;
}
```

Листинг 6. Иллюстрация различных аспектов правила MISRA C 2012 8.9.
Listing 6. Code markup according to the MISRA C 2012 Rule 8.9.

Первоначальная реализация детектор для этого правила представлена на листинге 7.

В “наивной реализации” для каждого глобального объявления переменной (`VarDecl`) ищутся все ее использования (`DeclRefExpr`) во всей единице трансляции (параметр `*Result.Context->getTranslationUnitDecl()`) с помощью функции `match`, с использованием шаблона поиска `findAll` (строки 9-14), который ищет все использования вложенного в него шаблона поиска. Такая конструкция может существенно снижать производительность на проектах с большим количеством глобальных переменных. Каждое использование рассматривается вместе с функцией, в которой оно находится. Далее производится подсчет количества функций, в которых используется рассматриваемая глобальная переменная, если такая функция всего одна, то выдается предупреждение.

При тестировании разработанного детектора на реальных проектах был выявлен ряд проблем, связанных с производительностью на больших единицах трансляции и ошибок в диагностике глобальных переменных, использование которых встречается вне функций. Данный аспект не был учтен при первоначальном проектировании детектора.

Обновленная реализация детектора представлена на листинге 8.

В обновленной реализации отбор всех функций, в которых наблюдается использование глобальных переменных, теперь происходит при регистрации шаблонов поиска (метод `registerMatchers`, строки 7-10). Был использован шаблон `forEachDescendant`, необходимый для связывания *“родительского объявления”* с объявлением каждой глобальной переменной, которая в его определении была использована. Соответственно метод `check` будет вызываться для каждой пары *“родительское объявление - глобальная переменная”*. Таким родительским объявлением может быть объявление глобальной переменной с инициализатором или определение функции. В методе `check` происходит лишь сбор информации в словарь `VDToFDs`, представляющий собой отображение определений глобальных переменных в множество определений функций, в которых они были использованы. Проверка количества использований глобальных переменных в функциях и выдача предупреждающей была перенесена в конец обработки единицы трансляции (метод

`onEndOfTranslationUnit`, строки 33-50), когда информация обо всех глобальных переменных и функциях была собрана.

```
1  void ObjectBlockScopeCheck::registerMatchers(MatchFinder *Finder) {
2    Finder->addMatcher(varDecl(unless(common::isLocal())).bind("vd"), this);
3  }
4
5  void ObjectBlockScopeCheck::check(const MatchFinder::MatchResult &Result) {
6    const auto *MatchedDecl = Result.Nodes.getNodeAs<VarDecl>("vd");
7
8    // Find all `DeclRefExpr`s related to matched variable and its parent functions
9    const auto DREs = match(
10       findAll(functionDecl(hasDescendant(
11                                 declRefExpr(to(varDecl(equalsNode(MatchedDecl))))
12                                     .bind("dre")))
13                   .bind("parent-function")),
14       *Result.Context->getTranslationUnitDecl(), *Result.Context);
15   // If there is no variable usage then it is compliant case
16   if (DREs.empty())
17     return;
18
19   // To store first `FunctionDecl` to compare it with another
20   const FunctionDecl *FirstFD = nullptr;
21   // To store at least one `DeclRefExpr` in each function where
22   // variable usage is present
23   llvm::DenseMap<const FunctionDecl *, SourceLocation> DRELocs;
24
25   for (const auto &DRE : DREs) {
26     const auto *FD = DRE.getNodeAs<FunctionDecl>("parent-function");
27     const auto *D = DRE.getNodeAs<DeclRefExpr>("dre");
28
29     if (!DRELocs.count(FD))
30       DRELocs[FD] = D->getExprLoc();
31
32     if (!FirstFD)
33       FirstFD = FD;
34     // If there is more than one function with matched variable usage
35     // then it is compliant case
36     else if (FirstFD != FD)
37       return;
38   }
39
40   diag(MatchedDecl->getLocation(),
41        "Объект должен быть объявлен в теле функции, если его "
42        "использование  встречается только в одной функции");
43   for (const auto &DRLoc : DRELocs) {
44     diag(DRLoc.second, "`%0` используется только в функции `%1`",
45          DiagnosticIDs::Note)
46         << MatchedDecl->getNameAsString() << DRLoc.first->getNameAsString();
47   }
48 }
```

Листинг 7. Первоначальная реализация детектора для правила MISRA C 2012 8.9 в Clang Tidy.
Listing 7. Initial implementation of checker for MISRA C 2012 Rule 8.9 in Clang Tidy.

