DOI: 10.15514/ISPRAS-2025-37(6)-4

Статический анализ исходного кода для языка Golang: обзор литературы

Дворцова В. В., ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>
Бородин А.Е., ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>
Институт системного программирования им. В.П. Иванникова РАН,
Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.

Аннотация. Методы статического анализа определяют свойства программы без ее выполнения, при этом различные свойства позволяют решать различные задачи. Мы выполнили обзор статей, посвященных статическому анализу Golang. В данной работе мы изучили 34 публикации, опубликованные с момента выхода первой версии языка Go 1.0 (с 2012 по 2025 год включительно), посвящённые статическому анализу исходного кода на языке Golang. На основе проведённого анализа мы выделили основные направления и методы использования статического анализа, а также рассмотрели используемые промежуточные представления, особенности языка Golang, влияющие на процесс анализа, и трудности, с которыми сталкиваются разработчики статических анализаторов. Этот обзор будет полезен как разработчикам статических анализаторов, так и разработчикам программ на языке Golang, предоставляя им систематизированное понимание текущего состояния исследований в области статического анализа исходного кода на языке Golang.

Ключевые слова: язык программирования Golang; статический анализ; обзор литературы.

Для цитирования: Дворцова В. В., Бородин А.Е. Статический анализ исходного кода для языка Golang: обзор литературы. Труды ИСП РАН, том 37, вып. 6, часть 1, 2025 г., стр. 59–82. DOI: 10.15514/ISPRAS–2025–37(6)–4.

Static Analysis of Golang Source Code: A Survey

V.V. Dvortsova, ORCID: 0000-0002-7424-8442 <vvdvortsova@ispras.ru>
A.E. Borodin, ORCID: 0000-0003-3183-9821 <alexey.borodin@ispras.ru>

Institute for System Programming of the Russian Academy of Sciences,
25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.

Abstract. Static analysis methods determine the properties of a program without executing it, while different properties allow solving different tasks. We have reviewed articles on Golang static analysis. In this paper, we have reviewed 34 papers published since the release of Go 1.0 (2012 – 2025) and focused on static analysis in Golang. Based on our analysis, we have identified the main trends and methods for performing static analysis as well as intermediate representations and features of Golang that affect the process. We have also examined the challenges faced by developers of static analyzers. This survey will be helpful for both developers of static analyzers and Golang developers, providing a systematic understanding of current research in static analysis for Go.

Keywords: Static analysis, Golang, Survey.

For citation: Dvortsova V.V., Borodin A.E. Static Analysis of Golang Source Code: A Survey. Trudy ISP RAN/Proc. ISP RAS, vol. 37, issue 6, part 1, 2025, pp. 59-82 (in Russian). DOI: 10.15514/ISPRAS-2025-37(6)-4.

1. Введение

С момента своего релиза 28 марта 2012 года язык программирования Golang (далее Go) [1] получил широкое применение среди разработчиков программного обеспечения. Go – язык с открытым исходным кодом, в нем сочетается нативная компиляция, статическая типизация, автоматическое управление памятью и упрощенный синтаксис. Как и любой другой язык программирования, Go не защищен от ошибок, которые могут возникнуть на этапах проектирования, написания или поддержки кода. Эти ошибки могут привести к уязвимостям безопасности или снижению производительности. Одним из ключевых подходов к минимизации количества ошибок и улучшению качества кода является статический анализ исходного кода. Статический анализ позволяет выявлять потенциальные проблемы на ранних этапах разработки без запуска программы, что значительно снижает затраты на исправление ошибок и повышает качество программного обеспечения.

Разработчики языка встроили в него легковесный статический анализатор `Go vet` [2], который ищет подозрительные шаблоны на абстрактном синтаксическом дереве (АСД представляет собой структурированное представление исходного кода в виде дерева, где каждый узел соответствует определенной конструкции языка программирования (например, операторы, выражения, функции), такие как несоответствия строки и аргументов формата в методе `fmt.Printf`, бесполезные сравнения между функциями и `nil`, неиспользуемые результаты вызовов некоторых функций и другие. Также создатели языка разработали фреймворк статического анализа `golang.org/x/tools` [3], с помощью которого можно создавать разнообразные анализаторы для широкого круга задач.

Несмотря на растущий интерес к статическому анализу Go, исследования в этой области остаются несистематизированными. Существующие работы охватывают широкий спектр задач, включая анализ параллелизма, обнаружение уязвимостей в зависимостях, оптимизацию управления памятью и выявление «запахов кода». Однако отсутствует целостный обзор, который бы систематизировал текущие достижения, методы и ограничения статического анализа для Go.

Статья структурирована следующим образом:

- в разделе 2 представлен обзор инструментов и характеристик, связанных со статическим анализом кода на языке Go: в подразделе 2.1 описываются ключевые

особенности языка Go, оказывающие влияние на процесс статического анализа, в подразделе 2.2 рассматриваются популярные анализаторы исходного кода;

- в разделе 3 рассматриваются существующие работы, посвященные обзорам исследований в области языка Go;
- в разделе 4 подробно описывается методология проведения обзора, включая используемые источники данных и формулировку исследовательских вопросов;
- в разделе 5 представлен систематизированный обзор исследований и их особенностей реализации: в подразделе 5.1 дается краткий анализ найденных публикаций, классифицированных по основным направлениям исследований, в подразделе 5.2 рассматриваются ключевые аспекты реализации анализа, включая промежуточное представление, используемые методы анализа и ограничения соответствующих работ;
- в разделе 6 обсуждаются ограничения данного исследования;
- в разделе 7 подводятся итоги работы и формулируются выводы.

2. Анализ и характеристики языка Go

2.1 Особенности языка Go

Дизайн языка Golang [1, 4-7] напрямую влияет на подходы к статическому анализу. Ниже перечислены ключевые особенности языка, которые формируют специфику анализа кода:

- Строгая типизация и компиляция:
  - Строгая типизация уменьшает количество ошибок времени выполнения, но увеличивает важность анализа типов на этапе компиляции.
  - Компилятор Go уже выполняет базовый статический анализ, выявляя очевидные ошибки, такие как использование неинициализированных переменных или неверное приведение типов, недостижимый код. Однако для более глубокого анализа требуются сторонние инструменты.
- Параллелизм:
  - Горутины и каналы являются источниками сложных ошибок, например, гонок данных и взаимоблокировок, а также усложняют поток управления программы. Статический анализ для выявления этих проблем требует применения специализированных методов, таких как анализ потока данных и моделирование взаимодействия между потоками.
- Управление ресурсами:
  - В язык встроен сборщик мусора, тем не менее, разработчики часто сталкиваются с утечками ресурсов, например, незакрытыми файловыми дескрипторами или соединениями с базами данных.
- Конструкции, усложняющие поток управления:
  - Инструкция `defer` отложенного вызова функции;
  - Замыкания.
- Другие особенности:
  - Встраивание типов (Struct Embedding);
  - Дженерики (Generics);
  - Коллекции в Go (`map`, `slice`);

- Полиморфизм через интерфейсы;
- Встроенная поддержка нескольких возвращаемых значений;
- Отсутствие исключений и механизма обработки исключений (например, как в Java, Kotlin, C++);
- Отсутствие классов и механизма наследования (например, как в языках Java, Kotlin, C++);
- Отсутствие перегрузки функций (Function overloading).

Вышеперечисленные особенности языка влияют на то, с какими проблемами и уязвимостями можно встретиться в исходном коде.

Статья [6] посвящена анализу ошибок, возникающих в экосистеме языка Go, с целью их понимания и классификации. Авторы статьи проанализировали 51020 отчетов об ошибках (issue reports) в репозитории Go на GitHub. Анализ ошибок помогает выявить слабые места языка. Среди ключевых причин, приводящих к ошибкам, авторы выделили следующие (от частых к редким):

1. неправильная логика кода;
2. некорректная проверка условия;
3. некорректная инициализация переменных;
4. некорректный вызов функции;
5. неправильная обработка исключительных ситуаций и предупреждений;
6. ошибки, связанные с параллелизмом;
7. ошибки при работе с памятью.

Исследование предоставляет данные для создания инструментов анализа, которые могут автоматически обнаруживать и предотвращать распространенные ошибки.

Статья [7] посвящена обзору возможностей линтеров (легковесных статических анализаторов, ищущих шаблоны кода, приводящие к различным ошибкам). В ней выделены основные проблемы исходного кода 20-ти открытых проектов, с которыми линтеры должны справляться (из отчетов по проектам на GitHub), и сформулировали 10 категорий ошибок:

1. слабая ссылка на элемент слайса внутри цикла (Weak element reference in slices) (неактуально после версии Go 1.22);
2. пропуск обработки ошибок (Missing error handling statement);
3. потенциально ошибочные использования операций сравнения (Loose boolean expressions);
4. неправильное использование горутин (Misuse of goroutine);
5. неиспользуемые параметры (Unused parameters);
6. избыточные и неполные объявления типов (Misuse of type conversion and declaration);
7. отсутствие defer (Missing defer);
8. отсутствие проверки на nil (Missing a null check);
9. неправильная обработка специальных символов при кодировании/декодировании (Failed to remove special characters in decoding and encoding);
10. отсутствие обработок ошибок (Absent error handling).

