`DOI: 10.15514/ISPRAS-2024-36(3)-2

Статический анализ ассоциативных массивов в Go

[1,2] *Д.Н. Субботин,* ORCID: 0009-0007-4429-9680, *<daniil.subbotin@ispras.ru>*
[1] *А.Е. Бородин,* ORCID: 0000-0003-3183-9821, *<alexey.borodin@ispras.ru>*
[1,2] *В.В. Дворцова,* ORCID: 0000-0002-7424-8442, *<vvdvortsova@ispras.ru>*

[1] *Институт системного программирования им. В.П. Иванникова РАН, Россия, 109004, г. Москва, ул. А. Солженицына, д. 25.*
[2] *Московский государственный университет им. М.В. Ломоносова, Россия, 119991, г. Москва, Ленинские горы д. 1.*

Аннотация. В статье описывается статический анализ ассоциативных массивов в языке Go для поиска разыменования нулевого указателя при извлечении ключа. Кратко вводятся внутреннее представление и алгоритмы анализатора Svace, в рамках которого выполнялась работа, затем описываются необходимые изменения внутреннего представления, моделирование семантики ассоциативных массивов, как для внутрипроцедурного, так и для межпроцедурного анализа на основе резюме, предлагается детектор поиска разыменований нулевого указателя. Приводятся экспериментальные результаты на широком круге открытых проектов.

Ключевые слова: статический анализ; Svace; Go; анализ коллекций; символьное выполнение.

Для цитирования: Субботин Д.Н., Бородин А.Е., Дворцова В.В. Статический анализ ассоциативных массивов в Go. Труды ИСП РАН, том 36, вып. 3, 2024 г., стр. 21–34. DOI: 10.15514/ISPRAS–2024–36(3)– 2.

Static Analysis of Go Maps

[1,2] *D.N. Subbotin*, ORCID: 0009-0007-4429-9680, *<daniil.subbotin@ispras.ru>*
[1] *A.E. Borodin*, ORCID: 0000-0003-3183-9821, *<alexey.borodin@ispras.ru>*
[1,2] *V.V. Dvortsova*, ORCID: 0000-0002-7424-8442, *<vvdvortsova@ispras.ru>*

[1] *Ivannikov Institute for System Programming of the RAS, 25, Alexander Solzhenitsyn st., Moscow, 109004, Russia.*
[2] *Moscow State University, 1, Leninskie Gory, Moscow, 119991, Russia.*

Abstract. The paper describes static analysis of map in the Go language for dereferencing a null pointer when extracting a key from a map. The work has been done within the Svace static analyzer. We begin with introducing Svace intermediate representation and algorithms. Then we describe the IR changes needed for modeling Go maps and their semantics. We explain how intraprocedural analysis is performed and how the null dereference detector works. Then we proceed with a summary-based interprocedural analysis. We show evaluation results on a wide range of open source projects.

Keywords: static analysis; Svace; Go; collection analysis; symbolic execution.

For citation: Subbotin D.N., Borodin A.E., Dvortsova V.V. Static Analysis of Go Maps. Trudy ISP RAN/Proc. ISP RAS, 2024, vol. 36, issue 3, pp. 11–34 (in Russian). DOI: 10.15514/ISPRAS–2024–36(3)– 2.

1. Введение

В данной работе рассматривается задача моделирования семантики ассоциативных массивов[1] (*map*) в языке Go. Алгоритмы реализованы в статическом анализаторе Svace [1-3], расширены существующие детекторы поиска разыменований нулевых указателей с учётом семантики ассоциативных массивов.

1.1 Тип *map* в Golang

Отображение в Go – это структура данных, позволяющая хранить пары «ключ-значение». Синтаксис операций с ассоциативными массивами в языке Go похож на синтаксис операций с обычными массивами. Соответствующие операторы позволяют устанавливать и извлекать хранимые значения:

```
map[key1] = val
v := map[key2]
```

Если запрошенный ключ не существует, то будет возвращено нулевое значение соответствующего типа.