Если глобальные переменные используются вне функций (например, в определении других глобальных переменных), то их необходимо исключить из списка рассматриваемых детектором, так как перенесение определения глобальной переменной в данном случае в тело функции влечет за собой ошибки компиляции и изменения семантики исходного кода. Этот аспект, как было указано выше, не был учтен в первом варианте реализации, соответственно были ложные срабатывания детектора. В обновленной реализации была добавлена проверка того, что сопоставленное *“родительское объявление”* не является функцией. В таком случае происходит добавление определения данной глобальной переменной в список `BlackListedVDs` (строка 20), чтобы исключить данную глобальную переменную из списка потенциальных кандидатов для выдачи предупреждений.


```
1  void ObjectBlockScopeCheck::registerMatchers(MatchFinder *Finder) {
2    const auto VD = varDecl(unless(common::isLocal()), unless(parmVarDecl()),
3                            unless(isImplicit()))
4                       .bind("vd");
5
6    // Match all `Decl`s that have a global scope and contains references to globals
7    Finder->addMatcher(
8        decl(unless(isImplicit()), unless(hasAncestor(functionDecl())),
9             forEachDescendant(declRefExpr(to(VD)))).bind("decl"),
10       this);
11 }
12
13 void ObjectBlockScopeCheck::check(const MatchFinder::MatchResult &Result) {
14   const auto *VD = Result.Nodes.getNodeAs<VarDecl>("vd");
15   const auto *FD = Result.Nodes.getNodeAs<FunctionDecl>("decl");
16
17   // If `VarDecl` was used not at function add it at blacklist
18   // to not check it at end of translation unit
19   if (!FD) {
20     BlackListedVDs.insert(VD);
21     return;
22   }
23
24   // Store `VarDecl` and it's `FunctionDecl` where it was used
25   if (!BlackListedVDs.count(VD)) {
26     if (VDToFDs.count(VD))
27       VDToFDs[VD].insert(FD);
28     else
29       VDToFDs[VD] = {FD};
30   }
31 }
32
33 void ObjectBlockScopeCheck::onEndOfTranslationUnit() {
34   for (const auto &VDToFD : VDToFDs) {
35     const auto *VD = VDToFD.first;
36     const auto FDs = VDToFD.second;
37
38     // Skip if variable is in blacklist or there is more then one function
39     // where variable is used
40     if (BlackListedVDs.count(VD) || FDs.size() > 1)
41       continue;
42
43     diag(VD->getLocation(),
44          "Объект должен быть объявлен в теле функции, если его "
45          "использование  встречается только в одной функции");
46     const auto *SingleFD = *FDs.begin();
47     diag(SingleFD->getLocation(), "`%0` используется только в функции `%1`",
48          DiagnosticIDs::Note) << VD;
49   }
50 }
```

Листинг 8. Итоговая реализация детектора для правила MISRA C 2012 8.9 в Clang Tidy.
Listing 8. Final implementation of checker for MISRA C 2012 Rule 8.9 in Clang Tidy.

5.2 Пример детектора CSA

В данном разделе будут рассмотрены примеры реализации детекторов с использованием инфраструктуры Clang Static Analyzer для правил MISRA C 2012 для которых требуется символьное исполнение. В разделе 5.2.1 приводится пример реализации детектора с использованием анализа регионов памяти. В разделе 5.2.2 приводится пример детектора, при реализации которого использовался анализ помеченных данных, доступный в CSA.

5.2.1 Правило 18.3

Для иллюстрации реализации простого CSA детектора рассмотрим правило 18.3 из стандарта MISRA C 2012, которое звучит следующим образом:

> «Реляционные операторы (>, >=, <, <=) не должны применяться к указателям на объекты, кроме тех случаев, когда они указывают на один и тот же объект.

Возможный вариант кода, нарушающего правило 18.3, представлен на листинге 9.

```
void rule18_3 ( int32_t param ) {
  int32_t a[10], b[10];
  int32_t *p;
  if ( param ) {
    p = a;
    if (p < b) {  // Нарушение - указатели относятся к разным объектам
      …
    }
  } else {
    p = b;
    if (p < b) {  // Нарушения нет - указатель p указывает на ту же память, что и b
      …
    }
  }
}
```

Листинг 9 Иллюстрация различных аспектов правила MISRA C 2012 18.3.
Listing 9. Code markup according to the MISRA C 2012 Rule 18.3.

Из примеров видно, что необходим анализ потока данных, недоступный в Clang-Tidy.
