ПРОГРАММИРОВАНИЕ, 2021, № 5, с. 12–21

АНАЛИЗ И ТРАНСФОРМАЦИЯ ПРОГРАММ

УДК 004.421.6

МЕЖПРОЦЕДУРНЫЙ СТАТИЧЕСКИЙ АНАЛИЗ ДЛЯ ПОИСКА ОШИБОК В ПРОГРАММАХ НА ЯЗЫКЕ GO

© 2021 г. И. В. Болотников[a,b,*], А. Е. Бородин[a,]**

[a] *Институт системного программирования им. В.П. Иванникова РАН, 119333 Москва, ул. Александра Солженицына, д. 25, Россия*

[b] *Московский государственный университет им. М.В. Ломоносова, 119991 Москва, ГСП-1, Ленинские горы, д. 1, Россия*

**E-mail: igor.bolotnikov@ispras.ru*

***E-mail: alexey.borodin@ispras.ru*

Поступила в редакцию 12.04.2021 г.
После доработки 21.04.2021 г.
Принята к публикации 11.05.2021 г.

За последние годы популярность языка Go значительно возросла. Вместе с тем в настоящее время для языка Go существуют только легковесные статические анализаторы. Мы восполнили этот пробел, адаптировав статический анализатор Svace для поиска ошибок в программах на языке Go. Нами был реализован межпроцедурный и межмодульный статический анализатор имеющий чувствительность к потоку и путям. Для оценки результатов использовалось 10 проектов с откры- тым исходным кодом. 16 оцениваемых детекторов выдали 6817 предупреждений с 76 срабатываний.

DOI: 10.31857/S0132347421050034

1. ВВЕДЕНИЕ

Язык программирования Go – компилируемый, строго типизированный, многопоточный язык программирования, созданный компанией Google в 2009 году. Применяется в основном в backend-е web-приложений [1], что не помешало ему попасть в 20 самых популярных языков программирования на момент написания статьи [2, 3].

Разработчики языка старались максимально обезопасить язык от часто допускаемых программистами ошибок. Языком не предусмотрено неявное приведение типов. Все переменные по умолчанию инициализированы нулевым значением, переполнение буфера не приводит к уязвимости, реализован механизм сборки мусора, помогающий бороться с утечками памяти.

Компилятор Go занимается поиском распространенных тривиальных ошибок, например, ситуаций с объявлением переменной, которая нигде не используется. Важным преимуществом языка Go является высокая скорость компиляции, так как при конструировании компилятора на первое место выходит скорость сборки и качество оптимизаций, а не поиск дефектов. Поэтому необходимость дополнительного статического анализа им не устранена.

Чтобы не перегружать компилятор, разработчики языка реализовали открытый статический анализатор go vet, промежуточным представлением анализа которого является абстрактное синтаксическое дерево (АСД) языка Go. На момент написания работы он насчитывал 21 вид предупреждений. Вот некоторые из них:

- copylocks – ошибочная передача lock-объекта по значению;
- nilfunc – избыточное сравнение функции с нулем;
- printf – несоответствие аргументов форматной строке;
- unusedresult – неиспользуемое возвращаемое значение.

Для языка существует также несколько подобных анализаторов, которые называют линтерами – staticcheck [4], go-critic [5], errcheck [6].

Целью работы была разработка статического анализатора языка Go с возможностями глубокого межпроцедурного семантического анализа. Все перечисленные выше анализаторы Go данным свойством не обладают.

Для решения данной задачи мы решили расширить существующий инструмент ИСП РАН – статический анализатор Svace [7–10]. Анализатор первоначально был создан для анализа кода на C/C++, позже расширен для анализа программ на Java [11] и Kotlin.

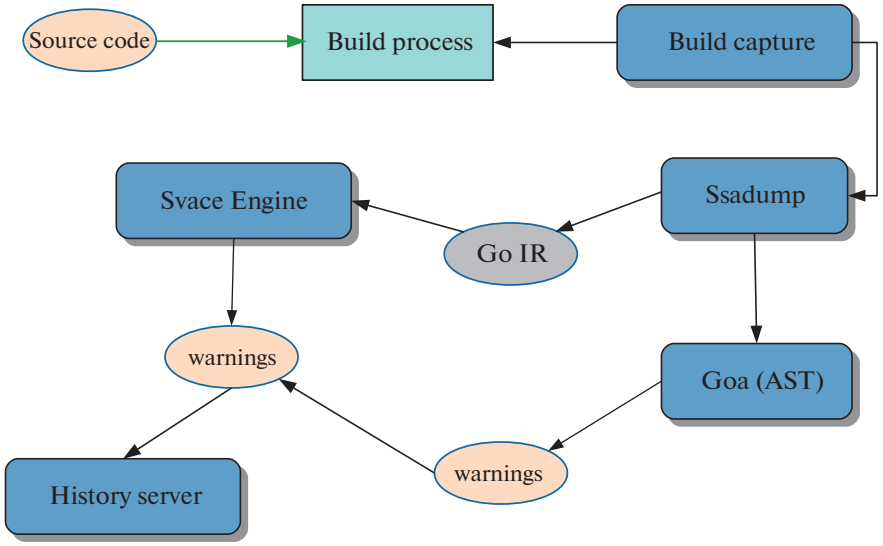

Рис. 1. Схема анализа.