Для извлечения значений из ассоциативных массивов в Go также существует версия с присвоением результата двум переменным, которая позволяет не только получить значения, но и проверить наличие ключа в отображении:

```
v, ok := map[key]
```

Флаг наличия `ok` будет равен `true`, если ключ присутствует в отображении, и `false` иначе.

1.2 Промежуточное представление

Анализатор Svace на вход получает промежуточное представление, сгенерированное с помощью модифицированной утилиты ssadump [4], которая строит SSA [5, 6] форму программ Go. Затем это промежуточное представление преобразуется во внутреннее представление Svace (далее Svace IR)[2], общее для всех языков. Для упрощения в данной статье будем считать, что они эквивалентны.

Svace IR является низкоуровневым типизированным языком в SSA-форме. Основным преимуществом такого представления является простота анализа и точное моделирование семантики программ.

Опишем инструкции промежуточного представления, анализ которых был модифицирован в рамках данной работы:

- `res = lookup1(map,key)`. Инструкция принимает отображение `map` и ключ `key`. Возвращается значение, ассоциированное с ключом в отображении.
- `t = lookup2(map,key)`. Вариант инструкции `lookup1`, у которой возвращается кортеж t из пары значений `(val,status)`, где `val` – это ассоциированное с ключом значение, а `status` – булево значение, означающее наличие ключа в отображении.
- `res = extract(tuple,index)`. Инструкция принимает кортеж `tuple` и целочисленный индекс `index`. Возвращаемым значением является элемент по индексу в кортеже. Кортеж можно считать набором бит. Инструкция `extract`, зная номер элемента, извлекает нужные биты из всего набора. В отличие от инструкции `lookup`, обращение к памяти программы при этом не происходит. Заметим, что индекс всегда имеет корректное значение, так как генерируется

[1] Мы будем также использовать термин *отображение* как синоним для краткости.

[2] Построение собственного представления позволяет унифицировать анализ и проводить его одинаково для всех языков.

компилятором для промежуточного кода.

- `mapUpdate(map,key,value)`. Инструкция принимает отображение `map`, ключ `key` и значение `val`. Функция ассоциирует (запоминает) пару ключ-значение для соответствующего отображения.
- `res = deref(ref)`. Инструкция для указателя `ref` получает значение `res`, находящееся в указываемой памяти.

Кроме того, опишем инструкции, анализ которых не был изменен:

- `m = makemap()`. Инструкция возвращает новое отображение m.
- r = `call func(`arg_1, arg_2, ... arg_n`)`. Инструкция вызова функции `func`, где arg_1, ... arg_n являются ее аргументами.
- `ret = make_tuple(val1, val2)`. Инструкция создаёт кортеж из двух значений `val1` и `val2`.
- `return val`. Инструкция возвращает значение `val` из текущей функции.

Для удобства мы добавим assume-инструкции для всех выходных рёбер инструкций ветвления. Эти инструкции являются линейными и сообщают анализатору свойства, которые выполняются на соответствующем пути. Анализ будет игнорировать семантику инструкций ветвления, но использовать assume-инструкции.

В табл. 1 показано, как функция на языке Go транслируется промежуточное представление.

Табл. 1. Промежуточное представление анализатора для исходного кода на языке Go
Table 1. Analysis IR for Go source code

Исходный код	Код в IR
`func foo(m map[int]*string, key1 int, key2 int)(string, bool){`	`func foo(m map[int]*string, key1 int, key2 int) (string, bool) {`
`val := m[key1]`	`val = lookup1(m, key1)`

`res, ok := m[key2]`	`tuple = lookup2(m, key2)`
	`res = extract(tuple, 0)`
	`ok = extract(tuple, 1)`

`if ok {`	`if ok {` `assume(ok)`
`m[key1] := res`	`mapUpdate(m, key1, res)`
`} else {`	`} else {` `assume(not(ok))`
`m[key2] := val`	`mapUpdate(m, key2, val)`
`}`	`}`
`d := *res`	`d = deref(res)`
`return d, ok`	`res_tpl = make_tuple(d, ok)` `return res_tpl`
`}`	`}`

1.3 Анализ Svace

Svace использует анализ на основе символьного выполнения. Мы расширили его моделированием семантики ассоциативных массивов.