В работе [7] также была исследована работа пяти популярных линтеров: `errcheck` [8], `Gosec` [9], `Go vet`, `revive` [10], `staticcheck` [11]. Авторы отмечают низкую производительность и низкую эффективность этих инструментов, а также отсутствие линтера, который бы умел находить все виды ошибок из указанных категорий (в среднем

линтер мог находить ошибки из одной или двух категорий). Делается вывод, что для повышения качества кода важно использовать комбинацию различных инструментов.

2.2 Статический анализ Go

Для Go разработано множество статических анализаторов (например, Go `vet`, `Gosec`, `errcheck`, `staticcheck`), ищущих различные ошибки в исходном коде, некоторые из них агрегируют в себе другие анализаторы (`golangci-lint` [12]). Так, результаты опроса разработчиков Go 2024 H2 [13] показали, что самым популярным инструментом анализа кода является `gopls` [14], который использовали 65% респондентов, поскольку `gopls` – языковый сервер, разработанный создателями языка Go и используемый в редакторе кода VS Code [15], популярном среди разработчиков на Go. Реже использовались `golangci-lint` (57% респондентов) и `staticcheck` (34%). Только 10% респондентов указали, что не используют никаких инструментов анализа.

В исследовании [16] описывается влияние особенностей языка программирования (в том числе и языка Go) на качество написанных на нем программ. Исследователи проанализировали большое количество открытых проектов на GitHub. Авторы отметили, что есть хоть и небольшая, но существенная связь между особенностями языка и возможными дефектами в программах на нем: функциональные, строго типизированные языки вызывали меньше ошибок, чем процедурные, слабо типизированные, без автоматического управления памятью. Также была отмечена зависимость некоторых типов дефектов, например, ошибок параллелизма и утечек памяти, от имеющихся в языке примитивов. Go имеет строгую статическую типизацию и автоматическое управление памятью, поэтому в программах на нем меньше потенциальных ошибок, чем в программах на языках с динамической типизацией (например, Python, JavaScript) и отсутствием сборщика мусора (например, C/C++).

Отметим, что задач, которые решаются методами статического анализа, много. Так, например, в статье [17] описываются существующие автоматические инструменты статического анализа для Go, их возможности и сценарии использования. Среди них: инструменты генерации кода (`Mockgen`, `Mockery`, `Scaffold`, `Gqlgen`, `gRPC`), инструменты для поиска ошибок или нарушения свойств программы (`Check` [18], `Dupl` [19], `Errcheck` [8], `Gocyclo` [20], `Gosec` [9], `Prealloc` [21], `Safesql` [22]), форматтеры кода (`gofmt` [23]), отладчик (`Delve` [24]) и генератор документации (`Godoc` [25]).

3. Связанные работы

Насколько нам известно, для языка Go отсутствует целостный обзор литературы, который бы охватывал различные методы, инструменты и ограничения статического анализа в контексте Go. Однако в литературе было предложено несколько обзоров, связанных с безопасностью приложений на Go. В уже упомянутой статье [7] ее авторы сделали обзор возможностей пяти популярных линтеров на 20 открытых проектах. В также уже упомянутой статье [17] описываются возможности и сценарии использования существующих автоматических инструментов статического и динамического анализа для языка Go.

4. Методология обзора литературы

Методология, которой мы следовали при создании обзора, основана на рекомендациях Kitchenham [26]. Эта методология уже использовалась другими обзорами [27-28].

Мы изучили 34 статьи, связанных со статическим анализом Go (полный список приведен в табл. 1).

Табл. 1. Полный список рассмотренных публикаций.
Table 1. The complete list of surveyed papers.

Год	Статья
2016	[29] Information flow analysis for Go
2016	[30] Static Deadlock Detection for Concurrent Go by Global Session Graph Synthesis
2016	[31] Detection of Bugs and Code Smells through Static Analysis of Go Source Code
2017	[32] Fencing off Go: liveness and safety for channel-based programming
2018	[33] A static verification framework for message passing in Go using behavioural types
2019	[34] An empirical study of messaging passing concurrency in Go projects
2019	[35] Verifying message-passing programs with dependent behavioural types.
2019	[36] Go-Sanitizer: Bug-Oriented Assertion Generation for Golang.
2019	[37] Godexpo: an automated god structure detection tool for golang.
2020	[38] Static race detection and mutex safety and liveness for Go programs
2020	[39] Escape from escape analysis of Golang
2020	[40] Uncovering the hidden dangers: Finding unsafe Go code in the wild
2021	[41] Breaking type safety in Go: an empirical study on the usage of the unsafe package
2021	[42] Interprocedural static analysis for finding bugs in Go programs
2021	[43] Static analyzer for Go
2021	[44] Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе Svace
2021	[45] Automated verification of Go programs via bounded model checking
2021	[46] GoDetector: Detecting concurrent bug in Go
2021	[47] Analysing GoLang Projects' Architecture Using Code Metrics and Code Smell
2021	[48] Gobra: Modular specification and verification of go programs
2021	[49] Automatically Detecting and Fixing Concurrency Bugs in Go Software Systems
2022	[50] How many mutex bugs can a simple analysis find in Go programs?
2022	[51] Cryptogo: Automatic detection of Go cryptographic api misuses
2022	[52] Detecting blocking errors in Go programs using localized abstract interpretation
2022	[53] Interprocedural static analysis for Go with closure support
2022	[54] Devirtualization for static analysis with low level intermediate representation
2023	[55] Static Analysis for Go: Build Interception
2024	[56] Статический анализ ассоциативных массивов в Go
2024	[57] GoGuard: Efficient Static Blocking Bug Detection for Go
2024	[58] Gopher: High-Precision and Deep-Dive Detection of Cryptographic API Misuse in the Go
2024	[59] Golang Defect Detection based on Value Flow Analysis
2024	[60] GoSurf: Identifying Software Supply Chain Attack Vectors in Go
2024	[61] MEA2: A Lightweight Field-Sensitive Escape Analysis with Points-to Calculation for Golang
2025	[62] An Empirical Study of CGO Usage in Go Projects–Distribution, Purposes, Patterns and Critical Issues.

При поиске статей использовались ключевые слова, такие как *static analysis*, *Golang*, *Go*, *Go programming language*, *code quality tools*, и их комбинации. Поиск проводился в научных базах данных (Google Scolar [63], IEEE Xplore [64], ACM Digital Library [65], SpringerLink [66]) и открытых источниках (GitHub [67]). В обзор включались статьи, описывающие инструменты или методы статического анализа, специфичные для Go, а также обзоры, использующие статический анализ для проведения исследования [34, 41, 62]. Исключались работы, не связанные с Go, а также работы, сосредоточенные только на динамическом анализе.

Данный обзор направлен на решение следующих исследовательских вопросов:

Вопрос 1: Каковы цели статического анализа?

В рамках данного вопроса мы рассмотрим использование статического анализа в контексте повышения качества кода и выделим его основные направления. Другие цели применения статического анализа, не связанные с качеством кода, в этой работе не рассматриваются.

Вопрос 2: Какие методы статического анализа используются?

В этом исследовательском вопросе мы подробно изучаем основные разработанные методы анализа. С этой целью мы исследуем следующие подвопросы:

2.1 Какие промежуточные представления используются при анализе?

2.2 Какие особенности, характерные для Go, принимаются во внимание?

2.3 С какими трудностями сталкиваются разработчики статического анализа приложений на Go?

5. Анализ литературы

После изучения указанных в табл. 1 статей мы выполнили их обзор, разбив их на темы, посвященные определенным ошибкам. Для каждой статьи из направления мы кратко описали основную цель статьи и используемые методы анализа, а также ограничения работы, если таковые были указаны в изучаемой публикации.

На рис. 1 показана динамика количества статей по годам в период с 2012 по 2025 год. Видно, что наибольшее количество статей приходится на 2021 год (9 публикаций), а минимальное – на 2017, 2018 и 2023 годы (по одной публикации).

5.1 Цели статического анализа

Мы выделили 10 направлений исследований (табл. 2 показывает разбиение статей по направлениям), среди которых:

- параллелизм в Go;
- неправильное использование криптографических библиотек;
- неправильное использование пакета unsafe;
- ошибки в CGO коде;
- escape-анализ;
- запахи кода (code smell);
- разыменование нулевого указателя;
- анализ зависимостей;
- статические анализаторы общего назначения;
- вспомогательный статический анализ.