2. АНАЛИЗАТОР SVACE

Целью анализатора является поиск как можно большего количества ошибок при приемлемом уровне ложных срабатываний. Выполняется межпроцедурный и межмодульный анализ на основе резюме. Анализ учитывает алиасы и значения переменных, моделирует содержимое полей в структурах и элементах массивов. Список реализованных детекторов включает в себя поиск ошибок разыменования нулевых указателей, переполнения массивов, утечки ресурсов, бесконечные циклы, небезопасное использование данных из внешних источников.

Типичная схема анализа выглядит следующим образом:

1. На вход анализатору подается исходный код и скрипт сборки.

2. Специальный компонент build-capture перехватывает команды запуска компилятора.

3. Модифицированный компилятор строит абстрактное синтаксическое дерево (АСД), которое подается на вход АСД-анализатору.

4. Также модифицированный компилятор генерирует промежуточное представление программы для последующего анализа.

5. Промежуточное представление подается на вход анализатору SvEng (Svace Engine – инфраструктура основного анализа Svace)

Чтобы анализатор мог работать с языком Go, были выполнены следующие шаги:

• Дополнена процедура перехвата сборки: расширен компонент build-capture. Реализованы только случаи для непосредственной компиляции вызовами компилятора Go.

• Выбрано промежуточное представление Go, достаточно близкое к промежуточному представлению анализатора. Реализован его генератор и парсер. Дополнено промежуточное представление анализатора.

• Написаны уникальные для Go анализы в рамках SvEng.

В приложении А на рис. 1 изображена реализованная для Go схема анализа. Обычно для генерации промежуточного представления используется компилятор. Для языка Go удалось обойтись без модификации компилятора, что является не только сложной процедурой, но и влечет за собой высокую стоимость поддержки. Вместо компилятора используется утилита ssadump, которая будет описана в разделе 3.2. Утилита ssadump генерирует промежуточное представление, а также запускает АСД анализатор, названный goa. Утилита SvEng выполняет межпроцедурный анализ для сгенерированного представления.

3. ПРОМЕЖУТОЧНОЕ ПРЕДСТАВЛЕНИЕ

3.1. Svace IR

Svace IR – промежуточное представление анализатора SvEng. Собственное промежуточное представление позволяет анализировать программы на различных языках программирования одним и тем же анализатором. Особенностью Svace IR является его схожесть с промежуточным представлением LLVM и нахождение в частичной SSA-форме. Более подробное описание можно посмотреть в [12, с. 32–41].

3.2. Промежуточное представление языка Go для Svace

В качестве промежуточного представления было выбрано SSA-представление инструмента ssadump, находящегося в свободном доступе и распространяемого с основными вспомогательными инструментами языка Go в наборе пакетов `golang.org/x/tools`. Данный выбор был обусловлен схожестью промежуточного представления ssadump и Svace IR и отсутствием более близких аналогов промежуточного представления языка Go. В качестве формата для генерации был выбран JSON из-за его простоты реализации, поддержки и самое важное на данный момент – простоты отладки и читабельности программистом. Этот формат не является оптимальным по скорости генерации/чтения и занимаемой памяти и при необходимости будет изменен.

Внутри ssadump был реализован JSON-генератор промежуточного представления. На стороне анализатора соответственно был реализован парсер. Отображение инструкций одного представления на другое осуществлено в два этапа. Те инструкции, которые имели аналоги в Svace IR, отобразились напрямую. Это основные операции, которые есть в большинстве языков, такие как логические, арифметические операции, вызовы, переходы и т.п. Однако в промежуточное представление ssadump входят специфические для языка инструкции, такие как:

• defer – отложенный вызов функции;

• go – запуск go-рутины с заданной функцией;

• typeassert – приведение типа, которое выделено в отдельную инструкцию и имеет две вариации:

– строгое – инструкция возвращает единственный результат в случае успеха или иначе приводит к ошибке выполнения;