Сделанные изменения использовались в качестве основы для поиска ошибок разыменования нулевых значений. Это довольно важный случай, т. к. отображения возвращают нуль в случае отсутствия запрашиваемого элемента. Но наш алгоритм позволяет моделировать поток данных, проходящий через ассоциативные массивы, и

поэтому может быть использован и в других детекторах, например, для поиска уяз- вимостей с использованием данных из недостоверных источников.

2. Анализ

2.1 Символьное выполнение

Для анализа отдельной функции используется символьное выполнение с объединением состояний анализа в точках слияния путей [7]. Рассмотрим граф потока управления (ГПУ) для некоторой функции. В качестве вершин графа используются инструкции промежуточного представления Svace IR. С каждым ребром ГПУ ассоциируется абстрактное состояние, которое описывает свойства функции в этой точке. Шаг анализа для каждой вершины графа из абстрактного состояния на её входном ребре формирует абстрактное состояние для выходного ребра: ⟨instr, Cin⟩ → Cout. Если вершина графа имеет несколько входящих ребер, то для нее выполняется операция объединения состояний.

При символьном выполнении переменным помимо реальных значений (констант) могут сопоставляться символьные переменные и символьные выражения.

Пусть S – множество переменных программы, V – множество символьных выражений, C – множество абстрактных состояний. Будем использовать вспомогательную функцию `val`: $S \times C \rightarrow V$, которая для каждого символа возвращает ассоциированное с ним символьное выражение. Для краткости параметр C будем опускать, так как из контекста понятно какое абстрактное состояние имелось в виду.

2.2 Анализ полей структур и элементов массивов

Популярный подход анализа полей структур и элементов массивов основан на путях доступа[3] [8-10]. Путь доступа состоит из указателя и последовательности полей структур, либо индексов массива. Мы используем похожий подход, но разделяем пути доступа на смещение и разыменование. Смещение создаётся с помощью родительского символьного выражения и смещения указателя. Разыменование, помимо родительского символьного выражения, также требует абстрактное состояние, так как зависит от памяти программы. Множество путей доступа будем обозначать как *P*.

Таким образом, доступ к элементам массивов и полям структур моделируются с помощью последовательности смещений и разыменований. Смещение в структуре обозначается именем соответствующего поля.

$$f_a = \texttt{shift(s, name)}$$
$$\texttt{f = deref (C, } f_a\texttt{)}$$

Параметр C функции `deref` обозначает абстрактное состояние. Функция `shift` формирует символьное выражение на основе указателя на структуру и имени поля. Для массивов вместо имени поля используется символьная переменная для индекса массива.

$$e_a = \texttt{shift}(a, v_i)$$
$$\texttt{e = deref(}e_a\texttt{)}$$

Функция `shift` берёт начало массива и смещает его на v_i элементов (не байт), где v_i = `val(`i`)`, i – индекс массива. Так как наше представление типизированное, то нет проблем с различием указателей на структуры и массивы. В отличие от структур, мы используем символьную переменную v_i вместо имени поля, так как смещения могут быть не только на константные значения, но и определяться переменными. Это усложняет задачу из-за

[3] Английский термин – Access Path.

проблемы алиасов: для двух символьных выражений `shift(s, i1)` и `shift(s, i2)` в общем случае мы не знаем, могут ли индексы i_1 и i_2 равняться друг другу.

Элементы ассоциативных массивов мы моделируем аналогично элементам массивов. Фактически наш анализ считает ассоциативные массивы Go частным случаем массивов со специальной индексацией. Подобное моделирование не отражает то, как элементы могут находиться в памяти программы во время выполнения. Но, с точки зрения анализа, это не влияет на результаты, т. к. нам достаточно понимать, какие значения соответствуют ключам, а не как именно эти значения расположены в памяти программы.

3. Моделирование семантики ассоциативных массивов

3.1Детектор на неправильную проверку флага состояния

Рассмотрим листинг 1, на котором приведен пример кода, содержащего ошибку разыменования нулевого указателя.