Реализация детектора для правила 18.3 с использованием CSA представлена на листинге 10.

```
1  void PointerRelationsChecker::checkPreStmt(const BinaryOperator *BO,
2                                             CheckerContext &C) const {
3    if (!BO->isRelationalOp())
4      return;
5
6    QualType LHSType = BO->getLHS()->IgnoreCasts()->getType();
7    QualType RHSType = BO->getRHS()->IgnoreCasts()->getType();
8    if ((!LHSType->isPointerType() && !LHSType->isArrayType()) ||
9        (!RHSType->isPointerType() && !RHSType->isArrayType()))
10     return;
11
12   const auto *LeftBase = C.getState()
13                          ->getSVal(BO->getLHS(), C.getLocationContext())
14                          .getAsRegion()
15                          ->getBaseRegion();
16   const auto *RightBase = C.getState()
17                           ->getSVal(BO->getRHS(), C.getLocationContext())
18                           .getAsRegion()
19                           ->getBaseRegion();
20
21   if (LeftBase != RightBase)
22     reportWarning(C, BO);
23   return;
24 }
```

Листинг 10. Реализация детектора для правила MISRA C 2012 18.3 в Clang Static Analyzer.
Listing 10. Implementation of checker for MISRA C 2012 Rule 18.3 in Clang Static Analyzer.

Для реализации детектора был использован единственный обработчик (`checkPreStmt`), перехватывающий бинарные операторы (`BinaryOperator`, строка 1). В обработчике проверяется, что бинарный оператор является оператором сравнения (строка 3), затем проверяется, что типы левой и правой частей оператора являются адресными (указателями или массивами, строки 8-9).
Из примеров на листинге 9 видно, что детектору правила 18.3 необходим анализ алиасов. CSA предоставляет возможность анализа памяти символов исследуемого исходного кода с помощью отображения каждого символа на предполагаемый регион памяти. Регионы могут быть разных типов, например, таких, как куча (`HeapSpaceRegion`), статическая память

(`StackSpaceRegion`) и т.д. Этой функциональности достаточно для корректной работы детектора в большинстве случаев. Точность анализа можно улучшить, если реализовать алгоритм анализа алиасов из модуля `LLVM Alias Analysis` [19] средствами CSA.
На листинге 10, для проверки того, что указатели являются алиасами, извлекаются их базовые регионы памяти (строки 12-19) и затем сравниваются (строка 21). Если регионы памяти не эквивалентны, то выдается предупреждающая диагностика (строка 22).

5.2.2 Правило 22.7

Рассмотрим правило 22.7 из стандарта MISRA C 2012, для которого потребовалась более сложная реализация детектора. Правило звучит следующим образом:

> «Макроопределение `EOF` должно сравниваться только с неизменным возвращаемым значением функции стандартной библиотеки, которая может его вернуть»

Возможный вариант кода, нарушающего правило 22.7, представлен на листинге 11.

```
void rule22_7 ( void ) {
  int c = getchar();
  if (EOF == c) {  // Нет нарушения
    ...
  }

  if (EOF == (char)c) {  // Нарушение - сужающее приведение типов приводит к изменению
    …                    // значения с точки зрения правила
  }
}
```

Листинг 11 Иллюстрация различных аспектов правила MISRA C 2012 22.7.
Listing 11. Code markup according to the MISRA C 2012 Rule 22.7.

Правило предполагает, что не должно быть сужающих преобразований типа для значений, возвращенных функциями, которые могут вернуть `EOF`.
Детектор для данного правила должен отслеживать возвращаемые значения вызовов функций, которые могут вернуть `EOF` с точки зрения стандарта языка `C`, а также их прямое изменение, посредством записи в переменную, содержащую возвращенное значение, и косвенное, посредством приведения типов.
Для реализации детектора для данного правила был применен анализ помеченных данных (*taint analysis*), доступный в CSA. Символьное значение считается помеченным (*tainted*), если известно, что оно было получено из «ненадежного» источника, например, путем чтения стандартного ввода или файлового дескриптора, или из переменных окружения. Анализ помеченных данных – это эффективный метод обнаружения дефектов безопасности, основанный на обнаружении использования помеченных значений в вызовах чувствительных функций.
Рассмотрим подробную реализацию обработчиков детектора (листинг 12).
При реализации детектора были использованы следующие обработчики – `checkPostCall`, `checkLocation`, `checkPostStmt` и `checkPreStmt`.