– нестрогое – инструкция возвращает два значения. Первое – результат приведения типа, второе – успешно ли приведение. Если не успешно, то в первом значении может возвращаться любое “мусорное” значение;

• extract – извлечение элемента кортежа;

• select, send – инструкции для работы с Go-каналами;

• makeClosure – инструкция создания lambda-объекта из заданной функции и перечисленных переменных из окружения точки вызова;

• вызовы встроенных в язык функций, которые имеют особый статус в Go (append, len, copy и др.);

• make* – инструкции создания различных объектов строенных небазовых типов;

• mapUpdate – вставка пары в map;

• lookup – инструкция для доступа к элементу строки и map с проверкой ключа;

• range, next – инструкции итерирования по Go-коллекциям (map, slice, array, string, channel).

Все вышеперечисленные инструкции ssadump были добавлены в виде инструкций Svace IR или в качестве спецификаций (builtin и make*), как описано в разделе 5.2.

Также был расширен набор типов в Svace IR:

• tuple – кортеж. Данный тип появляется в Go неявно, когда функция возвращает сразу несколько значений, которые могут быть различного типа;

• slice – встроенный тип динамического массива;

• map – встроенный тип отображения;

• chan – Go-канал для коммуникации между Go-рутинами (ближе всего к pipe в C);

• Go-интерфейс – тип, определяющий набор функций, которые должен поддержать другой тип. Не требует явного описания implements. Утиная типизация.

4. ПЕРЕХВАТ СБОРКИ

Процесс перехвата запускает оригинальную команду сборки, инструментируя ее таким образом, чтобы перехватывать все запускаемые процессы и выделять интересующие нас команды сборки. В данном случае вызовы компилятора Go, однако допустимо расширение и на другие инструменты сборки. Важно не повлиять при этом на исходную сборку, чтобы ее результаты совпадали с выполнением сборки без внешнего вмешательства. Подробно о перехвате сборки написано в работе [13].

Если в процессе сборки выясняется, что была запущена команда компиляции какого-то набора пакетов, то параллельно запускается скрипт, который находит пакеты-зависимости, фильтрует полученную совокупность и для оставшихся запускает модифицированную версию ssadump, в результате работы которой будет получена информация о промежуточном представлении оставшихся пакетов. Фильтр имеет несколько настроек. В первую очередь отбрасываются пакеты, для которых промежуточное представление уже было построено ранее. Для этого перед запуском ssadump для каждого пакета в файловой системе создается lock-файл, обладающий уникальностью, для набора параметров – имя пакета и имя рабочей директории, последнее необходимо для учета команд вида replace в случае использования Go-модулей.

Дополнительная возможность фильтрации появляется в том случае, когда пользователь использует Go-vendoring – режим компиляции, где часть зависимостей зафиксирована. Данный код располагается отдельно от остального исходного кода, редко обновляется, а также не является

пользовательским. В таком случае в инструмент сборки добавлена опция, позволяющая игнорировать зависимости, находящиеся в vendor, что ускорит сборку и анализ оставшегося кода, но ограничит его глубину и соответственно может повлиять на качество предупреждений.

В качестве анализируемой единицы используется пакет, а не файл, так как ssadump изначально оперировал данными на уровне пакета. Вся информация собиралась сразу для пакета, и не было причин для ее дополнительного дробления на файлы. Если же брать АСД для Go, то там дерево строится для каждого файла по отдельности, и в существующем виде там неполна даже информация о типах используемых переменных. В других анализаторах эта проблема решается с помощью аналогичного вспомогательного анализа набора АСД, которые относятся к одному и тому же пакету. По тем же причинам goa анализирует тоже не чистый АСД, а АСД с дополнительной информацией о пакетах, сбор которой не приводит к потерям производительности, так как она используется для генерации промежуточного представления для основного анализатора Svace.

В итоге работы ssadump в зависимости от аргументов будут сгенерированы следующие данные для каждого из пакетов:

• **Файл с промежуточным представлением** – JSON-файл с таблицей типов, таблицей символов и промежуточным представлением функций одного пакета. Нужен для построения промежуточного представления анализатора Svace.

• **Файл с графом вызовов.** Нужен для построения графа вызовов в анализаторе Svace. Для остальных языков программирования граф вызовов строится при обычном чтении файлов с промежуточным представлением. Для Go промежуточное представление проектировалось с нуля. Так как анализатор имеет фазу чтения графа вызовов, то с целью оптимизации был создан отдельный файл с графом вызовов.

• **Файлы, описывающие расположение объявлений и использований переменных, типов и функций** в CSV формате, которые необходимы для более удобного взаимодействия пользователя с интерфейсом выданных предупреждений.