На строке 1 из отображения извлекается значение `res` и флаг состояния `flag`. Если условие на строке 2 выполнено (флаг состояния равен `false`), то это значит, что указанного ключа не существует в отображении и извлеченное значение равно `nil`. В этом случае на 3-й строке возникнет ошибка разыменования нулевого указателя.

```
1 res, flag := m[key]
2 if !flag {
3     len(*res) // null-pointer dereference
4 }
```

Листинг 1. Разыменование нулевого указателя
Listing 1. Example of nil dereference

Данный код транслируется в промежуточное представление, показанное на листинге 2.

```
1 tuple = lookup2(m, key)
2 res = extract(tuple, 0)
3 flag = extract(tuple, 1)
4 if !flag {
5     assume not(flag)
6     d = deref(res)
7     call len(d)
8 }
```

Листинг 2. Промежуточное представление для примера с разыменованием
Listing 2. IR for dereference example

Рассмотрим, как анализируется каждая из представленных функций. Для этого нам потребуется ряд вспомогательных функций. Функция `create_tuple`: $V \times P \rightarrow V$ для заданного символьного выражения и пути доступа создаёт новое символьное значение. Функция `tuple_elem`: $V \times Z \rightarrow V$ из символьного выражения для кортежа и номера элемента создаёт символьное выражение для результата извлечения из кортежа.

Для анализа свойств будем использовать атрибуты – функции, которые принимают на вход абстрактное состояние и символьное выражение, а возвращают значение некоторого домена. Для анализа свойства, что значение может быть нулём, введём атрибут `nil`: $C \times V \rightarrow B$. Если в абстрактном состоянии C для некоторого ребра `nil`(C, v) = `true`, значит, на этом ребре символьное выражение v имеет только нулевые значения. Отметим, что если `nil`(C, v) = `false`, то это не значит, что символьное выражение не равно нулю, а только то, что анализ не имеет информации, что символьное выражение может иметь только нулевые значения. Для анализа свойств, что указатель не равен нулю, будем использовать атрибут

not_nil:$C \times V \to B$. Введём аналогичные атрибуты для булевых значений: is_true, is_false.

Мы будем использовать следующую нотацию: $C \vdash \texttt{val}(a) = v_a$ означает, что в контексте абстрактного состояния C функция val имеет значение v_a для a, то есть $\texttt{val}(C, a) = v_a$. Запись $C_2 = C_1\ \{v\ [a \to b]\}$ означает, что абстрактное состояние C_2 создано из C_1 с тем отличием, что функция v для аргумента a имеет значение b. Опишем шаги анализа для инструкций, влияющих на ассоциативные массивы:

- Шаг анализа для инструкции `t = lookup2(m, k)`:

$$\frac{\begin{array}{l} C_{\text{in}} \vdash \texttt{val}(m) = v_m, \quad \texttt{val}(k) = v_k \\ C_{\text{out}} = C_{\text{in}}\ \{\texttt{val}\ [t \to \texttt{create\_tuple}(v_m, v_k)]\} \end{array}}{\langle t = \texttt{lookup2}(m, k), \quad C_{\text{in}} \rangle \to C_{\text{out}}}$$

  Передаточная функция из входного состояния C_{in} получает символьные выражения для аргументов инструкции lookup2, на их основе создаёт символьное выражение $\texttt{create\_tuple}(v_m, v_k)$, которое помещается в выходное состояние C_{out} как результат функции val для аргумента переменной t.

- Шаг анализа для инструкции `r = extract(tuple, n)`, где $n \in \mathbb{Z}$:

$$\frac{\begin{array}{l} C_{\text{in}} \vdash \texttt{val(tuple)} = v_t \\ C_{\text{out}} = C_{\text{in}}\ \{\texttt{val}\ [r \to \texttt{tuple\_elem}(v_t, n)]\} \end{array}}{\langle \texttt{r = extract(tuple, n)}, \quad C_{\text{in}} \rangle \to C_{\text{out}}}$$

  Передаточная функция из входного состояния C_{in} получает символьное выражение для кортежа-аргумента инструкции extract и на основе его и ин- декса создаётся символьное выражение для извлечённого из кортежа значения. Оно помещается в выходное состояние C_{out} как результат функции val для аргумента переменной r.

