ИНФОРМАТИКА, ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА И УПРАВЛЕНИЕ

УДК 004.4'233
DOI 10.52684/2312-3702-2023-48-2-73-76

ОБЗОР СУЩЕСТВУЮЩИХ РЕШЕНИЙ ДЛЯ СТАТИЧЕСКОГО АНАЛИЗА КОДА GOLANG

Р. М. Галиев*, *О. И. Евдошенко

Галиев Раушан Минеязович, магистрант, МИРЭА – Российский технологический университет, г. Москва, Российская Федерация; e-mail: galiev.r.m@edu.mirea.ru;

Евдошенко Олег Игоревич, кандидат технических наук, доцент кафедры индустриального программирования, МИРЭА – Российский технологический университет, г. Москва, Российская Федерация, тел.: + 7 (989) 683-69-79; e-mail: evdoshenko@mirea.ru

В работе представлен обзор существующих решений для статического анализа кода языка Go, описаны сложности в его автоматическом тестировании, а также выявлены наиболее перспективные анализаторы для Golang. Для того чтобы понять, какое соотношения на рынке готовых инструментов статического анализа Golang, в статье формируется карта отношений готовых решений, которая кластеризуется на подгруппы: анализаторы; полностью входящие в агрегаторы; агрегаторы; анализаторы, не входящие в агрегаторы. Далее отсеиваются анализаторы, непригодные для работы и выявления наиболее перспективных анализаторов из оставшихся, детально исследуется их работа и выдаваемые ошибки с помощью популярного редактор текста, написанного на Golang. В итоге определяются три анализатора, комбинация из которых дает все доступные для автоматической проверки кода ошибки в проектах на Golang.

Ключевые слова: *код, Golang, анализатор, агрегатор, автоматическая проверка кода.*

REVIEW OF EXISTING SOLUTIONS FOR STATIC ANALYSIS OF GOLANG CODE

R. M. Galiyev, O. I. Yevdoshenko

Galiyev Raushan Mineyazovich, graduate student, MIREA – Russian Technological University, Moscow, Russian Federation; e-mail: galiev.r.m@edu.mirea.ru;

Yevdoshenko Oleg Igorevich, Candidate of Technical Sciences, Associate Professor the Department of Industrial Programming, MIREA – Russian Technological University, Moscow, Russian Federation, phone: + 7 (989) 683-69-79; e-mail: evdoshenko@mirea.ru

The paper provides an overview of existing solutions for static code analysis of the Golang, describes the difficulties in automatic code testing, and identifies the most promising analyzers for Golang. In order to understand the relationship in the market of ready-made Golang static analysis tools, the article forms a relationship map of ready-made solutions, which is clustered into subgroups: analyzers; fully included in aggregators; aggregators; analyzers not included in aggregators. Next, the analyzers that are unsuitable for work are screened out and to identify the most promising analyzers from the remaining ones, the work of the analyzers and the errors they produce are examined in detail using the popular text editor written in Golang. As a result, three analyzers are determined, the combination of which gives all available for automatic error code verification in Golang projects.

Keywords: *code, Golang, analyzer, aggregator, automatic code rewiew.*

Введение

В современной разработке все больше ценится скорость. При этом создаваемый программный продукт должен быть безопасен, поскольку проблемы с безопасностью могут повлечь убытки превышающие выгоды от быстрой разработки. К тому же проверка кода на безопасность – очень трудоемкий процесс для разработчиков, далеко не всегда приносящий необходимый результат. В связи с этим возникла потребность в автоматической проверке программного обеспечения.

Цель исследования – провести обзор существующих решений для статического анализа кода Golang и выявить анализаторы, комбинация из которых дает все доступные для автоматической проверки кода ошибки в проектах на Golang.

Материал и методы исследования

Для автоматической проверки программного обеспечения есть множество препятствий, главными из которых являются: невозможность определить понятие алгоритма и построить его математическую модель, что заставляет при автоматической проверке пользоваться техниками, лишь приближающимися к истине, а не полностью решающими проблему [1].

Так, в отрасли сформировались две основные группы подходов к автоматической проверке кода. Первая группа определяется использованием статического анализа кода, вторая – динамическим. Статический анализ кода, в противовес динамическому подходу, работает исключительно с исходным текстом программы, тогда как динамический производит проверку в момент выполнения программы [2].