• **Файл с предупреждениями, полученными от goa.** Анализ АСД требует некоторое количество информации, которая собирается во время сборки, поэтому выгоднее по времени проводить его сразу после сборки в рамках одного и того же процесса, так как goa реализован на том же языке, что и ssadump – на языке Go.

5. РАСШИРЕНИЕ АНАЛИЗАТОРА

5.1. Краткое описание

Инструмент SvEng использует межпроцедурный анализ на основе резюме. При таком анализе вначале строится граф вызовов, а затем функции обходятся таким образом, чтобы вызываемые функции анализировались до вызывающих. Циклы в графе вызовов принудительно разрываются. После построения графа вызовов выполняется предварительная потоко-нечувствительная фаза, во время которой собирается информация о вызываемых функциях и используемых константах.

После анализа функции создается ее резюме, которое затем используется для анализа инструкций вызова функции. Для отдельной функции выполняется анализ на основе символьного выполнения с объединением состояний в точках слияния путей. Анализ моделирует значения переменных, полей структур, элементы массивов и ячеек памяти, до которых можно дотянуться с помощью разыменований и вызовов. Для описания свойств используются атрибуты, которые разделяются между различными детекторами. В настоящий момент реализовано более 350 атрибутов.

Для реализации чувствительности к путям строятся формулы, описывающие условия возникновения ошибки, и в тех случаях, когда детектор может выдать предупреждение, запускается SMT-решатель для проверки выполнимости формулы.

5.2. Спецификации

Для моделирования библиотечных функций в Svace используются спецификации. Спецификация описывает поведение функции и представляет собой еще одно определение функции на анализируемом языке. Спецификации могут содержать вызовы специальных функций (спец-функций), которые нигде не определены, но имеют особую семантику для анализатора. Анализ спецификаций производится так же, как и всех остальных функций, и в результате анализа будет создано резюме, которое будет использоваться для анализа вызова функции. Svace имеет спецификации для часто используемых библиотек, кроме этого, пользователь может добавлять свои спецификации.

Мы написали более 170 спецификаций для следующих пакетов: bytes, crypto, database, encoding, errors, flag, fmt, io, log, os, strconv, strings, zap.

В языке Go есть встроенные в компилятор функции, такие как *len*, *make*. Для них мы создали файл builtin.go, где можно писать спецификации для предопределенных функций. Таким образом, поведение этих функций не жестко задано в анализаторе, а вынесено в настраиваемую часть.

Пример спецификации:

```
func makemap(size interface{}) inter-
face{} {
    Sf_set_trusted_sink_int(size)

    var res interface{}
    Sf_overwrite(&res)
    return res
}

func len(v interface{}) int {
    var res int
    Sf_overwrite(&res)
    Sf_assert_cond(res, ">=", 0)

    if res != 0 {
       Sf_assert_cond_ptr(v, "!=", nil)
  }

  Sf_pure(res, v)
  return res
}
```

В приведенных спецификациях спец-функция *Sf_set_trusted_sink_int* сообщает анализатору, что параметр *size* не должен принимать данные из непроверенных источников. Спец-функция *Sf_pure* означает, что результат функции *len* зависит только от ее аргумента. Спец-функция *Sf_assert_cond* сообщает, что возвращаемое значение не отрицательное.

5.3. Моделирование особенных для Go инструкций

Язык Go имеет инструкцию `defer`, которая помещает вызов функции в стек и выполняет функции из этого стека, когда завершается вызывающая функция. В Go IR находится инструкция `defer`, для анализа же более интересны сами вызовы функций. Вся необходимая логика была реализована в плагине, который запоминает аргументы инструкции `defer`, а в точках завершения функции смотрит на запомненные аргументы и для всех них запускает обработку инструкции вызова функции. Фактически эмулируется представление, где удалены все вызовы инструкции defer, а в точках выхода из функции добавлены инструкции вызова соответствующих функций.

Язык Go имеет поддержку для упрощения многопоточного программирования. Go-рутина позволяет запустить вызов функции параллельно. Благодаря встроенной в язык поддержке эта инструкция активно используется. Анализатор Svace не имеет анализа для функций, выполняющихся параллельно. Мы решили просто пометить, что вызов происходит параллельно, а затем анализировать его как обычный вызов функции. Такое поведение вызова go-рутины является допустимым, но не исчерпывает все возможные варианты.

Функции в Go могут возвращать несколько значений с помощью кортежей (tuples). Мы расширили Svace IR, теперь все функции могут возвращать несколько значений. Кроме этого, потребовалось менять множество обработчиков возвращаемых значений и резюме. В язык был добавлен новый тип данных – кортеж, при этом значения кортежей моделируются существующим структурным типом с соответствующими полями.