- Шаг анализа для инструкции `assume(not(f))`:

$$\frac{\begin{array}{l} C_{\text{in}} \vdash v_t = \texttt{create\_tuple}(v_m, v_k) \\ C_{\text{in}} \vdash v_r = \texttt{tuple\_elem}(v_t, 0) \\ C_{\text{in}} \vdash v_f = \texttt{tuple\_elem}(v_t, 1) \\ C_{\text{in}} \vdash C_{\text{out}} = C_{\text{in}}\ \{\texttt{nil}\ [v_r \to \texttt{true}], \texttt{is\_false}[v_f \to \texttt{true}]\} \end{array}}{\langle \texttt{assume(not(f))}, \quad C_{\text{in}} \rangle \to C_{\text{out}}}$$

  assume(not(f)) – вспомогательная инструкция, которая сообщает, что управление перешло на ветку, где булева переменная f имеет ложное значение.

  Передаточная функция из входного состояния C_{in} получает символьные выражения для первого элемента кортежа v_r (значения, ассоциированного с ключом в отображении) и второго элемента v_f (булевой переменной). Функция обновляет атрибут nil для переменной v_r и атрибут is_false для переменной v_f.

- `d = deref(res)`. Если известно, что аргумент разыменования имеет нулевое значение, то есть $\texttt{nil}(v_{res}) = \texttt{true}$, где $v_{res} = \texttt{val(res)}$, то Svace выдаст предупреждение о разыменовании нулевого указателя.

Описанный выше анализ позволяет найти ошибку в коде листинга 1, что возможно за счёт отслеживания взаимосвязей между переменными кортежа для результата извлечения из отображения. Таким образом, наш подход заключается в точном моделировании содержимого отображений в тех случаях, когда это возможно. Если, согласно моделированию, указатель имеет нулевое значение, то выдаётся предупреждение об этом.

3.2 Моделирование состояния отображения

В данном разделе опишем, как можно определять отсутствие ключа в отображении, когда при попытке извлечения данного ключа значение будет нулевым. Рассмотрим пример кода, содержащего ошибку разыменования нулевого указателя (листинг 3).

```
1 func foo(str string) {
2     m = make(map[int]*string)
3     m[8] = &str
4     res = m[9]
5     v = *res // null-pointer dereference
6 }
```

Листинг 3. Отсутствие элемента
Listing 3. Element missing

Отображение создается внутри процедуры, и единственный ключ, с которым ассоциировано значение – это `8`. При извлечении значения, ассоциированного с ключом `9`, будет получено нулевое значение `nil`, следовательно, на 4 строке произойдет ошибка разыменования нулевого указателя. Для поиска ошибок потребуется моделирование элементов, содержащихся в отображении.

Данный код транслируется в промежуточное представление, показанное на листинге 4.

```
1 m = makemap()
2 mapUpdate(m, 8, str)
3 res = lookup1(m, 9)
4 v = deref(res)
```

Листинг 4. Промежуточное представление для примера с отсутствием элемента
Listing 4. Svace IR for example with element missing

Для поиска ошибок такого рода смоделируем состояние содержимого ассоциативного массива. Введём атрибут steps: $C \times V \to \{V\}$, который означает множество всех ключей, с которыми ассоциировано значение.

Нам потребуется вспомогательный атрибут `created: C × V → B`, означающий, что отображение было создано внутри анализируемой функции, и, следовательно, анализатор может отследить его содержимое. Наличие атрибута `created` у символьного выражения для отображения играет важную роль: если отображение создано внутри процедуры, то у нас есть информация о всех символьных выражениях для ключей, с которыми ассоциировано значение. В противном случае мы не можем точно смоделировать элементы отображения, поскольку, вообще говоря, мы не знаем состояние отображения в произвольном контексте вызова этой процедуры.