Функция `mayRetEOF` (строка 2) в обработчике `checkPostCall`, который используется для отслеживания вызовов функций, проверяет принадлежность вызванной функции заранее составленному списку имен библиотечных функций, которые могут вернуть `EOF`, таких, как, например, `getchar` из библиотеки *`stdio.h`*. Если проверка успешна, то возвращенное значение и выражение вызова функции сохраняются в контейнере состояния (вызов `addCallExprToSym`, строка 7). Затем возвращенное значение помечается для его дальнейшего отслеживания (вызов `addTaint`, строка 8).

```
1  void EOFCompUnmodResChecker::checkPostCall(const CallEvent &Call, CheckerContext &Ctx) const {
2    if (!mayRetEOF(Call))
3      return;
4
5    ProgramStateRef State = Ctx.getState();
6    SymbolRef RetVal = Call.getReturnValue();
7    State = addCallExprToSym(RetVal, Call.getOriginExpr(), State);
8    Ctx.addTransition(taint::addTaint(State, RetVal, PointerToEOFRet));
9  }
10
11 void EOFCompUnmodResChecker::checkLocation(SVal Loc, bool IsLoad, const Expr *E,
12                                            CheckerContext &C) const {
13   ProgramStateRef State = Ctx.getState();
14   if (!IsLoad && taint::isTainted(State, Loc, PointerToEOFRet))
15     Ctx.addTransition(addExprToSym(Loc, E, State));
16 }
17
18 void EOFCompUnmodResChecker::checkPostStmt(const CastExpr *Exp,   CheckerContext &Ctx) const {
19   const Expr *CastedExpr = Exp->getSubExpr();
20   if (Exp->getType().getUnqualifiedType() ==
21       CastedExpr->getType().getUnqualifiedType())
22     return;
23
24   ProgramStateRef State = Ctx.getState();
25   const SVal CExpVal = Ctx.getSVal(Exp);
26   if (taint::isTainted(State, CExpVal, PointerToEOFRet))
27     Ctx.addTransition(addExprToSym(CExpVal, Exp, State));
28 }
```

Листинг 12. Реализация обработчиков checkPostCall, checkLocation и checkPostStmt детектора правила MISRA C 2012 22.7 в Clang Static Analyzer.
Listing 12. checkPostCall, checkLocation and checkPostStmt handlers implementation for MISRA C 2012 Rule 22.7 checker in Clang Static Analyzer.

На листинге 13 представлена структура использованного контейнера состояния, представляющего собой отображение символов на выражения, в которых данные, относящиеся к ним, изменяются:

```
1  REGISTER_SET_FACTORY_WITH_PROGRAMSTATE(ExprSet, const Stmt *)
2  REGISTER_MAP_WITH_PROGRAMSTATE(ChangeExprs, const SymExpr *, ExprSet)
3  REGISTER_MAP_WITH_PROGRAMSTATE(EOFFuncCallExprs, const SymExpr *, const CallExpr *)
```

Листинг 13. Объявление контейнера для детектора правила MISRA C 2012 22.7 в CSA.
Listing 13. Container declaration for MISRA C 2012 22.7 Rule checker in Clang Static Analyzer.

Библиотека CSA предоставляет несколько видов нетипизированных хранилищ данных (*GDM*), таких как список (`SET`), словарь (`MAP`), линейный динамический массив (`LIST`) и переменная (`TRAIT`), в которых можно хранить пользовательские данные. Каждое изменение контейнера привязывается напрямую к текущему состоянию, соответственно, все состояния, порожденные из текущего, наследуют все изменения контейнера. Библиотека CSA также предоставляет возможность создания фабрик, позволяющих описывать структуры контейнеров с более сложными типами данных. На листинге 13 создана фабрика (строка 1) для хранения списка выражений, ключом для которой является символьное выражение в словаре `ChangeExprs` ниже (строка 2). В строке 3 определен словарь `EOFFuncCallExprs`, содержащий выражения вызова функций, возвращающих `EOF`.

На листинге 12 обработчик `checkLocation` используется для сбора информации об изменениях помеченных символов (строка 13) при прямой записи в них. Далее выражение добавляется в контейнер состояния (строка 15). В обработчике `checkPostStmt`, который используется для сбора информации об изменениях помеченных символов посредством приведения типов, происходит проверка (строки 20-21), что приводимый тип отличается от

исходного, чтобы исключить случаи ложных срабатываний в местах бесполезных приведений типов, затем проверяется, что символ был помечен, а значит был возвращен функцией, которая может вернуть `EOF` (строка 26). Если символ был помечен, то выражение приведения типа добавляется в контейнер состояния (строка 27) для дальнейшего анализа и выдачи диагностики. Необходимо обратить внимание на первый параметр обработчика, который имеет тип `const CastExp*`, означает, что обработчик будет срабатывать только на выражениях приведения типа, что существенно уменьшает количество его вызовов.