5.4. Существующие детекторы

Для языка Go мы включили некоторые детекторы, работающие для других языков. Мы не включали все детекторы, а сосредоточили свои усилия только на некотором подмножестве. В большинстве случаев не потребовалось никакой их адаптации к коду Go и промежуточному представлению.

Список включенных детекторов:

• DEREF_AFTER_NULL – неконсистентная работа с указателями: в коде есть проверка указателя на ноль, а также разыменование без проверки на ноль;

• DEREF_AFTER_NULL.EX – усовершенствованная версия DEREF_AFTER_NULL, использующая SMT-решатель;

• DEREF_OF_NULL.RET.EX – функция возвращает ноль, который затем разыменовывается;

• OVERFLOW_UNDER_CHECK – обращение к массиву по индексу для которого есть проверка диапазона, но она не исключает переполнения массива;

• TAINTED_INT – использование непроверенных данных из внешних источников в критических операциях;

• DIVISION_BY_ZERO.EX – ошибка деления на ноль. Источником является либо нулевая константа, либо инструкция сравнения с нулем;

• DIVISION_BY_ZERO.UNDER_CHECK – деление на число, значения которого проверили, при этом нулевое значение не исключено;

• REDUNDANT_COMPARISON.ALWAYS_FALSE – проверка заведомо ложного условия на истинность в условном операторе.

Пример срабатывания DEREF_AFTER_NULL для проекта etcd (etcd/embed/serve.go):

```
func (ac *accessController)
ServeHTTP(rw http.ResponseWriter, req *http.Request) {

        if req != nil && req.URL != nil
            && strings.HasPrefix(req.URL.Path, "/v3beta/") {
                req.URL.Path = strings.Replace(req.URL.Path,
                            "/v3beta/", "/v3/", 1)

        }

        if req.TLS == nil {
```

В коде переменная *req* сравнивается с нулем, после указатель разыменовывается при доступе к полю *req.TLS*. Возможно, следует заменить *&&* на ||. Также для этого проекта была найдена межпроцедурная ошибка, задействующая 3 функции.

Детектор DEREF_OF_NULL.RET.EX потребовал существенной доработки. Основная причина – наличие в языке Go кортежей (tuple), которые используются при обработке ошибок. Типичный паттерн показан ниже:

```
func create(arg int) (*MyStruct, error) {
   if arg >= 0 {
       s := createImpl(arg)
       return s, nil
   }

   return nil, errors.New("Negative argument")
}
```

Таким образом, указатель будет нулем, только если *error* не ноль. Мы добавили в условие ошибки значения других элементов в кортеже. Также была добавлена специальная обработка error-части в кортеже. Для нее был создан атрибут, имеющий ссылку на потенциально нулевой указатель. При проверке error-части с помощью этого атрибута снимается информация о том, что указатель нулевой.

Для детекторов переполнения массивов потребовалось моделирование предопределенной функции *len*, которое было выполнено с помощью спецификаций (5.2).

5.5. Детекторы, специфичные для Go

Мы реализовали несколько детекторов для поиска подозрительных паттернов в исходном коде, которые могут быть полезны программистам на Go:

• UNCHECKED_TUPLE.RET – не проверяется error-часть кортежа;

• UNCHECKED_TUPLE.CHAN – версия UNCHECKED_TUPLE.RET для чтения из канала;

• DEREF_OF_NULL.GLOBAL – поиск ситуаций, когда используется глобальная переменная, которая нигде не инициализирована и соответственно имеет нулевое значение;

• INFINITE_LOOP.GOROUTINE – использование в качестве Go-рутины функции, содержащей бесконечный цикл;

• DEREF_OF_NULL.RET.GO_INTERFACE – некорректная проверка на ноль интерфейса.

В Go кортежи часто используются для обработки ошибок. Детектор UNCHECKED_TUPLE.RET выдает предупреждения для случаев, когда error-часть функций игнорируется:

```
func parse(par string) (*Res, error) {
    if par == "" {
       return nil, fmt.Errorf("...")
    }
    return getRes(par), nil
}

func use(para string) {
    res, _ := parse(para)
    //error part is ignored
    res.handle() //error
}
```

Заметим, что детектор является межпроцедурным и имеет чувствительность к путям. Поэтому добавление некорректной проверки не подавит предупреждение.

Выдачу детектора UNCHECKED_TUPLE.CHAN мы дополнительно ограничили случаями, когда чтение происходит в цикле и канал передается в Go-рутину, в которой он закрывается. Этот паттерн опасен, так как если канал будет закрыт, то цикл никогда не завершится.