Опишем шаги анализа, которые позволяют смоделировать состояние содержимого отображения:

- Шаг анализа для инструкции `m = makemap()`:

  $$\mathrm{v}_m - \textbf{new}\ (C_{\mathtt{in}} \vdash \nexists v_m : \mathtt{val}(a) = v_m)$$
  $$C_{\mathtt{out}} = C_{\mathtt{in}}\{\mathtt{val}\ [m \to \mathbf{v_m}],\ \mathtt{steps}\ [v_m \to \{\}],\ \mathtt{created}\ [v_m \to \mathit{true}]\}$$
  $$\langle m = \mathtt{makemap()},\ C_{\mathtt{in}}\rangle \to C_{\mathtt{out}}$$

  Передаточная функция создаёт новое символьное выражение $\boldsymbol{v_m}$ для отображения m. Функция обновляет для этого выражения атрибут `steps`, записывая в него пустое множество (отображение создано и ещё не содержит ни одного ключа) и `created` (значение атрибута равно `true`) для выходного состояния $C_{\mathtt{out}}$.

- Шаг анализа для инструкции `mapUpdate(m, key, value)`:

$$\frac{C_{in} \vdash \texttt{val}(\texttt{m}) = v_m, \quad \texttt{val}(\texttt{key}) = v_k \qquad C_{out} = C_{in}\ \{\texttt{steps}\ [v_m \to \texttt{steps}(v_m) \cup \{v_k\}]\}}{\langle \texttt{mapUpdate}(\texttt{m},\ \texttt{key},\ \texttt{value}),\ C_{in}\rangle \to C_{out}}$$

Передаточная функция получает значение атрибута `steps` из входного состояния C_{in} и добавляет к нему выражение v_k. Новое множество записывается в выходное состояние C_{out} как значение атрибута `steps` для v_m. Это означает, что при обращении к отображению по ключу v_k будет возвращено ненулевое значение[4].

- Шаг анализа для инструкции `r = lookup1(m, key)`:

$$\frac{\begin{array}{l} C_{in} \vdash \texttt{val}(m) = v_m, \quad \texttt{val}(\texttt{key}) = v_k \\ C_{in} \vdash \texttt{created}(v_m), \quad \texttt{steps}(v_m) \\ C_{out} = C_{in}\ \{\texttt{val}\ [r \to v_r]\ ,\ \texttt{nil}\ [v_r \to \mathrm{A}]\} \end{array}}{\langle \texttt{res = lookup1(m, key)},\ C_{in}\rangle \to C_{out},}$$

где $\mathrm{A} = (v_k \notin \texttt{steps}(v_m)) \wedge (\texttt{created}(v_m) = \texttt{true})$. Передаточная функция получает из входного состояния C_{in} символьные выражения для отображения v_m и ключа v_k и создаёт новое символьное выражение для извлечённого значения v_r, которое помещается в выходное состояние C_{out}. Если значение запрашиваемого ключа v_k отсутствует в множестве $\texttt{steps}(v_m)$, и отображение создано внутри анализируемой процедуры, т. е. $\texttt{created}(v_m) = \texttt{true}$, то это означает, что запрос вернёт нулевое значение. В таком случае передаточная функция записывает в выходное состояние C_{out} новое значение атрибута `nil` для v_r.

- `d = deref(res)`. Алгоритм из предыдущего раздела, проверяющий значение атрибута `nil`, позволяет найти ошибку.

4. Межпроцедурный поиск ошибок

4.1 Анализ на основе резюме

Для межпроцедурного поиска ошибок используется анализ на основе резюме [11]. При таком подходе после анализа функции создаётся её резюме – краткое описание поведения и побочных эффектов от вызова функции. Резюме используется для анализа инструкций вызова данной функций. В Svace каждая функция анализируется только один раз[5]. Используемый межпроцедурный анализ описан в [12], ниже мы кратко опишем основные особенности используемого анализа.

Анализ состоит из создания резюме по абстрактному состоянию на выходном ребре графа потока управления функции и трансляции резюме в контекст вызова при анализе инструкции вызова функции.