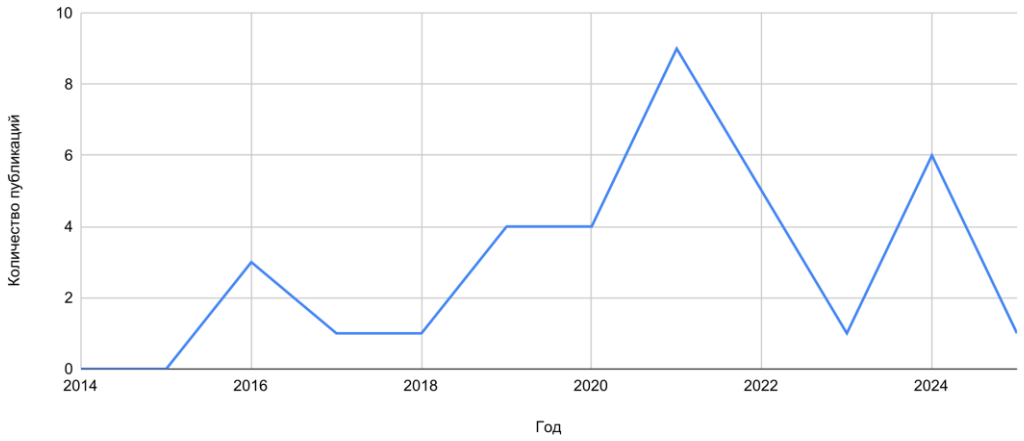

Рис. 1. Количество публикаций по годам.
Fig. 1. Number of papers split by year.

Табл. 2. Основные цели публикаций по статическому анализу языка Go.
Table 2. Goals of Golang static analysis papers.

Цель	Публикации
Параллелизм в Go	Lange [32], Ng [30], Lange [33], Veileborg [52], Scalas [35], Dilley [34], Khatchadourian [50], Gabet [38], Zhang [46], Wolf [48], Dilley2021 [45], Liu [49], Goguard [57]
Запахи кода (code smell)	Yasir [37], Sarker [47], Bergersen [31]
Неправильное использование криптографических библиотек	Li [51], Zhang [58]
Неправильное использование пакета unsafe	Lauinger [40], Costa [41]
Escape-анализ	Wang [39], Ding [61]
Разыменование нулевого указателя	Fu [59], Svace [56]
Статические анализаторы общего назначения	Svace [42-44, 53-56]
Анализ зависимостей	Cesarano [60]
Ошибки в CGO коде	Chen [62]
Вспомогательный статический анализ	Bodden [29], Wang [36]

5.1.1 Параллелизм в Go (Go concurrency)

Go – это статически типизированный язык, предназначенный для эффективного и надежного параллельного программирования. Язык предлагает описывать параллельные вычисления через встроенные в язык механизмы.

Горутины. Горутина [68] – это легковесный поток по сравнению с потоком (thread/тред/ветвь) операционной системы. За управление горутин отвечает библиотека времени выполнения Go, а не операционная система; библиотека мультиплексирует

множество горутин на меньшее количество потоков операционной системы. Важно отметить, что горутины, как и треды, взаимодействуют с общей памятью. Это означает, что все горутины имеют доступ к одним и тем же переменным и данным программы. Поэтому при работе с горутинами необходимо помнить о синхронизации доступа к общим данным.

Каналы. Для синхронизации с помощью передачи сообщений в Go существует встроенный примитив – канал [69-70], обеспечивающий безопасное взаимодействие между горутинами.

Механизмы синхронизации. В Go для синхронизации с использованием общей памяти используются традиционные примитивы, встречающиеся и в других языках программирования: мьютексы (sync.Mutex), которые используются для взаимного исключения доступа к общим ресурсам, предотвращая одновременное изменение данных несколькими горутинами; мьютексы чтения-записи (sync.RWMutex); механизм ожидания группы горутин sync.WaitGroup; атомарные операции (sync/atomic).

Данная особенность языка Go делает актуальным изучение уязвимостей, связанных с параллельностью, и методов поиска соответствующих ошибок. В нашем обзоре мы включили 12 публикаций, относящихся к этому направлению, среди них есть работы, направленные на изучение ошибок, чаще всего встречающихся в реальных проектах, а также на подходы к статическому анализу параллельных программ.

Проблемы, которые решаются в изученных работах, можно разделить на три вида (в табл. 3 показано разделение работ по этим видам):

- ошибки синхронизации с помощью передачи сообщений;
- ошибки синхронизации с использованием общей памяти;
- комбинированные работы.

Табл. 3. Цели публикаций по статическому анализу параллелизма в Go.
Table 3. Goals of papers devoted to static analysis of Go parallel features.

Цель	Публикации
Ошибки с каналами (передачей сообщений)	Lange [32], Ng [30], Lange [33], Veileborg [52], Scalas [35]
Ошибки с примитивами синхронизации	Khatchadourian [50]
Комбинированные работы	Gabet [38], Zhang [46], Wolf [48], Liu [49], Goguard [57]
Обзор	Dilley [34]

В Go рекомендуется передавать сообщения по каналам как средствам взаимодействия, наименее подверженным ошибкам. Эмпирическое исследование на открытых проектах, проведенное в 2019 году [71], показывает, что передача сообщений также подвержена ошибкам, как и общая память, и что неправильное использование каналов с еще большей вероятностью приведет к ошибкам взаимной блокировки (*deadlocks*, когда две или более горутины блокируют друг друга, ожидая ресурс, занятый другой), чем неправильное использование мьютексов. Большинство ошибок блокировки, вызванных синхронизацией с общей памятью, имеют те же причины и пути исправления, что и в традиционных языках программирования (Java, C++/C).

Но, например, реализация `sync.RWMutex` отличается от `pthread_rwlock_t` в языке C. В Go операции записи (*write lock*) имеют более высокий приоритет, чем операции чтения (*read lock*). Это означает, что если есть запрос на запись, то новые запросы на чтение будут отложены до завершения операции записи. В языке C в `pthread_rwlock_t` приоритет по умолчанию отдается операциям чтения. Это может привести к различиям в поведении, включая возможность взаимной блокировки в Go, которая невозможна в аналогичных сценариях в C. В исследовании пришли к выводу, что ошибки передачи сообщений чаще

всего связаны с неправильным закрытием каналов или неверной обработкой закрытых каналов; ошибки доступа к разделяемой памяти чаще всего возникают из-за неправильного использования мьютексов.

В 2019 было проведено другое исследование [34] использования механизма передачи сообщений в 865 открытых проектах на Go (с GitHub [67]). Исследование показало, что большинство проектов использует каналы, причем обычно применяются относительно простые паттерны взаимодействия (отправка в канал и получение сообщение из него).

Одна из первых работ [30] по поиску ошибок взаимной блокировки написана в 2016 году, где авторы предлагали искать такие ошибки с помощью проверки глобального графа сессий (*Global session graph* – объединяет все взаимодействия в программе в единый граф, который описывает общий протокол коммуникации). В публикации под типами сессий [72] понимаются операции Go, используемые для коммуникации между горутинами (например, чтение из канала или запись в него). Далее этой же группой исследователей в работах [32-33, 38] было предложено свое промежуточное представление для анализа многопоточных программ в виде поведенческих типов (*behavioral types*) и проверка такой модели [73-74] с помощью mCRL2 [75].

Другая часть работ стремится к полноте анализа, чтобы пропустить как можно меньше связанных с параллельностью ошибок. В статье [45] предложен метод ограниченной проверки моделей программы, который использует поведенческие типы и трансляцию в Promela [76] для верификации через Spin [77]. В рамках данной работы был разработан инструмент GOMELA [78]. Предложенный подход отличается поддержкой параметризованных программ, комбинированным анализом каналов, мьютексов и sync.WaitGroup, а также направлен на выявление ошибок взаимной блокировки, гонок данных и некорректного использования каналов.

В работе [50] описан простой внутрипроцедурный анализ потока данных для поиска ошибок, связанных с некорректным использованием мьютексов. Простой анализ работает быстрее и легче масштабируется. Есть ложноположительные срабатывания, связанные с отсутствием межпроцедурности детектора и проблемами в алгоритме.

В работе [49] описывается инструмент GCatch [79]. Он объединяет несколько статических детекторов ошибок, в том числе новый детектор блокирующих ошибок (BMOC). Для каждого канала `ch` в программе GCatch находит множество операций, в которых используется данный канал. Далее GCatch определяет набор горутин, обращающихся к каналу `ch`, путём анализа всех горутин, созданных в пределах области анализа канала `ch`. GCatch вычисляет комбинации путей, перебирая все возможные пути выполнения для каждой горутины. Детектор чувствителен к путям и использует SMT-решатель для определения того, приводят ли они к блокирующим ошибкам.

В статье [46] описывается инструмент GoDetector для обнаружения ошибок параллелизма в Go, основанный на анализе АСД. Инструмент анализирует только операции, связанные с каналами, а также игнорирует вызовы функций. Инструмент минимизирует ложноположительные срабатывания за счет анализа мертвого кода. Для каждой переменной `sync.WaitGroup` GoDetector использует конечный автомат с магазинной памятью, чтобы обнаружить ошибку, связанную с неправильным использованием примитива.

В работе [52] для поиска ошибок работы с каналами используется локализованная абстрактная интерпретация (*Localized Abstract Interpretation*), при которой программа разбивается на фрагменты, где фрагмент – это все операции с конкретным каналом. На этапе абстрактной интерпретации анализируется каждый фрагмент программы и строится граф суперлокаций (*superlocation graph*) – конечная система переходов, моделирующая пространство состояний фрагмента. На этапе обнаружения блокирующих ошибок выполняется обход графов суперлокаций с целью поиска абстрактных потоков (*abstract threads*) (множество горутин, встречающиеся в программе) в точках операций с каналами,

для которых отсутствует возможный путь разблокировки. Такие потоки во время выполнения программы могут блокироваться бесконечно. Данный подход не является полным или консервативным, но позволяет анализировать большие программы.