Детектор DEREF_OF_NULL.GLOBAL является потоково-нечувствительным. Предупреждение выдается, только если нигде нет инициализации глобальной переменной. Реализация выполнена на предварительной фазе анализа, запускающейся после построения графа вызовов и доосновной фазы. Предварительная фаза по очереди анализирует все инструкции всех модулей.

Чтобы реализовать предупреждение INFINITE_LOOP.GOROUTINE, был расширен существующий детектор бесконечных циклов. На предварительной фазе собирается информация о функциях, запускаемых как Go-рутины. При обнаружении бесконечных циклов в таких функциях выдается предупреждение INFINITE_LOOP.GOROUTINE.

В языке Go интерфейс содержит информацию о значении и типе указателя. Проверка интерфейса на ноль не означает, что захваченное значение не равно нулю и его можно разыменовывать. Это может приводить к неочевидным ошибкам в коде:

```
var data *byte
var in interface{}

fmt.Println(data,data == nil)
//prints: <nil> true

fmt.Println(in,in == nil)
//prints: <nil> true

in = data
fmt.Println(in,in == nil)
//prints: <nil> false
```

Для последнего *Println* значение переменной *in* не равно нулю, несмотря на то, что ей был присвоен нулевой указатель.

Для предотвращения таких ошибок мы написали два типа детектора DEREF_OF_NULL.RET.GO_INTERFACE. Первая версия выдает предупреждения, если указатель, который может иметь нулевое значение, возвращается из функции с типом интерфейс. Вторая версия проверяет, что это возвращаемое значение затем разыменовывается. Оба детектора выдают предупреждения только для возвращаемых значений функций. Мы посчитали паттерн с возвращаемым значением наиболее опасным, т.к. в результате рефакторинга тип возвращаемого значения может поменяться с указателя на интерфейс.

6. АНАЛИЗ НА ОСНОВЕ АСД

У нас не было задачи дублировать существующие анализаторы. Тем не менее, любой статический анализатор только выиграет от наличия детекторов на основе АСД. Такие детекторы, как правило, имеют высокий уровень истинных срабатываний и могут находить множество опечаток.

На основе ssadump был реализован статический анализатор goa[1]. Анализатор предоставляется в двух вариантах:

• В рамках ssadump. В таком случае сразу после генерации промежуточного представления вся собранная информация передается goa;

• Независимо. Тогда goa самостоятельно пытается собрать информацию, которую ему в первом варианте предоставлял ssadump. В случае неудачи запускаются только те детекторы, которые работают непосредственно с АСД.

В Goa реализованы следующие виды предупреждений:

• INVARIANT_RESULT – предупреждение о выражении, результат которого известен на этапе компиляции и может быть заменен на константу;

• UNSAFE_TYPE_CONVERSION – предупреждение об арифметических операциях, допускающих более безопасное явное приведение типов, по сравнению с используемым, что позволяет избежать переполнение типов;

• UNSAFE_TYPE_SWITCH – предупреждение об отсутствии default ветки в type-switch выражении. Желательно указывать явно хоть и пустой default, тем самым явно показывая, что даже если появятся новые реализации данного интерфейса, функция не должна к ним адаптироваться;

• UNSAFE_TYPE_ASSERTION – предупреждение о том, что возможна runtime-ошибка в данной инструкции приведения интерфейса к другом типу (typeassert);

• LOOPVAR_IN_CLOSURE – предупреждение о ссылке внутри вложенной в цикл Go-рутины на переменную, которая может не быть постоянной на разных итерациях данного цикла. Аналогичное предупреждение есть в go vet. Наша версия отличается более широким подходом – анализируются не только последние инструкции цикла, и дополнительно выдаются предупреждения для Go-рутин методов с объектами, передаваемыми по указателю.

Детектор для INVARIANT_RESULT обходит вершины АСД в глубину и для каждой пытается

[1] От Go analyzer.

Таблица 1. Оценка размеров

Проект (https://github.com/*)	LOC проекта	Количество файлов в проекте	LOC проекта с зависимостями	Количество файлов проекта с зависимостями	Размер сгенерированного промежуточного представления (MB)
anacrolix/dht	3647	51	241540	531	25
taskctl/taskctl	4492	59	262344	562	54
unidoc/unioffice	9438	48	9438	48	289
quasilyte/go-ruleguard	11136	120	27362	203	20
percona/percona-server-mongodb-operator	12805	121	1080637	2923	889
nanovms/ops	17416	156	789421	2127	788
jesseduffield/lazygit	22714	153	248406	975	133
pdfcpu/pdfcpu	48703	186	58169	212	83
prometheus/prometheus	96971	331	1219480	4104	1066
ovh/cds	199302	1323	1286867	5846	1533

применить по очереди одно из правил, список которых может быть довольно широким. Примером такого правила служит проверка всех вершин, являющихся бинарными выражениями **a** ⩽ **b**. Оно точно является известным на этапе компиляции, если размер типа **a** меньше, чем значение операнда **b**.