Алгоритм реализации обработчика `checkPreStmt`, который используется для идентификации сравнения помеченных переменных со значением `EOF` и выдачу предупреждения, представлен на рис. 4. Обработчик перехватывает только бинарные операторы. После проверки, что бинарный оператор представляет собой оператор сравнения с `EOF`, извлекается символ, который с `EOF` сравнивается. Затем происходит отбор всех выражений, в которых данный символ изменялся и выдача диагностики с подсказками.

Algorithm 1 Алгоритм выдачи диагностического предупреждения и подсказок в месте сравнения с EOF

Input
EOFFuncCallExprs - словарь, содержащий отображения возвращенного значения к выражению вызова, возвращающего EOF
ChangeExprs - словарь, содержащий отображения символов к выражениям их порождающим
BOp - выражение бинарного оператора
Output
Выдача диагностического предупреждения и подсказок в месте сравнения с EOF

```
if isRelationalOp(BOp) is false then                          ▷ Проверка, что оператор является реляционным
    return
end if
if isComparisonWithEOF(BOp) is false then                     ▷ Проверка, что в операторе есть сравнение с EOF
    return
end if
RelOpSym ← getSymFromRelOp(BOp)                               ▷ Извлечение символа, который сравнивается с EOF
EOFCallExpr ← None
for each Sym in getSymDependenceList(RelOpSym) do             ▷ Перебор зависящих символов
    if Sym in EOFFuncCallExprs then                           ▷ Поиск символа среди всех, которые были возвращены EOF функциями
        EOFCallExpr ← getExprBySym(RelOpSym, EOFFuncCallExprs)
        break
    end if
end for
if EOFCallExpr is None then
    return
end if
NoteList                                                      ▷ Инициализация пустого листа подсказок
for each Sym in getSymDependenceList(RelOpSym) do             ▷ Перебор зависящих символов
    if Sym in SymToExprs then                                 ▷ Поиск выражения порождающего символ
        ChangeExpr ← getExprBySym(Sym, ChangeExprs)           ▷ Извлечение выражения, которое изменяет символ
        addNoteToNoteList(ChangeExpr,"Symbol was changed here", NoteList)   ▷ Генерация подсказки
    end if                                                    ▷ в месте изменяющего символ выражения
end for
if isEmpty(NoteList) is true then
    return
end if
addNoteToNoteList(EOFCallExpr,"Symbol was produced here.")    ▷ Генерация подсказки в месте вызова EOF функции
generateWarn(BOp, NoteList)                                   ▷ Генерация предупреждающей диагностики с подсказками в месте сравнения с EOF
```

Рис. 4. Алгоритм обработчика checkPreStmt для детектора правила MISRA C 2012 22.7 в CSA.
Fig. 4. checkPreStmt algorithm for Clang Static Analyzer MISRA C 2012 Rule 22.7 checker.

6. Тестирование

Для оценки качества реализованных детекторов использовались синтетические примеры из стандарта безопасного кодирования MISRA C 2012 [20], а также следующие проекты с открытым исходным кодом:

- zlib [21] версии 1.2.11 – свободная кроссплатформенная библиотека для сжатия данных;

- openjpeg [22] версии 2.5.0 – свободная библиотека для кодирования и декодирования JPEG 2000 изображений;
- openssl [23] версии 3.0.2 – полноценная криптографическая библиотека с открытым исходным кодом;
- coreJSON [24] версии 3.2.0 – библиотека, соответствующая стандарту кодирования MISRA C 2012, для работы с данными в формате JSON.

Тестирование осуществлялось на компьютере с 4-ядерным процессором Intel E3- 1265LV2 с ограничением тактовой частоты в 2.6 ГГц под управлением операционной системы Ubuntu 20.04.5 (Focal Fossa), объем оперативной памяти 32GB. Для запуска и сбора результатов анализа использовалась инфраструктура статического анализа CodeChecker [25].
На проектах с открытым исходным кодом качество анализа оценивалось путем ручной разметки результатов анализа и подсчета точности, которая вычислялась как отношение истинных срабатываний к сумме истинных (TP) и ложных (FP) рассмотренных срабатываний. Так как срабатываний на проектах с открытым исходным кодом крайне много, то детекторы с большим числом срабатываний оценивались по 100 случайным срабатываниям в различных файлах проекта. Результаты представлены в табл 2.