Статические анализаторы в большинстве своем работают на основе абстрактных синтаксических деревьев и правил. Синтаксические деревья получаются за счет лексического и синтаксического анализа кода программы. Лексический анализ вычленяет с помощью конечного автомата идентификаторы и ключевые слова, выдает их в промежуточном представлении. Синтаксический анализатор на основе этого формирует и представляет синтаксическое дерево. Далее анализаторы применяют правила к исходному тексту программы в более простых случаях и в более сложных – к синтаксическому дереву [3, 4].

В связи с отсутствием четкой формальной модели проверки алгоритмов и невозможностью ее описать, имеются серьезные недочеты, с которым приходится мириться – это ложноположительные срабатывания, которые сопоставимы с термином ошибки первого рода и ложноотрицательные срабатывания, которые сродни ошибке второго рода. В итоге с ними справляются, придумывая такие правила, согласно которым данные срабатывания фиксируются, а потом стараются снизить их количество.

Результаты исследования и их обсуждение

Хотя язык Go является относительно молодым, из-за его популярности существует большое количество статических анализаторов [5]. Все инструменты статического анализа можно разделить на две группы:

- анализаторы, чекеры (от англ. checker);
- агрегаторы.

Анализаторы – это неделимые проекты, которые предоставляют какой функционал по проверке исходного кода программы.

Агрегаторы – это проекты, которые включают в себя множество анализаторов и предоставляют к ним единый и удобный доступ.

Для того, чтобы понять, какое соотношения на рынке готовых инструментов статического анализа Golang, была построена карта отношений готовых решений и на ее основе анализаторы были кластеризованы на подгруппы. Информация о существующих решениях для построения карты была взята из открытой базы данных анализаторов [6]. Карта отношений представляет из себя своего рода круги Эйлера только для многомерного случая (рис. 1).

На данной карте строки отображают инструмент, а пересечения по столбцам – что одна и та же технология входит в два инструмента.

На карте явно выделяются три группы:

- группа А – анализаторы, полностью входящие в агрегаторы;
- группа Б – агрегаторы;
- группа В – анализаторы, не входящие в агрегаторы.

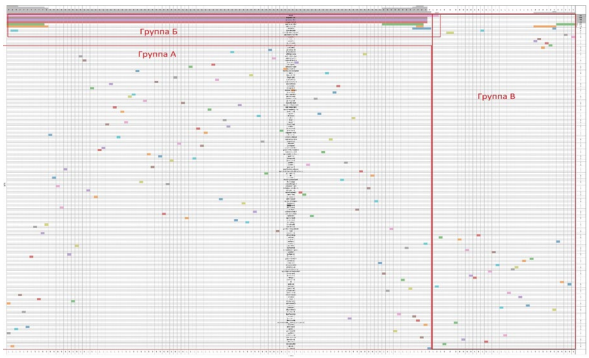

Рис. 1. Карта отношений инструментов статического анализа кода Golang

Группа А характеризуется тем, что туда входят анализаторы, которые прошли проверку эффективности и попали в агрегаторы. Значит, ими можно пользоваться с определенной степенью доверия. Лучше всего их применять в составе одного из агрегаторов группы Б. В группе Б выделяется golangci-lint – агрегатор, включающий в себя все анализаторы из группы А, а также являющийся бесплатным и открытым. Остальные агрегаторы этой группы вызывают внутри себя именно его либо анализаторы, которые уже присутствуют в golangci-lint.

В группе В же остается простор для исследования, поскольку там присутствуют малоиспользуемые анализаторы либо анализаторы, про которые очень мало информации в открытых источниках. В рамках данного изучения были отсеяны анализаторы, непригодные для работы, а среди пригодных – выделены наиболее перспективные, на основе заявленных в документации к ним возможностям и проверкам анализаторов на тестовых программах с ошибками. Все выбранные анализаторы представлены в таблице 1.

Таблица 1

Исследуемые анализаторы

semgrep	sonarqube	bearer	sonatype	codescene	scanmycode_ce	aligncheck
kiuwan	svace	shisho	todocheck	gokart	gotype	lll
nargs	snykcode	applicationinspector	codeintelligence	codefactor	dingohunter	errwrap
goconsistent	goone	goroutineinspect	safesql	sigrid	sqlvet	structslop
unittest	godepgraph	countcode	depth	flen	testify	gochecknoinits
checkmarx_cxsast	golangcilint	goreporter	gometalinter			

Далее из этого списка исключены анализаторы, непригодные для работы:

1) не являются статическими анализаторами, но попали в базу данных (используются при работе некоторых статических анализаторов): testify, flen, depth, countcode, godepgraph, unittest, goroutineinspect, goconsistent, codefactor, lll, gotype, todocheck, sonatype, aligncheck;