Детектор UNSAFE_TYPE_CONVERSION обходит все выражения приведения целочисленных типов. Если внутри приведения арифметическое выражение над типами меньшей размерности, то возможно переполнение типа во время арифметической операции, чего не могло бы возникнуть, будь расширение типов проведено перед операцией. Рассмотрим пример:

```
func example(slice []byte) {
    length := len(slice)
    for {
        if int(slice[0])+1 > length {
        return
        }
        /* UNSAFE_TYPE_CONVERSION */
        length -= int(slice[0] + 1)
        if length == 0 {
           break
        }
        slice = slice[slice[0]+1:]
    }
}
```

Если значение slice[0] на входе в цикл равно 255, а len(slice) больше или равно 256, то он будет бесконечным, так как выражение int(slice[0] + 1) будет всегда равно 0 из-за переполнения типа.

Детектор LOOPVAR_IN_CLOSURE обходит в глубину АСД. При достижении цикла он начинает собирать переменные (варианты), значение которых зависит от итераторов цикла. При виде узла go или defer детектор проверяет, является ли вызываемая функция анонимной и захватывает ли она какие-то значения извне, которые являются вариантами текущего или внешнего цикла. Если же это не анонимная функция, а метод для указательного типа, то детектор также проверяет, что объект, для которого этот метод вызывается, не является вариантом.

Пример для первого случая:

```
func( values []int) {
    var wait sync.WaitGroup
    wait.Add(len(values))
    for key, value := range values {
       go func() {
             /* LOOPVAR_IN_CLOSURE */
             fmt.Println(key, value)
             wait.Done()
       }()
    }
    wait.Wait()
}
```

Пример для второго:

```
func (v *val) MyPtrMethod() {
    fmt.Println(v.String())
}

func test(values []val) {
    for _, val := range values {
        /* LOOPVAR_IN_CLOSURE */
        defer val.MyPtrMethod()
    }
}
```

Таблица 2. Оценка времени сборки и анализов*

Проект (https://github.com/*)	Время сборки с goa (сек.)	Время сборки без goa (сек.)	Время основного анализа (сек.)	Время первичной сборки без вмешательства (сек.)
anacrolix/dht	26.077	21.469	16.335	13.097
taskctl/taskctl	18.398	17.987	61.524	4.432
unidoc/unioffice	38.718	33.815	329.054	11.959
quasilyte/go-ruleguard	10.139	9.745	15.675	6.477
percona/percona-server-mongodb-operator	207.338	197.267	393.428	68.578
nanovms/ops	160.655	159.805	172.399	47.506
jesseduffield/lazygit	33.314	30.287	74.635	13.753
pdfcpu/pdfcpu	15.73	14.615	50.001	5.838
prometheus/prometheus	255.099	240.48	404.897	75.85
ovh/cds	280.615	268.112	365.773	66.142
всего	1031.926	777.150	1883.721	245.367

* Данные приведены для Ubuntu 20.04, RAM: 32 Gb, CPU: Intel Core i7-7700 3.60GH

7. РЕЗУЛЬТАТЫ

Для оценки результатов мы выполнили анализ для 10 проектов с открытым исходным кодом. В табл. 1 приведены данные для этих проектов, включающие размеры проектов в строках кода, количество файлов, размер проекта и количество файлов вместе с зависимостями, а также размер сгенерированного промежуточного представления.

Таблица 2 содержит данные о времени сборки и анализов для каждого из проектов. Вторая колонка содержит время перехвата сборки вместе с запуском анализатора goa, третья – только время перехвата сборки. Четвертая колонка содержит время полновесного анализа, и пятая колонка – время оригинальной сборки. Время инструментальной сборки замедляет основную сборку. В худшем случае время сборки увеличилось в 4.24 раза для проекта cds. Время анализа в среднем больше, чем время сборки. Наибольшее время анализа относительно оригинальной сборки было для проекта