Табл. 2. Результаты анализа разработанных детекторов для Clang-Tidy / CSA на проектах с открытым исходным кодом
Table 2. Results of the analysis of developed checkers for Clang-Tidy / CSA on open source projects

Проект	**Срабатываний**			**Точность**	**Время анализа**
	Всего	Рассмотренных	FP		
zlib	8388	1034	0	100%	33 сек
openjpeg	20371	1433	9	99.4%	5 мин
openssl	275528	2229	7	99.7%	49 мин, 29 сек
coreJSON	7	7	0	100%	10 сек

Из результатов анализа следует, что разработанные детекторы имеют высокую точность и скорость работы. Большая часть ложных срабатываний связана с детекторами, использующими модуль Clang-Tidy – *MutationAnalyzer*, предназначенный для поиска мутаций переменных в переданном контексте. *MutationAnalyzer* использует достаточно неконсервативный подход к поиску мутаций, например, оператор взятия адреса (&) по умолчанию трактуется как мутация, что не всегда верно. Часть ложноположительных срабатываний связана с CSA детекторами, использующими модуль *RangeConstraintManager*, в котором есть проблема с определениями диапазонов значений символов при расширяющих и сужающих преобразованиях типов.
Также было проведено сравнение реализованных детекторов с анализатором Cppcheck (раздел 2.1) на открытых проектах coreJSON и zlib. Стоит отметить, что coreJSON разрабатывался с учетом стандарта MISRA C 2012, за некоторыми задокументированными исключениями [26]. Соответствие кода стандарту MISRA разработчики coreJSON проверяют коммерческим статическим анализом Coverity (раздел 2.2). При тестировании coreJSON срабатывания детекторов на правилах из исключений не учитывались.
В табл. 3 представлены результаты анализа проекта coreJSON.

Табл. 3. Сравнение разработанных детекторов для Clang-Tidy/CSA с Cppcheck на проекте coreJSON
Table 3. Comparison of developed checkers for Clang-Tidy/CSA witch Cppcheck on coreJSON project

Анализатор	Clang-Tidy/CSA			Cppcheck		
Срабатываний / Правило	TP	FP	FN	TP	FP	FN
10.3	1	0	0	1	1	1
10.4	0	0	0	7	7	0
12.3	0	0	0	11	11	0
13.5	5	0	0	0	0	5
20.9	0	0	0	1	1	0
20.12	1	0	0	0	0	1

На проекте coreJSON с помощью разработанных детекторов были найдены несоответствия кода правилам 10.3, 13.5 и 20.12, которые не были найдены анализатором Coverity.
В качестве примера рассмотрим срабатывание разработанного детектора для правила MISRA C 2012 13.5, которое звучит следующим образом:

> «Правый операнд логического оператора && или || не должен содержать устойчивых побочных эффектов»

Стандарт языка C гласит:

1) вычисление логических операторов && и || происходит слева направо;
2) вычисление логического оператора прекращается, когда результат может быть определен.

Побочный эффект считается *устойчивым* в определенной точке выполнения, если он может повлиять на состояние выполнения программы в этой точке.
Следующие побочные эффекты являются устойчивыми в определенной точке программы:

- изменение файла;
- изменение объекта;
- доступ к *volatile* объекту.

На листинге 14 представлен отрывок исходного кода coreJSON, на котором произошло срабатывание детектора. В данном коде возможен случай, при котором функции `skipAnyString` или `skipAnyLiteral` возвращают значение `true` в операторе `if` (строки 15-17). В таком случае вычисление логических операторов прекращается и правый операнд оператора || – функция `skipNumber` не будет вызвана (строка 17). Вызов функции `skipNumber` может иметь побочный эффект, заключающийся в изменении значения переменной `start` по указателю: `*start = i;` (строка 7). Этот побочный эффект является *устойчивым*: значение указателя `start` далее используется в функции `skipArrayScalars` (строка 34), что может не соответствовать ожиданиям разработчика.
О найденных несоответствиях стандарту MISRA C 2012 было сообщено разработчикам coreJSON в результате чего были внесены соответствующие исправления в код [27].
В табл 4. приведены результаты анализа проекта zlib для правил, результат анализа которых отличался между анализаторами Clang-Tidy/CSA и Cppcheck.

```
1  static bool skipNumber( const char * buf, size_t * start, size_t max ) {
2      bool ret = false;
3      size_t i;
4      ...