В статье [57] описывается инструмент статического анализа GoGuard, предназначенный для обнаружения блокирующих ошибок. Программа преобразуется в абстрактное представление (*Resource Flow Graph*, RFG), где каждая операция с каналами или мьютексами анализируется на предмет потенциальных блокировок. Ресурс в данном графе представляет каждый канал или объект мьютекса. В этом представлении используется концепция производитель-потребитель (*producer-consumer*) для каждого ресурса, чтобы моделировать поток его коммуникаций между горутинами, например, отправку и получение сообщений в канале или из канала. Ключевая идея заключается в том, что потребитель всегда блокирует выполнение горутины, если только производитель того же ресурса не сможет с ним сопоставиться. Далее RFG обходится, начиная с функции main, пока не встретится ситуация, в которой отсутствует соответствующий потребитель для производителя или наоборот. В таких случаях приостанавливается обход узла внутри его горутины и продолжается обход в других горутинах. После завершения процедуры все приостановленные горутины считаются заблокированными, что указывает на некорректное использование примитивов параллелизма.

Резюме. Go предоставляет несколько механизмов для работы с параллелизмом: горутины, каналы, мьютексы, `sync.WaitGroup`, `sync.Once`. Среди основных ограничений анализа, присущих почти каждой работе, можно выделить: проблемы масштабируемости, наличие ложноположительных срабатываний, покрытие не всех примитивов, а только ограниченного их количества (например, игнорирование `sync.WaitGroup`, `sync.Once` приводит к ложным срабатываниям). Среди параллельных ошибок чаще всего встречаются простые ошибки, которые не требуют сложного статического анализа. Чтобы находить более сложные ошибки, необходимо учитывать все примитивы параллельного программирования в коде и моделировать их взаимодействие, использовать чувствительный к путям и полям структур анализ. Чтобы увеличить скорость анализа, анализируется не весь граф потока управления (ГПУ), а только его подмножество, связанное с примитивами (каналами, мьютексами). Основная проблема статического анализа с помощью построения модели программы, использующей параллельные примитивы, и ее последующей верификации – это масштабируемость на большие проекты.

5.1.2 Неправильное использование криптографических библиотек

В данном разделе обсуждается некорректное использование криптографических библиотек (в том числе устаревших версий), что может угрожать безопасности проекта. Такие ошибки – часть более широкого класса ошибок, неправильного использования программных интерфейсов (Application Programming Interface, API), для которого в англоязычной литературе используется термин API misuse. Существующие инструменты поиска этих библиотек основаны на черных или белых списках.

Первая статья, написанная в 2022 году [51], описывает работу автоматического инструмента CryptoGo, предназначенного для обнаружения проблем безопасности в криптографических библиотеках (`crypto/...` [80] и `golang.org/x/crypto/...` [81]), используемых проектами на языке Go. Инструмент использует прямой и обратный *анализ помеченных данных* (*Taint Analysis*) и 12 криптографических правил. В анализе помеченных данных используются четыре типа функций для выявления опасных потоков: источники (*sources*), распространители (*propagators*), фильтры (также называемые санитайзерами, *sanitizers*) и приемники (*sinks*). В частности, источник – это функция, которая генерирует ненадёжный ввод; приемник – функция, принимающая ненадёжный ввод и передающая его в чувствительное место назначения. Распространитель – это функция, которая передаёт

ненадёжные данные из одной части программы (через переменную) в другую. Фильтр – функция, проходя через которую, переменная становится безопасной. Результат работы был протестирован на 120 открытых проектах, использующих криптографические библиотеки: CryptoGo выявил 622 предупреждения неправильного использования API (с точностью 95,5%) и обнаружил, что 83,33% проектов Go имеют хотя бы одно такое использование.

Ограничение работы [51]. Реализовано покрытие только официального API. CryptoGo может выдать ложноотрицательные результаты в случае вызова API из неофициальных криптографических библиотек Go, а также ложноположительные результаты из-за отсутствия чувствительности к путям.

В работе [58] описывается автоматический инструмент обнаружения неправильного использования криптографических библиотек (`crypto/`... [80] и `golang.org/x/crypto/`... [81]) – Gopher. Gopher состоит из двух основных компонентов: CryDict и Detector. CryDict переводит криптографические правила в формальные ограничения; Detector выявляет неправильные использования криптографических библиотек на основе этих ограничений и предоставляет необходимую информацию о потоке данных CryDict для дальнейшего вывода и разработки новых ограничений.

Ограничение работы [58]. Ложноположительные срабатывания из-за ошибок в описании ограничений, отсутствия чувствительности к путям, построения графа вызовов с помощью анализа иерархии классов. Ложноотрицательные срабатывания из-за возможности анализировать файлы только на языке Go; кроме того, не поддерживаются формальные ограничения для неофициальных (пользовательских) криптографических библиотек.

Резюме. Поиск ошибок, связанных с криптографическими библиотеками, выполняется с помощью анализа помеченных данных, чувствительности к путям и генерации формальных ограничений (например, черные или белые списки; спецификации; пометка источников и приемников) для этих библиотек. Основная трудность – наличие неофициальных библиотек, для которых необходимой информации нет.

5.1.3 Неправильное использование пакета unsafe

Язык программирования Go обеспечивает безопасность памяти и потоков через автоматическое управление памятью (сборки мусора) и строгую систему типов, однако использование пакета `unsafe` позволяет обойти эти механизмы безопасности, что может привести к уязвимостям, таким как переполнение буфера или повреждение памяти.

В статье [40] было исследовано 500 открытых проектов на GitHub [67] и найдено 1400 использований пакета `unsafe`. Среди этих проектов 38% проектов используют `unsafe` напрямую, 91% имеют зависимости, которые используют `unsafe`.

В работе были определены основные паттерны использования пакета `unsafe`:

- приведение типов (например, строки к байтам);
- управление памятью (например, использование `uintptr` для арифметики указателей);
- оптимизация производительности (например, уменьшение числа лишних копирований данных).

Здесь также определены основные потенциальные уязвимости:

- гонки сборщика мусора: неправильное использование `uintptr` может привести к освобождению памяти, которая все еще используется;
- ошибки escape-анализа: неправильное использование указателей может привести к утечкам памяти или повреждению данных;
- переполнение буфера: неправильное приведение типов может привести к доступу за пределами выделенной памяти.

В работе также был предложен АСД-детектор [82] go-safer, который ищет некорректные использования `reflect.SliceHeader`, `reflect.StringHeader` и небезопасные преобразования между структурами с архитектурно-зависимыми типами полей. Важно отметить, что в 2025 году структуры `reflect.SliceHeader` и `reflect.StringHeader` являются устаревшими (deprecated), так как, начиная с версии Go 1.20, вместо них появились функции `unsafe.String` и `unsafe.StringData`; `unsafe.Slice` и `unsafe.SliceData` соответственно.

Исследование [41] изучает использование пакета `unsafe` в реальных проектах. В статье авторы проанализировали 2438 популярных открытых проектов на Go. Исследование показывает, что хотя `unsafe` широко используется для производительности и интеграции, этот пакет также создает значительные риски и проблемы, требующие улучшенных инструментов и практик для их устранения.

Резюме. Ошибки, связанные с использованием пакета `unsafe`, могут приводить к серьёзным проблемам, включая нарушения целостности памяти и неопределённое поведение. В одной из рассмотренных работ небезопасные преобразования обнаруживались с помощью анализа АСД. Однако в ходе обзора мы не выявили публикаций, посвящённых систематическому анализу современного API `unsafe` в языке Go.

5.1.4 Запахи кода

Запахи кода (*code smells*)–синтаксически корректные участки кода, которые дают основание заподозрить проблемы с архитектурой приложения или качеством кода, при этом, строго говоря, не являющиеся ошибками.

В магистерской диссертации [31] описывается статический анализ для поиска запахов кода. Инструмент поставляется в виде плагина для SonarQube [83]. В работе с помощью анализа АСД вычисляется цикломатическая сложность [84] функции и метода; небезопасное использование переменной цикла в горутине (потеряло актуальность после Go 1.22); игнорирование проверки error из функции и другие.

В работе [37] описывается реализация инструмента для поиска *God Struct* [85] (структура, выполняющая слишком много функций) в программах на языке Go. С помощью анализа АСД вычисляются три метрики: *WMC (Weighted Method Count), TCC (Tight Class Cohesion)* и *ATFD (Access To Foreign Data)*. Авторы успешно протестировали инструмент на двух открытых проектах.

В статье [47] описывается анализ архитектуры проектов на языке Go с использованием метрик *MOOD (Metrics for Object-Oriented Design)* [86] и *CK (Chidamber and Kemerer Metrics)* [87] – это два набора метрик, используемых для оценки качества объектно-ориентированного программного обеспечения. Они помогают анализировать архитектурные характеристики программного кода, такие как сложность, связность, наследование и инкапсуляция. Для обнаружения запахов кода в Go в работе эти метрики помогают выявить *God Struct* или *Feature Envy* (метод, использующий данные других структур). Описывается успешное тестирование на 5 открытых проектах. В статье большее внимание уделяется вычислению метрик, чем организации анализа.