2) заявляют о наличии статического анализа Golang, но в действительности его не поддерживают: errwrap, snykcode;

3) имеют слабую поддержку Golang и скорее рассчитаны на анализ других языков: checkmarx_cxsast, sigrid, codeintelligence, kiuwan, semgrep;

4) невозможно собрать бинарный файл анализатора: applicationinspector, shisho, codescene, sonarqube;

5) работают только с sql запросами из Golang: sqlvet, safesql, goone;

6) заявляется маленькое количество правил для проверки кода Golang: gochecknoinits, structslop, dingohunter, nargs.

Оставшиеся для дальнейшего анализа статические анализаторы – bearer, scanmycode_ce (betterscan.io), svace, gokart, golangcilint, goreporter, gometalinter.

В связи с тем, что для использования svace требуется активация, он также исключен из

дальнейшего анализа. При этом важно отметить, что данный анализатор является полезным и перспективным, его нельзя просто игнорировать, но в рамках данной статьи он рассмотрен не был.

Для выявления наиболее перспективных анализаторов из оставшихся была более детально исследована их работа, а также выдаваемые ими ошибки. Для тестирования выбран проект micro, достаточно популярный консольный редактор текста, написанный на Golang [7].

Детальное исследование статических анализаторов

После просмотра исходного кода и зависимостей было проведено тестирование рассматриваемых анализаторов на проекте micro и собрана статистика по их выходам. Результаты представлены в таблице 2.

Таблица 2

Первичная оценка анализаторов на проекте micro

Анализатор	Количество срабатываний	Вызывает те же бинарные файлы других анализаторов (да/нет)	Пригоден для дальнейшего исследования (да/нет)
bearer	52	Нет	Да
scanmycode_ce (betterscan.io)	2	Нет	Нет
gokart	3	Нет	Нет
golangcilint	851	Нет	Да
goreporter	72	Да, вызывает те же анализаторы, что и golangcilint	Нет
gometalinter	57	Да, вызывает те же анализаторы, что и golangcilint	Нет

После первичной оценки были исключены scanmycode_ce (betterscan.io) и gokart, поскольку на большом проекте зафиксировано малое количество срабатываний, а также исключены goreporter и gometalinter. Они вызывают под собой те же бинарные файлы, что и golangcilint. Это приводит к тому, что все срабатывания анализаторов перекрываются golangcilint.

Для оценки двух оставшихся анализаторов построено отношение срабатываний. Отношение представлено в виде кругов Эйлера на рисунке 2.

По соотношению срабатываний видно, что на метриках, оценивающих выявление ошибок, golangcilint будет показывать себя лучше. Для оценки качества срабатываний была оценена значимость ошибок, выявленных bearer, но не выявленных golangcilint. Описание и значимость этих ошибок представлены в таблице 3.

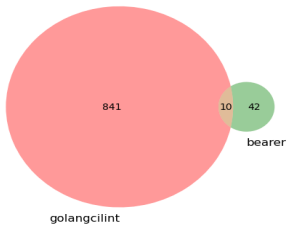

Рис. 2. Отношение срабатываний анализаторов golangcilint и bearer

Таблица 3

Описание уникальных срабатываний bearer

Наименование ошибки	CWE	Описание	Работает ли с такими ошибками golangcilint
Unsanitized dynamic input in OS command	78	Необработанный динамический ввод в команде ОС	Да
Usage of weak hashing library (MD5)	328	Использование библиотеки слабого хеширования	Нет
Permissive file assignment	276	Устанавливаются общедоступные правила доступа к файлу	Нет
Unsanitized user input in file path	73	Несанкционированный пользовательский ввод в пути к файлу	Нет
Permissive temporary file creation	378	Создание временного файла с небезопасными разрешениями	Нет
Unsanitized user input in deserialization method	502	Десериализация ненадежных данных без достаточной проверки достоверности полученных данных	Нет
Missing validation for regular expression	625	Продукт использует регулярное выражение, которое недостаточно ограничивает набор допустимых значений	Нет
Leakage of information in logger message	532	Вставка конфиденциальной информации в файл журнала	Нет

Из данной таблицы видно, что слабой старой golangcilint является выявление ошибок, связанных с работой с внешними файлами, выставление прав, несанкционированный доступ к вводу путей файлов, десериализация данных. Ошибки с регулярными выражениями и чрезмерным логированием можно отнести к незначительным.