5      if( ret == true )
6      {
7          *start = i;      // устойчивый побочный эффект
8      }
9      return ret;
10 }
11
12 static bool skipAnyScalar( const char * buf,  size_t * start, size_t max ) {
13     bool ret = false;
14
15     if( ( skipString( buf, start, max ) == true ) ||
16         ( skipAnyLiteral( buf, start, max ) == true ) ||
17         ( skipNumber( buf, start, max ) == true ) )  // Нарушение правила 13.5
18     {
19         ret = true;
20     }
21     return ret;
22 }
23
24 static void skipArrayScalars( const char * buf, size_t * start, size_t max ) {
25     size_t i;
26     ...
27     while( i < max )
28     {
29         if( skipAnyScalar( buf, &i, max ) != true )
30         {
31             break;
32         }
33     }
34     *start = i;
35 }
```

Листинг 14. Фрагмент кода coreJSON с найденным несоответствием правилу 13.5.
Listing 14. coreJSON code snippet showing non-compliance with Rule 13.5.

Табл. 4. Сравнение разработанных детекторов для Clang-Tidy/CSA с Cppcheck на проекте zlib
Table 4. Comparison of results for developed Clang-Tidy/CSA checkers and Cppcheck on zlib project

Анализатор	Clang-Tidy/CSA			Cppcheck		
Срабатываний / Правило	TP	FP	FN	TP	FP	FN
7.2	1172	0	0	61	0	1111
10.3	98	0	0	20	0	78
15.7	15	0	0	11	0	4
16.3	38	0	0	13	0	25
12.3	26	0	0	30	24	20

На проекте zlib (34 файла) время анализа Cppcheck составляет 42 секунды, а разработанные детекторы на основе Clang Tidy / CSA анализируют тот же проект за 33 секунды. Разница в производительности составляет около 27%. Также стоит отметить существенное количество ложноотрицательных (FN) и ложноположительных (FP) срабатываний в Cppcheck, которые обусловлены некачественным разбором AST. Так, например, анализ правила 15.7 не учитывает конструкции «`if ... else if`» без «{», а ложноположительные срабатывания на правиле 12.3 обусловлены некорректной интерпретацией бинарного оператора «запятая»

(,) – запятая-разделитель в списке объявлений множества переменных не является бинарным оператором с точки зрения стандарта языка C.

Также было проведено сравнительное тестирование разработанных детекторов и анализатора Cppcheck на синтетических примерах MISRA C 2012 [20] для правил, которые классифицируются стандартом как *обязательные*. Тестирование показало, что разработанные детекторы на основе Clang-Tidy / CSA не имеют ложных срабатываний на синтетических примерах MISRA в отличие от Cppcheck, поэтому являются эффективнее с точки зрения точности анализа инструмента Cppcheck даже на *обязательных* правилах.

7. Заключение

В данной работе представлены разработанные статические детекторы для проверки кода на соответствие рекомендациям стандарта безопасного кодирования MISRA C 2012. Статические детекторы реализованы на основе инфраструктуры анализаторов Clang-Tidy и Clang Static Analyzer из компиляторной инфраструктуры LLVM. На синтетических примерах разработанные детекторы не имели ложных срабатываний, а на представленных проектах с открытым исходным кодом точность анализа составила порядка 99%. По сравнению с анализатором Cppcheck разработанные детекторы дают более точные и полные результаты, а также высокую скорость анализа.

В рамках дальнейшей работы планируется разработать опцию компилятора *clang*, которая позволит компилировать, проверяя на полное соответствие исходного кода стандарту MISRA C 2012, а также разработка статических детекторов для проверки правил из стандартов AUTOSAR C++14.

Также планируется провести сравнительное тестирование разработанных детекторов с коммерческими анализаторами, поддерживающими проверку кода на соответствие стандарту MISRA C 2012.

Список литературы / References

[1]. MIRSA official website. https://www.misra.org.uk/, accessed 01.11.2023.
[2]. SEI CERT C Coding Standard. https://wiki.sei.cmu.edu/confluence/display/c, accessed 01.11.2023.
[3]. AUTOSAR official website. https://www.autosar.org/, accessed 01.11.2023.
[4]. The LLVM Compiler Infrastructure. https://llvm.org/, accessed 01.11.2023.
[5]. Clang Tidy. https://clang.llvm.org/extra/clang-tidy/, accessed 01.11.2023.
[6]. Clang Static Analyzer. https://clang.llvm.org/docs/ClangStaticAnalyzer.html, accessed 01.11.2023.