Резюме. Ошибки типа запахи кода можно выявлять по различным метрикам: *MOOD, CK, WMC, TCC, ATFD*, которые могут использоваться как для популярных объектно-ориентированных языков программирования, так и для Go. Для их вычисления достаточно анализа АСД.

5.1.5 Разыменование нулевого указателя

В работе [59] с помощью анализа потока значений ищется ошибка разыменования нулевого указателя. На основе (*points-to*) [88] анализа в работе строится граф значений.

Авторы расширили этот алгоритм моделированием семантики Go-каналов.

Ограничения работы [59]. Алгоритм использует только анализ потока значений, что приводит к наличию ложноположительных срабатываний в параллельных программах. В качестве решения данной проблемы авторы рекомендуют использовать SMT-решатель, отмечая, что предложенный алгоритм будет работать медленнее. Отсутствие результатов тестирования на средних и больших проектах.

Статья [56] посвящена статическому анализу ассоциативных массивов (map) в языке Go для поиска разыменования нулевых указателей при извлечении ключей. Анализ использует символьное выполнение, на основе которого была промоделирован семантика Go map.

Ограничения работы [56]. Ложные срабатывания возникают из-за ограниченной точности при моделировании условий достижимости, когда анализатор не может полностью выявить взаимосвязи между переменными. Они также могут быть обусловлены сложной логикой программы, например, при итерации по всем ключам отображения без предварительной проверки их существования.

Резюме. В описанных работах используются методы анализа, применимые и для анализа других языков, но с расширением для специфичных конструкций языка Go: каналы (channels), ассоциативные массивы (maps).

5.1.6 Статический анализатор общего назначения

Ранее перечисленные публикации были сфокусированы только на какой-то одной задаче: поиск ошибок в параллельных программах, неправильное использование API, неправильное использование пакета unsafe и т.д. Для Go существуют как открытые инструменты (Go vet [2], Gosec [9]), так и закрытые (Svace, Coverity [89]), которые ищут сразу множество ошибок или самостоятельно, или агрегируя несколько инструментов (golangci-lint [12]).

Go vet, Gosec, golangci-lint – открытые анализаторы, ищут шаблоны на АСД, которые потенциально могут вызывать ошибки. Публикаций по особенностям анализа для данных инструментов мы не нашли.

В работах [42-43, 54], посвященных статическому анализу языка Go в инструменте Svace, описывается межпроцедурный статический анализ на основе резюме. В нем реализован движок символьного выполнения, с помощью которого инструмент может находить множество сложных ошибок: разыменование нулевого указателя, деление на ноль, целочисленные переполнения, утечки ресурсов и чувствительных данных и другие. Для проверки выполнимости путей в Svace используется SMT-решатель. Инструмент также отличается наличием анализов, посвященным особенностям языка Go, таким как анализ замыканий [53] и анализ ассоциативных массивов (maps) [56]. Стоит отметить, что для получения промежуточного представления Svace использует технику автоматической контролируемой сборки [55] (*build capture*), которая адаптирована для Go (отслеживаются как вызовы команды go build/install/run, так и прямые вызовы компилятора Go compile).

Coverity – закрытый инструмент статического анализа. По документации можно сделать вывод, что в инструменте есть движок межпроцедурного чувствительного к путям и контексту анализа, а также поддержка контролируемой сборки; используется ли это для Go – неизвестно.

Резюме. Для Go существуют как открытые легковесные анализаторы, выявляющие типичные ошибки на уровне АСД, так и закрытые тяжеловесные инструменты, использующие более сложные методы для обнаружения нетривиальных ошибок, такие как Svace и Coverity.

5.1.7 Escape-анализ

Работа [39] посвящена исследованию и улучшению escape-анализа (он помогает определить, должна ли переменная храниться на стеке (*stack*) или на куче (*heap*), если переменная может

использоваться после завершения функции – она размещается на куче) в языке Go. Авторы изучили работу этого алгоритма в компиляторе Go и предложили подходы для его улучшения: оптимизация для ситуаций, в которых переменная-указатель используется в качестве аргумента функции. Чтобы получить информацию об утекающих переменных, инструмент изучает журналы сборщика мусора (*garbage collection (GC)*), на основании чего оптимизирует исходный код программы. Затем с помощью анализа АСД- и LLVM-промежуточных представлений проверяется целостность памяти. Метод на основе АСД проверяет, является ли вызов функции синхронным. Указатели, передаваемые в синхронный вызов функции в качестве аргументов, не будут освобождены до завершения вызова, поскольку такой вызов не создаёт новую горутину (или поток). Следовательно, в этом случае оптимизация является корректной. Если вызов функции является асинхронным, то используется метод на основе LLVM-представления.

В публикации [61], посвященной улучшению escape-анализа в Go, авторы предлагают новый алгоритм, который использует чувствительность к полям и *points-to* анализ, чтобы повысить точность без значительного увеличения времени компиляции. Алгоритм реализован в фреймворке MEA, который использует промежуточное представление LLVM. Для улучшения межпроцедурного анализа в работе используется подход, основанный на резюме функций. Анализируются замыкания: в месте создания структуры замыкания и передачи в нее захваченных переменных вставляется псевдовызов замыкания, где используется его резюме (анализ делает консервативное предположение, что адрес захваченной переменной получается из кучи, тем самым выделяя захваченную переменную в куче). Также в работе анализируются инструкции defer и определяются функции, вызываемые через них. Затем для выявленных отложенных вызовов вставляются псевдовызовы в конце родительской функции в соответствии с их порядком в графе потока управления.

Резюме. Основная проблема escape-анализа в Go – его консервативность; часто переменные размещаются в куче, хотя могли бы остаться на стеке. К улучшению escape-анализа можно подойти с разных сторон: оптимизируя исходный код или создание нового алгоритма. В первой работе оптимизируется ситуация, когда переменная-указатель является формальным параметром функции, а во второй работе предлагается чувствительный к полям анализ.

5.1.8 Вспомогательный статический анализ

Статья [36] посвящена разработке инструмента Go-Sanitizer – генератора проверок (*assertions*), которые помогают выявлять и локализовывать ошибки с помощью дальнейшего фаззинга и модульного тестирования. Рекомендация проверок и вставка их в код проводится с помощью анализа АСД. Проверки генерируются для 9 типов ошибок (из CWE [90]: 128, 190, 191, 785, 466, 824, 478, 1077, 777). Например, для CWE-824 авторы используют проверку указателя на `nil`. Для проекта [91] было вставлено 42 проверки и обнаружено 12 ошибок, которые не удалось найти с помощью фаззинга или модульного тестирования.

В одной из первых работ [29] по статическому анализу Go описывается разработка анализа потоков данных, включающего анализ объектных типов (структуры, массивы) и коммуникации через каналы; также обсуждается, как эта информация может использоваться во время выполнения (инструментирование кода). Для точного отслеживания потоков данных используется комбинация методов статического анализа: контекстно-чувствительный межпроцедурный анализ, анализ алиасов, анализ помеченных данных.

Резюме. В данной секции рассматриваются работы, в которых статический анализ применяется как инструмент для извлечения дополнительной информации, используемой в дальнейшем при проведении динамического анализа, включая фаззинг, инструментирование кода и модульное тестирование.

5.1.9 Анализ зависимостей

Рост сложности программ привел к использованию сторонних библиотек (*third-party libraries*) для снижения затрат на разработку. Вместе с тем такие зависимости могут содержать дополнительные уязвимости [92]. Go изначально был разработан с поддержкой децентрализованных реестров, где зависимости могут быть импортированы напрямую из git-репозиториев [93]. Отсутствие централизованного реестра в Go ограничивает возможности по отслеживанию уязвимостей в модулях [94].

В статье [60] описывается инструмент GoSurf, предназначенный для облегчения анализа зависимостей с открытым исходным кодом и аудита с целью обнаружения вредоносного кода. В работе рассматриваются специфические для Go векторы атаки (12 векторов), например, статическая генерация кода (`//go:generate`), тестовые функции (запускаемые через `go test`), методы-конструкторы, интерфейсы, `unsafe` и другие. Эти векторы атаки GoSurf обнаруживает через анализ АСД программы. Анализатор был протестирован на 500 часто импортируемых Go модулях, было выявлено проявление некоторых из 12 векторов атак в каждом из проанализированных модулей.

Резюме. Перед тем, как интегрировать стороннюю зависимость, разработчики должны провести тщательный аудит безопасности, сделать это можно с помощью статического анализа. Для Go можно выделить специфичные векторы атак, чтобы их обнаружить, достаточно анализа АСД.

5.1.10 Ошибки в CGO-коде

CGO – интерфейс внешних функций в Go, обеспечивающий взаимные вызовы функций между Go и Си, позволяющий компонентам среды выполнения Go, таким как сборщик мусора, аллокатор памяти и планировщик горутин, взаимодействовать с программами на Си во время выполнения.

В исследовании [62] авторы представили первое исследование по использованию CGO-кода в открытых проектах на GitHub [67]. Они разработали статический анализатор CGOAnalyzer, работающий над АСД представлении, и применили анализатор для определения основных шаблонов использования CGO.