Таблица 3. Качество детекторов*

Детектор	всего	TP rate
DEREF_AFTER_NULL	34	70%
DEREF_AFTER_NULL.EX	141	20%
DEREF_OF_NULL.RET.EX	1212	50%
DIVISION_BY_ZERO.EX	12	45%
DIVISION_BY_ZERO.UNDER_CHECK	7	100%
UNSAFE_TYPE_ASSERTION	1286	75%
LOOPVAR_IN_CLOSURE	11	90.9%
UNSAFE_SWITCH	1915	75%
UNSAFE_TYPE_CONVERSION	90	80%
UNSAFE_TYPE_SWITCH	269	100%
INFINITE_LOOP.GOROUTINE	10	100%
INVARIANT_RESULT	14	85.7%
OVERFLOW_UNDER_CHECK	3	100%
REDUNDANT_COMPARISON.ALWAYS_FALSE	34	85%
TAINTED_INT	3	100%
UNCHECKED_TUPLE.RET	1776	40%

* Для разметки выбиралось двадцать случайных предупреждений или все, если их было выдано меньше.

unioffice (27.5). В среднем перехват сборки медленнее оригинальной сборки в 3.16 раза, перехват вместе с goa – в 4.2 раза, время анализа – в 7.67 раза. Перехват вместе с анализом медленнее оригинальной сборки в 11.88 раза. Мы считаем это приемлемым временем анализа за возможность найти нетривиальные ошибки.

Для анализируемых проектов было выдано 6817 предупреждений шестнадцатью детекторами. Для каждого типа мы вручную разметили хотя бы 20 предупреждений для оценки качества детекторов. Результаты приведены в табл. 3. В среднем доля истинных срабатываний составила 76%. Процент истинных является достаточно высоким, тем не менее, хотелось бы его улучшить в следующих версиях анализатора, а также увеличить количество покрываемых ошибочных случаев.

Основным источником ложных срабатываний был недостаток спецификаций о библиотечных функциях, особенно о функциях, возвращающих кортежи с зависимыми данными.

Для шести предупреждений были отправлены отчеты об ошибках разработчикам проектов. На данный момент было получено три ответа:

1. golang.org/x/text – выход за пределы массива (issue 42147). Исправлено;

2. github.com/pdfcpu/pdfcpu – разыменование нулевого указателя (issue 303). Исправлено;

3. github.com/minio/minio-go – избыточное сравнение указателя с nil в одном участке кода либо ошибочное разыменование nil в другом (issue 1457). Отклонено, как не имеющее ценности.

8. ЗАКЛЮЧЕНИЕ

Насколько нам известно, мы построили инструмент статического анализа для Go, не имеющий аналогов. Анализатор позволяет находить межпроцедурные и межмодульные дефекты со средним процентом истинных срабатываний, равным 76. Было отправлено 6 отчетов об ошибках разработчикам проектов с открытым исходным кодом. Для трех из них был получен ответ и две ошибки были исправлены.

Будем рады любой критике, а также предложениям по усовершенствованию анализатора.

СПИСОК ЛИТЕРАТУРЫ

1. Golang-2019 Survey. https://blog.golang.org/survey2019-results. Accessed: 2020-10-3.
2. PYPL. http://pypl.github.io/PYPL. Accessed: 2020-10-3.
3. IEEE Spectrum's programming languages top. https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019. Accessed: 2020-10-3.
4. StaticCheck main page. https://staticcheck.io. Accessed: 2020-10-10.
5. go-critic main page. https://github.com/go-critic/go-critic. Accessed: 2020-10-10.
6. errcheck main page. https://github.com/kisielk/errcheck. Accessed: 2020-10-10.
7. *Бородин А.Е., Дудина И.А.* Внутрипроцедурный анализ для поиска ошибок на основе символьного выполнения // Труды Института системного программирования РАН. 2020. Т. 32. № 6. С. 87–100. https://doi.org/10.15514/ISPRAS-2020-32(6)-7
8. *Бородин А.Е., Белеванцев А.А.* Статический анализатор Svace как коллекция анализаторов разных уровней сложности // Труды Института системного программирования ИСП РАН. 2015. Т. 27. № 2. С. 111–134. https://doi.org/10.15514/ISPRAS-2015-27(6)-8
9. *Belevantsev A. et al.* Design and Development of Svace Static Analyzers // In 2018 Ivannikov Memorial Workshop (IVMEM). 2018. P. 3–9.
10. *Ivannikov V.P.* Static analyzer Svace for finding defects in a source program code // Programming and Computer Software. 2014. V. 40. № 5. P. 265–275.
11. *Меркулов А.П., Поляков С.А., Белеванцев А.А.* Анализ программ на языке Java в инструменте Svace // Труды Института системного программирования РАН. 2017. Т. 29. № 3.
12. *Бородин А.Е.* Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++. Диссертация на соискание ученой степени кандидата физ.-мат. наук. ИСП РАН, 2016.
13. *Белеванцев А.А., Избышев А.О., Журихин Д.М.* Организация контролируемой сборки в статическом анализаторе Svace // Системный администратор. 2017. № 7–8. С. 135–139.