Создание резюме можно рассматривать как шаг анализа, который из абстрактного состояния в точке выхода из процедуры создаёт абстрактное состояние, описывающее интересующие нас свойства: $\langle \texttt{annotate},\ C_{exit}\rangle \to C_{sum}$. В резюме добавляются не все символьные переменные, а только те, которые имеют смысл на стороне вызова: возвращаемые значения, параметры функции, а также символьные выражения, зависящие от них. Для этого анализатор создаёт множество `visible`, в которое добавляются возвращаемые значения функции, входные параметры, глобальные переменные, а также захваченные переменные в замыканиях Go. Затем проводится последовательное добавления зависимых элементов в это

[4] Кроме случаев, при которых само ассоциированное значение является нулевым. В рамках данной работы мы не проводим анализ хранимых значений.

[5] В общем анализ на основе резюме может переанализировать функцию, если это требуется. Например, для анализа рекурсивных функций.

множество. Элемент считается зависимым, если он может быть создан из другого с помощью пути доступа либо разыменования. Само резюме представляет собой специальное абстрактное состояние, которое содержит свойства символьных выражений из множества `visible`. Помещаемые свойства зависят от реализации конкретного детектора, который реализует функцию: `annot`: $V \times C \to C$. Задача этой функции — создать значение в резюме на основе абстрактного состояния в точке выхода из процедуры для символьного выражения.

При трансляции резюме каждому символьному выражению из резюме ставится в соответствие множество элементов в контексте вызова: `trans`: $V \to V^6$. Конкретные детекторы для трансляции отслеживаемых свойств реализуют функцию `apply`: $V \times V \times C \times C \to C$. Эта функция применяется для пары значений

$\langle$`trans`$(v), v\rangle$, соответствующим фактическим и формальным параметрам функции. На основе резюме и абстрактного состояния перед выполнением функции формируется абстрактное состояние после выполнения функции.

Рассмотрим пример ошибки разыменования нулевого указателя при межпроцедурном взаимодействии:

```
1  func get(k int) (int, bool) {
2      v, f := m[k]
3      fmt.Print(v)
4      return v, f
5  }
6
7  func deref() {
8      res, ok = get(key)
9      if !ok {
10         len(*res) // null-pointer dereference
11     }
12 }
```

Листинг 5. Межпроцедурный анализ
Listing 5. Interprocedural analysis

Функция `get` возвращает результат `v` обращения к отображению `map` и флаг `f` успешности извлечения. Если условие на 9 строке в функции `deref` выполнено, то это означает, что в отображении с ключом `key` не ассоциировано никакое значение и переменная `res` равна `nil`. В этом случае на 10 строке возникнет ошибка разыменования нулевого указателя.

При анализе функции `get` анализатор не выявит ошибки, поскольку разыменование извлечённого значения происходит в вызывающей функции. Для обнаружения ошибки необходимо сохранить в резюме информацию о взаимосвязи между извлеченным значением и флагом наличия. Опишем, как это происходит при анализе функции `get`. Введём атрибут `nil_if`: $C \times V \to B$, который означает условие, при котором символьное выражение имеет нулевое значение. Также будем использовать атрибут `ness`: $C \to A$, который означает необходимые условия достижения ребра в графе потоке управления. Значения обоих атрибутов представляют собой булеву формулу, где в качестве переменных используются символьные переменные.

Рассмотрим промежуточное представление функции `get` (табл. 2). Анализ функции проводится аналогично тому, как описано в 3.1. Опишем дополнительные шаги анализа, которые делает анализ для поиска ошибки.