Резюме. Инструментов статического анализа кода, ищущих ошибки неправильного использования CGO, мы не обнаружили. Данное направление открыто для разработок.

5.1.11 Типы ошибок

На основе анализа публикаций, попавших в исследование, мы выделили 10 направлений использования статического анализа, а также следующие основные типы ошибок (без учета анализаторов общего назначения):

- неправильное использование каналов, горутин и примитивов синхронизации;
- запахи кода;
- неправильное использование криптографических библиотек;
- неправильное использование пакета unsafe;
- разыменование nil;
- векторы атак;
- неправильное использование CGO.

Вопрос 1: Мы выделили 9 направлений публикаций по статическому анализу Go и 7 типов ошибок. Статический анализ чаще всего проводится, чтобы находить ошибки, связанные с параллельным программированием.

5.2 Реализация анализа

5.2.1 Промежуточное представление

В табл. 4 перечислены промежуточные представления, которые чаще всего используются инструментами статического анализа кода. Чаще всего используется АСД, предоставляемое стандартной библиотекой Go – `go/ast`, для поиска определенных шаблонов дерева, например, блоков `unsafe` [40] кода или CGO [62].

Вторым по популярности является SSA-представление [95]. Создатели языка Go специально для статического анализа разработали свое SSA-представление (`golang.org/x/tools/go/ssa`), генерируемое инструментом `ssadump` [96]. Некоторые работы на основе стандартного SSA-представления языка Go строят свое собственное промежуточное представление (например, SvaceIR [43] – низкоуровневое типизированное промежуточное представление).

Несколько работ выбрали LLVM-представление [97] компилятора Go, развиваемого компанией Google в рамках проекта gollvm [98]. В работе [61] отмечается, что алгоритмы, разработанные на основе LLVM-представления, легче адаптировать для других языков программирования. Это связано с тем, что LLVM предоставляет универсальное промежуточное представление и уже включает набор инструментов для анализа.

В другой части работ для реализации анализа авторы использовали собственный парсер языка Go, строящий SSA-представление. Выбор в пользу SSA-представления устраняет сложности, связанные с изменением значений переменных в процессе выполнения программы, так как в SSA каждой переменной значение присваивается только один раз, что значительно упрощает анализ потока данных. SSA позволяет выполнять такие анализы, как символьное выполнение, абстрактная интерпретация и анализ потока данных, межпроцедурный анализ, чувствительный к путям и контексту анализ, которые трудно реализовать на уровне АСД. SSA облегчает обработку сложных языковых конструкций, например, в АСД замыкания могут быть представлены как вложенные функции, но их анализ требует дополнительной работы для отслеживания захваченных переменных, в SSA же захваченные переменные явно представлены через их адреса, что упрощает анализ. АСД, в свою очередь, больше подходит для синтаксического анализа.

Табл. 4. Промежуточное представление, используемое для статического анализа.
Table 4. Intermediate representations utilized for static analysis.

Промежуточное представление	Публикации
go/ast	Wang [36, 39], Bergersen [31], Lauinger [40], Zhang [46], Chen [62], Khatchadourian [50], Sarker [47], Dilley2019 [34], Dilley2021 [45], Yasir [37], GoSurf [60], Costa [41]
golang.org/x/tools/go/ssa	Bodden [29], Lange [32, 33], Liu [49], Li [51], Veileborg [52], Zhang [58], Svace [42–44, 53–56]
Gollvm IR	Wang [39], Fu [59], Ding [61]
Собственный парсер	Ng [30], Gabet [38]
Явно не указано	Scalas [35], Gobra[48], Liu2024 [57]

Вопрос 2.1: Чаще всего для статического анализа программ на языке Go используются промежуточные представления АСД- или SSA-, предоставляемые экосистемой Go: `go/ast` и `golang.org/x/tools/go/ssa`.

5.2.2 Методы анализа

Во всех рассмотренных публикациях анализируется поток управления или данных программы, а также используются специальные методы статического анализа. Для анализа параллельных программ чаще всего используется метод верификации типов и модели (*Type/Model Checking*), с помощью которого анализируется поведение программы и проверяется ее корректность [30, 32-33, 38, 45, 71]. Также в некоторых работах используется символьное выполнение (например, в работе [58] для вывода формальных ограничений, в анализаторе Svace внутрипроцедурный анализ основан на символьном выполнении с объединением состояний в точках слияния путей). В работе [52] используется локализованная абстрактная интерпретация для поиска ошибок при работе с каналами в Go. Для отслеживания потока помеченных данных применяется анализ помеченных данных (в работах [29, 44, 51]). Во многих работах комбинируются сразу несколько методов статического анализа (например, в работах [43, 49, 58, 61]).

Вопрос 2: Основные методы статического анализа, используемые для анализа программ на Go, включают верификацию типов и моделей, символьное выполнение, анализ потока данных, анализ помеченных данных. Часто для повышения точности анализа методы комбинируются.

Отметим, что для языка Go общие методы статического анализа работают или без изменений, или со сравнительно небольшими вариациями для учета специфики языка (каналы, замыкания, инструкции defer, коллекции). Это позволяет реализовывать поддержку Go в многоязыковых анализаторах, в частности, в анализаторе Svace.

Для отсеивания несуществующих путей в статическом анализаторе может использоваться SMT-решатель [49, 99]. Для того, чтобы разрешить вызов процедур для указателей на функции в Go, можно использовать девиртуализацию [54] или построение среза программы (извлекается подмножество кода программы (срез), которое влияет на определенные переменные или операции). Например, в работе [58] срезы используются для анализа иерархии структур в Go. Также для Go применимы техники анализа потока данных (например, анализ недостижимого кода [43, 46]). Часть работ для межпроцедурного анализа используют резюме (например, [43, 61]). Для моделирования поведения библиотечных функций в Go в работах [43, 56] используются спецификации (модели функций, представляющие из себя код на анализируемом языке, в котором используются функции для обозначения некоторых свойств, например, фиксируется, что функция создает или освобождает ресурсы).

Вопрос 2.2: Чаще всего при статическом анализе программ на языке Go принимаются во внимание примитивы параллельного программирования, встраивание структур, замыкания, инструкция `defer` в Go.

Большая часть работ посвящена одной особенности языка Go: примитивам параллельного программирования, встраиванию структур (*Struct Embedding* [54], коллекциям в Go ([56]), замыканиям и инструкции отложенного вызова defer [53].

5.2.3 Ограничения работ

Задача выявления всех ошибок с помощью статического анализа в программе является алгоритмически неразрешимой [100]. Это означает, что невозможно гарантированно найти все ошибки определённого типа в произвольной программе без ложных срабатываний.

К ложноположительным срабатываниям (*false positive*, то есть ошибкам первого рода, когда выдалось ложное предупреждение об ошибке) приводят следующие причины: отсутствие чувствительности к путям [51-52, 58]; отсутствие чувствительности к потоку (например, [53]); сложность ГПУ [52]; реализация метода (например, учитываются только неизменяемые переменные, не учитываются конструкции внутри структур данных, бесконечные состояния

[33]), виртуальные методы [45], ограничения пакетов анализа (анализ алиасов, анализ графа вызовов) [49], отсутствие межпроцедурности [49], не все конструкции языка учитываются (например, могут не учитываться конструкции `Once` и `defer` [46]).

К ложноотрицательным срабатываниям (*false negative*, то есть ошибкам второго рода, когда ошибка в программе есть, но она не была обнаружена анализатором) приводят следующие причины: реализация метода – отсутствие необходимых данных о пользовательских библиотеках [51, 58]; недостаточная масштабируемость анализа (например, [30, 32-33, 46, 50]), неполный учет ошибочных паттернов [49, 52], неполный анализ межпроцедурных связей [52], неполный учет конструкций языка [32, 46] (например, не учитываются конструкции `Mutex`, `WaitGroup` и `Cond` [52]).

Вопрос 2.3: Чаще всего инструменты статического анализа, описанные в публикациях, сталкиваются с проблемами масштабируемости, ограничений в реализации методов анализа, отсутствия чувствительности к путям или потоку.

6. Ограничения данного обзора

В обзор включены публикации за период с 2012 по 2025 год. Для части инструментов с закрытым исходным кодом не удалось найти публикации с подробным описанием реализуемого анализа; однако известно, что эти инструменты поддерживают язык Go.

В исследовании учитываются публикации только на английском и русском языках. Выбор ключевых слов, интерпретация результатов и классификация публикаций могут зависеть от предпочтений авторов, что снижает объективность обзора.

7. Заключение

Чтобы перечислить задачи, стоящие перед разработчиками статических анализаторов, мы выполнили обзор публикаций о подходах, связанных с использованием статического анализа в программах на языке Go. В процессе этого обзора мы изучили 34 научных статьи, опубликованные на конференциях и в журналах по разработке программного обеспечения, языкам программирования и безопасности. Мы ставили себе целью изучить задачи, на которые нацелен статический анализ; основные методы статического анализа, используемые в публикациях; реализации самого анализа и его ограничений.