[7]. Introduction to the Clang AST. https://clang.llvm.org/docs/IntroductionToTheClangAST.html, accessed 01.11.2023.
[8]. Cppcheck A tool for static C/C++ code analysis. http://cppcheck.net/, accessed 01.11.2023.
[9]. SonarQube. https://www.sonarsource.com/products/sonarqube/, accessed 01.11.2023.
[10]. Coverity Static Analysis. https://www.synopsys.com/software-integrity/static-analysis-tools-sast/coverity.html, accessed 01.11.2023.
[11]. Klocwork static analyzer. https://www.perforce.com/products/klocwork, accessed 01.11.2023.
[12]. PVS-Studio static analysis system. https://pvs-studio.com/en/, accessed 01.11.2023.
[13]. Motor Industry Software Reliability Association, MISRA-C:1998, Guidelines for the use of the C language in vehicle based software. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jul. 1998.
[14]. The Motor Industry Research Association, Development Guidelines For Vehicle Based Software. Nuneaton, Warwickshire CV10 0TU, UK: The Motor Industry Research Association, Nov. 1994.
[15]. MISRA, MISRA C:2012 Amendment 1 – Additional security guidelines for MISRA C:2012. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Apr. 2016.
[16]. MISRA, MISRA C:2012 Addendum 2 – Coverage of MISRA C:2012 (including Amendment 1) against ISO/IEC TS 17961:2013 “C Secure”

[17]. AST Matcher Reference, https://clang.llvm.org/docs/LibASTMatchersReference.html, accessed 01.11.2023.
[18]. CRTP pattern. https://en.cppreference.com/w/cpp/language/crtp, accessed 01.11.2023.
[19]. LLVM Alias Analysis Infrastructure, https://llvm.org/docs/AliasAnalysis.html, accessed 01.11.2023.
[20]. MISRA-C-2012 Example Suite, https://gitlab.com/MISRA/MISRA-C/MISRA-C-2012/Example-Suite, accessed 01.11.2023.
[21]. Библиотека zlib. http://zlib.net/, accessed 01.11.2023.
[22]. Библиотека openjpeg. http://www.openjpeg.org/, accessed 01.11.2023.
[23]. Библиотека openssl. https://www.openssl.org/, accessed 01.11.2023.
[24]. Библиотека coreJSON. https://github.com/freertos/corejson, accessed 01.11.2023.
[25]. Инфраструктура статического анализа CodeChecker. https://codechecker.readthedocs.io/en/latest/, accessed 01.11.2023.
[26]. coreJSON: MISRA Compliance, https://github.com/FreeRTOS/coreJSON/blob/main/MISRA.md, accessed 01.11.2023.
[27]. coreJSON: Fix short-circuiting operations with side-effects, https://github.com/FreeRTOS/coreJSON/pull/148, accessed 01.11.2023.

Информация об авторах / Information about authors

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 in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Ян Андреевич ЧУРКИН – стажер-исследователь отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Yan Andreevich CHURKIN – researcher in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

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

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

Михаил Вячеславович ПАНТИЛИМОНОВ – научный сотрудник отдела компиляторных технологий. Научные интересы: статический анализ, компиляторные технологии, СУБД.

Mikhail Vyacheslavovich PANTILIMONOV – researcher in Compiler Technology department. Research interests: static analysis, compiler technologies, DBMS.

Егор Викторович ДОЛГОДВОРОВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Egor Viktorovich DOLGODVOROV – a student at MIPT, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Андрей Викторович ВЯЗОВЦЕВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Andrey Viktorovich VYAZOVTSEV – a student at MIPT, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Алексей Георгиевич ВОЛОХОВ – ведущий инженер отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Aleksey Georgievich VOLOKHOV – leading engineer in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Владимир Владимирович ТРУНОВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Vladimir Vladimirovich TRUNOV – a student at MIPT, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

Гаяне Оганнесовна МИРАКЯН – студентка Российско-Армянский Университета. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Gayane Hovhannes MIRAKYAN – a student at Russian-Armenian University. Research interests: static analysis, compiler technologies, optimizations.

Константин Николаевич КИТАЕВ – студент МФТИ, лаборант отдела компиляторных технологий ИСП РАН. Научные интересы: статический анализ программ, компиляторные технологии, оптимизации.

Konstantin Nikolaevich KITAEV – a student at MIPT, laboratory assistant in Compiler Technology department at ISP RAS. Research interests: static analysis, compiler technologies, optimizations.

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.