Анализаторы были также проверены на функционал анализа потока данных. Для этого были

написаны тесты на разыменование нулевого указателя. Код тестов представлен в таблице 4.

Ни один из исследуемых анализаторов не нашел ошибок в данных тестах, хотя программы не являются работающими. Возможно, эти ошибки выявляет svace [8], поскольку в его документации заявляется работа с потоками данных, но это остается за рамками описанного исследования.

Таблица 4

Тесты на работу с нулевым указателем

Первый тест	Второй тест

package main
 	
import (
"fmt"
)
 	
const nnum = 10
 	
func retNil(num *int) *int {
if *num > nnum {
return nil
}
 	
return num
}
 	
func main() {
num := 11
v := retNil(&num)
fmt.Println(*v)
}
	package main
	
	import (
	"fmt"
	)
	
	func main() {
	var p *int
	num := *p
	fmt.Println(num)
	}

Заключение

В итоге выявлены три анализатора, комбинация из которых дает все доступные для автоматической проверки кода ошибки в проектах на Golang. Следует понимать, что это не все уязвимости, присутствующие в проектах, а лишь полное собрание ошибок, которые возможно выявить работающими на данный момент пакетами для статического анализа. Golangcilint мож-но использовать для нахождения массы тривиальных ошибок, bearer будет дополнять его в рамках работы с операционной системой, а svace – даст возможность находить уязвимости в потоке данных. Использование комбинации из этих трех анализаторов с настройкой их под разрабатываемый проект является наиболее перспективным решением для автоматической проверки кода Golang на данный момент.

Список литературы

1. Хопкрофт Д. Введение в теорию автоматов, языков и вычислений / Д. Хопкрофт, Р. Мотвани, Д. Ульман. – 2-е издание. – Москва : Вильяме, 2008. 528 с.
2. Menshikov M. A. Review of static analyzer service models / M. A. Menshikov // Proceedings of the Institute for System Programming of the RAS. – 2021. – Vol. 33, No. 3. – P. 27–40. – DOI: 10.15514/ISPRAS-2021-33(3)-2.
3. Кошелев В. К. Формализация определения ошибок при статическом символьном выполнении / В. К. Кошелев // Труды Института системного программирования РАН. – 2016. – Т. 28, № 5. – С. 105–118. – DOI: 10.15514/ISPRAS-2016-28(5)-6.
4. Афанасьев В. О. Статический анализатор для языков с обработкой исключений / В. О. Афанасьев // Труды Института системного программирования РАН. – 2022. – Т. 34, № 6. – С. 7–28. – DOI: 10.15514/ISPRAS-2022-34(6)-1.
5. Есилевский С. Языки программирования «новой волны». Язык Go / С. Есилевский // Системный администратор. – 2021. – № 7–8 (224–225). – С. 65–73.
6. Реестр статических анализаторов // analysis-tools.dev. – Режим доступа: https://analysis-tools.dev/ (дата обращения: 24.03.2024), свободный. – Заглавие с экрана. – Яз. рус.
7. Документация проекта micro // https://github.com. – Режим доступа: https://github.com/zyedidia/micro (дата обращения: 30.04.2024), свободный. – Заглавие с экрана. – Яз. рус.
8. Бородин А. Е. Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace / А. Е. Бородин, А. В. Горемыкин, С. П. Вартанов, А. А. Белеванцев // Программирование. – 2021. – № 6. – С. 62–80. – DOI: 10.31857/S0132347421060030.
9. Зарипова В. М. Унаследованные информационные системы. Проблемы и решения / В. М. Зарипова, И. Ю. Петрова // Инженерно-строительный вестник Прикаспия. – 2022. – № 2 (40). – С. 150–158. – DOI: 10.52684/2312-3702-2022-40-2-150-158.
10. Князева Н. В. Использование эволюционных алгоритмов для автоматизации рутинных задач перебора вариантов проектных решений / Н. В. Князева // Инженерно-строительный вестник Прикаспия. – 2021. – № 3 (37). – С. 73–77. – DOI: 10.52684/2312-3702-2021-37-3-73-77.

Ссылка для цитирования:
Галиев Р. М., Евдошенко О. И. Обзор существующих решений для статического анализа кода Golang // Инженерно-строительный вестник Прикаспия : научно-технический журнал / Астраханский государственный архитектурно-строительный университет. Астрахань : ГБОУ АО ВО «АГАСУ», 2024. № 2 (48). С. 73–76.