Наш обзор помог сделать следующие ключевые выводы:

- на основе найденных публикаций мы выделили 9 направлений использования статического анализа для Go;
- больше всего публикаций посвящено поиску ошибок взаимной блокировки и ошибкам с использованием традиционных примитивов синхронизации (мьютексов);
- чаще всего для статического анализа программ на языке Go используются АСД- или SSA- промежуточные представления;
- основные методы статического анализа, используемые для анализа программ на Go, включают верификацию типов и моделей, символьное выполнение, анализ потока данных, анализ помеченных данных;
- примитивы параллельного программирования, встраивание структур, замыкания, defer в Go чаще всего принимаются во внимание при статическом анализе программ;
- инструменты статического анализа, описанные в публикациях, сталкиваются с проблемами масштабируемости, ограничений в реализации методов анализа, отсутствия чувствительности к путям или потоку;
- в целом, методы статического анализа для языка Go такие же, как и для анализа других языков программирования, с корректировкой на особенности языка.

Надеемся, что данный обзор литературы поможет как разработчикам статических анализаторов, так и разработчикам программ на языке Go.

Список литературы / References

[1]. Golang main page. https://go.dev/. Доступ: 2024-01-10.

[2]. Go vet main page. https://golang.org/cmd/vet/. Доступ: 2023-10-01.

[3]. Go tools. https://godoc.org/golang.org/x/tools. Доступ: 2023-10-05.

[4]. A. A. Donovan и B. W. Kernighan. The Go programming language. Addison-Wesley Professional, 2015.

[5]. Effective go– the go programming language. https://go.dev/doc/effective_go. Доступ: 2024-10-05.

[6]. Y. Feng и Z. Wang. Towards understanding bugs in Go programming language. В 2024 IEEE 24th International Conference on Software Quality, Reliability and Security (QRS), страницы 284–295, 2024. DOI: 10.1109/QRS62785.2024.00036.

[7]. J. Wu и J. Clause. Assessing Golang static analysis tools on real-world issues. Available at SSRN 5208109.

[8]. Errcheck main page. https://github.com/kisielk/errcheck. Доступ: 2024-01-10.

[9]. Go security checker – gosec. https://github.com/securego/gosec. Доступ: 2024-01-10.

[10]. Revive. https://github.com/mgechev/revive. Доступ: 2025-02-11.

[11]. Staticcheck main page. https://staticcheck.io. Доступ: 2024-01-10.

[12]. Go linters runner– golangci-lint. https://github.com/golangci/golangci-lint. Доступ: 2023-10-01.

[13]. Go developer survey 2024 h2 results. https://go.dev/blog/survey2024-h2-results. Доступ: 2024-12-21.

[14]. Gopls. https://pkg.go.dev/golang.org/x/tools/gopls. Доступ: 2023-10-03.

[15]. Visual studio code. https://code.visualstudio.com/. Доступ: 2023-10-04.

[16]. E. D. Berger, C. Hollenbeck, P. Maj, O. Vitek и J. Vitek. On the impact of programming languages on code quality: a reproduction study. ACM Transactions on Programming Languages and Systems (TOPLAS), 41(4):1–24, 2019.

[17]. M. H. Ruge. Analysis of software engineering automation tools for Go. Universidad de los Andes. https://hdl.handle.net/1992/54945. Доступ: 2025-02-11.

[18]. Opennota/check. https://gitlab.com/opennota/check. Доступ: 2025-02-11.

[19]. M. bohusl'avek, mibk/dupl. https://github.com/mibk/dupl. Доступ: 2025-02-11.

[20]. Fzipp, fzipp/gocyclo. https://github.com/fzipp/gocyclo. Доступ: 2025-02-11.

[21]. A. kohler, alexkohler/prealloc. https://github.com/alexkohler/prealloc. Доступ: 2025-02-11.

[22]. Stripe/safesql. https://github.com/stripe/safesql. Доступ: 2025-02-11.

[23]. Gofmt. https://pkg.go.dev/cmd/gofmt. Доступ: 2023-10-04.

[24]. Delve. https://github.com/go-delve/delve. Доступ: 2025-02-11.

[25]. Godoc. https://pkg.go.dev/golang.org/x/tools/cmd/godoc. Доступ: 2025-02-11.

[26]. B. Kitchenham. Procedures for performing systematic reviews. Keele, UK, Keele University, 33(2004):1–26, 2004.

[27]. L. Li, T. F. Bissyand'e, M. Papadakis, S. Rasthofer, A. Bartel, D. Octeau, J. Klein и L. Traon. Static analysis of android apps: a systematic literature review. Information and Software Technology, 88:67– 5, 2017. DOI: https://doi.org/10.1016/j.infsof.2017.04.001.

[28]. P. H. Nguyen, M. Kramer, J. Klein, M. Schulz, B. R. de Supinski и M. S. M..uller. An extensive systematic review on the model-driven development of secure systems. Scientific Programming, 21(3-4):109–121, 2013.

[29]. E. Bodden, K. I. Pun, M. Steffen, V. Stolz и A.-K. Wickert. Information flow analysis for go. В International Symposium on Leveraging Applications of Formal Methods, страницы 431–445. Springer, 2016.

[30]. N. Ng и N. Yoshida. Static deadlock detection for concurrent go by global session graph synthesis. В Proceedings of the 25th International Conference on Compiler Construction, страницы 174–184, 2016.

[31]. C. B. Bergersen. Detection of Bugs and Code Smells through Static Analysis of Go Source Code. Дис. маг., 2016.

[32]. J. Lange, N. Ng, B. Toninho и N. Yoshida. Fencing off go: liveness and safety for channel-based programming. ACM SIGPLAN Notices, 52(1):748–761, 2017.

[33]. J. Lange, N. Ng, B. Toninho и N. Yoshida. A static verification framework for message passing in go using behavioural types. В Proceedings of the 40th International Conference on Software Engineering, страницы 1137–1148, 2018.

[34]. N. Dilley и J. Lange. An empirical study of messaging passing concurrency in go projects. В 2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER), страницы 377–387. IEEE, 2019.

[35]. A. Scalas, N. Yoshida и E. Benussi. Verifying message-passing programs with dependent behavioural types. В Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, страницы 502–516, 2019.

[36]. C. Wang, H. Sun, Y. Xu, Y. Jiang, H. Zhang и M. Gu. Go-sanitizer: bug-oriented assertion generation for Golang. В 2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW), страницы 36–41. IEEE, 2019.

[37]. R. M. Yasir, M. Asad, A. H. Galib, K. K. Ganguly и M. S. Siddik. Godexpo: an automated god structure detection tool for golang. В 2019 IEEE/ACM 3rd International Workshop on Refactoring (IWoR), страницы 47–50. IEEE, 2019.

[38]. J. Gabet и N. Yoshida. Static race detection and mutex safety and liveness for go programs. В 34th European Conference on Object-Oriented Programming (ECOOP 2020), страницы 4–1. Schloss Dagstuhl–Leibniz-Zentrum f..ur Informatik, 2020.

[39]. C. Wang, M. Zhang, Y. Jiang, H. Zhang, Z. Xing и M. Gu. Escape from escape analysis of Golang. В Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice, страницы 142–151, 2020.

[40]. J. Lauinger, L. Baumg..artner, A.-K. Wickert и M. Mezini. Uncovering the hidden dangers: finding unsafe go code in the wild. В 2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom), страницы 410–417. IEEE, 2020.

[41]. D. E. Costa, S. Mujahid, R. Abdalkareem и E. Shihab. Breaking type safety in Go: an empirical study on the usage of the unsafe package. IEEE Transactions on Software Engineering, 48(7):2277–2294, 2021.

[42]. I. Bolotnikov и A. Borodin. Interprocedural static analysis for finding bugs in go programs. Programming and Computer Software, 47:344–352, 2021.

[43]. A. Borodin, V. Dvortsova, S. Vartanov и A. Volkov. Static analyzer for Go. В 2021 Ivannikov Ispras Open Conference (ISPRAS), страницы 17–25. IEEE, 2021.

[44]. А. Е. Бородин, А. В. Горемыкин, С. П. Вартанов и А. А. Белеванцев. Поиск уязвимостей небезопасного использования помеченных данных в статическом анализаторе svace. Труды Института системного программирования РАН, 33(1):7–32, 2021.

[45]. N. Dilley и J. Lange. Automated verification of go programs via bounded model checking (artifact), 2021.

[46]. D. Zhang, P. Qi и Y. Zhang. Godetector: detecting concurrent bug in go. IEEE Access, 9:136302–136312, 2021.

[47]. M. K. Sarker, A. A. Jubaer, M. S. Shohrawardi, T. C. Das и M. S. Siddik. Analysing GoLang projects' architecture using code metrics and code smell. В Proceedings of the First International Workshop on Intelligent Software Automation: ISEA 2020, страницы 53–63. Springer, 2021.

[48]. F. A. Wolf, L. Arquint, M. Clochard, W. Oortwijn, J. C. Pereira и P. M..uller. Gobra: modular specification and verification of go programs. В International Conference on Computer Aided Verification, страницы 367–379. Springer, 2021.

[49]. Z. Liu, S. Zhu, B. Qin, H. Chen и L. Song. Automatically detecting and fixing concurrency bugs in go software systems. В Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, страницы 616–629, 2021.

[50]. F. T. H. M. R. Khatchadourian и Y. Cong. How many mutex bugs can a simple analysis find in Go programs? В Annual Conference of the Japanese Society for Software Science and Technology, 2022.

[51]. W. Li, S. Jia, L. Liu, F. Zheng, Y. Ma и J. Lin. Cryptogo: automatic detection of go cryptographic api misuses. В Proceedings of the 38th Annual Computer Security Applications Conference, страницы 318–31, 2022.

[52]. O. H. Veileborg, G.-V. Saioc и A. Møller. Detecting blocking errors in go programs using localized abstract interpretation. В Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, страницы 1–12, 2022.

[53]. A. Borodin, V. Dvortsova и A. Volkov. Interprocedural static analysis for Go with closure support. В 2022 Ivannikov Ispras Open Conference (ISPRAS), страницы 1–6. IEEE, 2022.

[54]. A. Galustov, A. Borodin и A. Belevantsev. Devirtualization for static analysis with low level intermediate representation. В 2022 Ivannikov Ispras Open Conference (ISPRAS), страницы 18–23. IEEE, 2022.

[55]. V. Dvortsova, A. Izbyshev, A. Borodin и A. Belevantsev. Static analysis for Go: build interception. В 2023 Ivannikov Ispras Open Conference (ISPRAS), страницы 52–57. IEEE, 2023.

[56]. Д. Н. Субботин, А. Е. Бородин и В. В. Дворцова. Статический анализ ассоциативных массивов в Go. Труды Института системного программирования РАН, 36(3):21–34, 2024.

[57]. B. Liu и D. Joshi. Goguard: efficient static blocking bug detection for Go. В International Static Analysis Symposium, страницы 216–241. Springer, 2024.

[58]. Y. Zhang, B. Li, J. Lin, L. Li, J. Bai, S. Jia и Q.Wu. Gopher: high-precision and deep-dive detection of cryptographic api misuse in the go ecosystem. В Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security, страницы 2978–2992, 2024.

[59]. S. Fu и Y. Liao. Golang defect detection based on value flow analysis. В 2024 9th International Conference on Electronic Technology and Information Science (ICETIS), страницы 358–363. IEEE, 2024.

[60]. C. Cesarano, V. Andersson, R. Natella и M. Monperrus. Gosurf: identifying software supply chain attack vectors in Go. arXiv preprint arXiv:2407.04442, 2024.

[61]. B. Ding, Q. Li, Y. Zhang, F. Tang и J. Chen. Mea2: a lightweight field-sensitive escape analysis with points-to calculation for Golang. Proceedings of the ACM on Programming Languages, 8(OOPSLA2):1362–1389, 2024.

[62]. J. Chen, B. Ding, Y. Zhang, Q. Li и F. Tang. An empirical study of Cgo usage in Go projects–distribution, purposes, patterns and critical issues. Purposes, Patterns and Critical Issues.

[63]. Google scholar. https://scholar.google.com. Доступ: 2025-02-11.

[64]. Ieee xplore. https://ieeexplore.ieee.org/Xplore/home.jsp. Доступ: 2025-02-11.

[65]. Acm digital library. https://dl.acm.org. Доступ: 2025-02-11.

[66]. Springerlink. https://link.springer.com. Доступ: 2025-02-11.

[67]. Github. https://github.com. Доступ: 2025-02-11.

[68]. Goroutines. https://go.dev/doc/effective_go#goroutines. Доступ: 2025-02-11.

[69]. Go channels. https://go.dev/doc/effective_go#channels. Доступ: 2025-02-11.

[70]. C. Hoare. Communicating sequential processes. В Theories of Programming: The Life and Works of Tony Hoare, страницы 157–186. 2021.

[71]. T. Tu, X. Liu, L. Song и Y. Zhang. Understanding real-world concurrency bugs in go. В Proceedings of the twenty-fourth international conference on architectural support for programming languages and operating systems, страницы 865–878, 2019.

[72]. K. Honda, N. Yoshida и M. Carbone. Multiparty asynchronous session types. В Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages, страницы 273–284, 2008.

[73]. Migoinfer+. https://github.com/JujuYuki/gospal. Доступ: 2025-02-11.

[74]. Godel 2 benchmarks. https://github.com/JujuYuki/godel2-benchmark. Доступ: 2025-02-11.

[75]. O. Bunte, J. F. Groote, J. J. Keiren, M. Laveaux, T. Neele, E. P. de Vink, W. Wesselink, A. Wijs и T. A. Willemse. The mcrl2 toolset for analysing concurrent systems: improvements in expressivity and usability. В Tools and Algorithms for the Construction and Analysis of Systems: 25th International Conference, TACAS 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6–11, 2019, Proceedings, Part II 25, страницы 21–39. Springer, 2019.

[76]. Promela. https://en.wikipedia.org/wiki/Promela. Доступ: 2025-02-11.

[77]. Spin. https://spinroot.com/spin/whatispin.html. Доступ: 2025-02-11.

[78]. Gomela. https://github.com/nicolasdilley/gomela-ase21/. Доступ: 2025-02-11.

[79]. Gcatch. https://github.com/system-pclub/GCatch. Доступ: 2023-10-05.

[80]. Crypto. https://pkg.go.dev/crypto. Доступ: 2025-02-11.

[81]. Crypto. https://pkg.go.dev/golang.org/x/crypto. Доступ: 2025-02-11.

[82]. Go-safer. https://github.com/jlauinger/go-safer. Доступ: 2025-01-15.

[83]. Goanalysis. https://github.com/chrisbbe/GoAnalysis. Доступ: 2025-02-11.

[84]. T. J. McCabe. A complexity measure. IEEE Transactions on software Engineering, (4):308–320, 1976.

[85]. S. M. Olbrich, D. S. Cruzes и D. I. Sjøberg. Are all code smells harmful? A study of god classes and brain classes in the evolution of three open source systems. В 2010 IEEE international conference on software maintenance, страницы 1–10. IEEE, 2010.

[86]. R. Harrison, S. J. Counsell и R. V. Nithi. An evaluation of the mood set of object-oriented software metrics. IEEE Transactions on Software Engineering, 24(6):491–496, 2002.

[87]. R. Subramanyam и M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: implications for software defects. IEEE Transactions on software engineering, 29(4):297–310, 2003.

[88]. P. Anderson, D. Binkley, G. Rosay и T. Teitelbaum. Flow insensitive points-to sets. Information and Software Technology, 44(13):743–754, 2002. DOI: https://doi.org/10.1016/S0950-5849(02)00105-2. URL: https://www.sciencedirect.com/science/article/pii/S0950584902001052. Special Issue on Source Code Analysis and Manipulation (SCAM).

[89]. Coverity 2021.03: Supported Platforms. Доступ: 2025-02-11. 2021. URL: https://sigdocs.synopsys.com/polaris/topics/r_coveritycompatible-platforms_2021.03.html.

[90]. Common weakness enumeration. https://cwe.mitre.org. Доступ: 2024-10-01.

[91]. Badgerdb. https://github.com/hypermodeinc/badger. Доступ: 2025-02-11.

[92]. J. Hu, L. Zhang, C. Liu, S. Yang, S. Huang и Y. Liu. Empirical analysis of vulnerabilities life cycle in Golang ecosystem. В Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, страницы 1–13, 2024.

[93]. About - git. https://git-scm.com/about/data-assurance. Доступ: 2025-02-11.

[94]. Go modules services. https://proxy.golang.org/. Доступ: 2025-02-11.

[95]. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman и F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. ACM Trans. Program. Lang. Syst., 13(4):451–490, окт. 1991. DOI: 10.1145/115372.115320. URL: https://doi.org/10.1145/115372.115320.

[96]. Ssadump. https://pkg.go.dev/golang.org/x/tools/cmd/ssadump. Доступ: 2023-10-05.

[97]. C. Lattner и V. Adve. A compilation framework for lifelong program analysis and transformation. В CGO, том 4, страница 75, 2003.

[98]. Gollvm is an llvm-based Go compiler. https://go.googlesource.com/gollvm/. Доступ: 2024-10-05.

[99]. N. Malyshev, I. Dudina, D. Kutz, A. Novikov и S. Vartanov. Smt solvers in application to static and dynamic symbolic execution: a case study. В 2019 Ivannikov Ispras Open Conference (ISPRAS), страницы 9–15. IEEE, 2019.

[100]. W. Landi. Undecidability of static analysis. ACM Letters on Programming Languages and Systems (LOPLAS), 1(4):323–337, 1992.

Информация об авторах / Information about authors

Варвара Викторовна ДВОРЦОВА – сотрудник ИСП РАН, аспирант ИСП РАН. Сфера научных интересов: компиляторные технологии, статический анализ, анализ Golang.

Varvara Viktorovna DVORTSOVA – ISP RAS researcher, postgraduate student at ISP RAS. Her research interests: compiler technologies, static analysis, Golang analysis.

Алексей Евгеньевич БОРОДИН – кандидат физико-математических наук, старший научный сотрудник ИСП РАН. Сфера научных интересов: статический анализ исходного кода программ для поиска ошибок.

Alexey Evgenevich BORODIN – Cand. Sci. (Phys.-Math.), senior researcher. His research interests: static analysis for finding errors in source code.