- Шаг анализа для инструкции `f = extract(tuple, 1)`:

[6] Для упрощения будем рассматривать только ситуации, когда в контексте вызова есть не более одного соответствующего элемента. В случае, если таких элементов больше одного, мы не будем транслировать анализируемые здесь свойства.

$$\frac{\begin{array}{l} C_{\texttt{in}} \vdash \texttt{val(tuple)} = v_t \\ C_{\texttt{in}} \vdash v_v = \texttt{tuple\_elem}(v_t, 0) \\ C_{\texttt{in}} \vdash \texttt{ness}(C_{\texttt{in}}) = \mathrm{A} \\ C_{\texttt{out}} = C_{\texttt{in}}\ \{\texttt{val}\ [f \to v_f], \texttt{nil\_if}\ [v_v \to \overline{v_f} \wedge \mathrm{A}]\} \end{array}}{\langle \texttt{f = extract(tuple, 1)},\ \ C_{\texttt{in}} \rangle \to C_{\texttt{out}}}$$

Описанным в 3.1 способом передаточная функция создаёт символьное выражение для v_v и v_f. Функция создаёт новую формулу, в которую добавляется конъюнкция условия, при котором извлечённое значение будет равно `nil` ($\overline{v_f}$) и условие достижимости A. Полученная формула помещается в выходное состояние $C_{\texttt{out}}$ как новое значение атрибута `nil_if` для символьной переменной v_v. Отметим, что в момент анализа этой инструкции Svace только запоминает условие для переменной v_v и не делает предположений об её значении.

- Детектор реализует функцию `annot` следующим образом: добавит в резюме атрибут `nil_if` для переменной v_v, равный $\overline{v_f} \wedge$ A. Формула атрибута может содержать локальные переменные, которые не принадлежат множеству видимых символов `visible`, поэтому перед сохранением в резюме она упрощается: все атомарные формулы, содержащие символьные выражения, отсутствующие в множестве `visible`, заменяются на `false`. Благодаря такому упрощению формула будет описывать условия, при которых значение может быть равно нулю. При этом возможна потеря информации о равенстве нулю, если условие зависит от локальных переменных. То есть описанный детектор не вносит ложных срабатываний, но может пропустить реальные ошибки.

4.3 Трансляция резюме

При трансляции резюме соответствие формальных и фактических аргументов будет описываться функцией `trans` = [$v \to res$, $f \to ok$, $key \to k$, $m \to m$]. Отметим, что глобальная переменная m является сразу и формальным и фактическим аргументом.

Для всех символьных выражений из `trans` будет вызвана функция `apply`. В данном случае интерес представляет только вызов для соответствия $v \to res$.`nil_if`(C_{sum}, v_v) = $\overline{v_f}$. Для трансляции этого свойства также используется функция `trans`, которая заменит все символьные выражения в формуле, и сохранит результат в выходное состояние. Так как `trans`(v_f) = `ok`, то в выходном состоянии будет `nil_if`(C_{sum}, $val(res)$) = $\overline{val(ok)}$.

Табл. 2. Межпроцедурное промежуточное представление
Table 2. Interprocedural analysis IR

Строка исходного кода	Соответствующее представление в IR
func get(k **int**){	func get(k int){
v, f := m[k]	tuple = lookup2(m, k) v = extract(tuple, 0) f = extract(tuple, 1)
fmt.**Print**(v)	call fmt.Print(v)
return v, f	ret = make_tuple(v, f) return ret
}	}

Дальнейший анализ будет выполнен с помощью внутрипроцедурного анализа. Для обнаружения ошибки в данном примере нам потребуется подход с чувствительностью к путям. Для этого мы будем использовать SMT-решатель [13] Z3 [14], который получает на вход формулу P из символьных переменных. Обнаружение ошибки разыменования нулевого

указателя происходит при анализе инструкции `deref(res)`. Формула P, переданная решателю, будет содержать конъюнкцию условия, сохраненного в атрибуте `nil_if` для значения переменной `res`, и значения атрибута `ness` для абстрактного состояния текущей инструкции. Добавление значений атрибута `ness` позволяет отсеять несовместные с ошибкой пути. Если решатель подберёт значения переменных, то будет выдано предупреждение об ошибке.

5. Результаты

Чтобы оценить описанный подход, мы проанализировали[7] нашим инструментом 10 проектов с открытым исходным кодом (табл. 3) с общим количеством строк более 1 миллиона без учета зависимостей и более 7.6 миллионов строк кода с учетом зависимостей[8].

Табл. 3. Используемые открытые проекты
Table 3. Open source projects for evaluation

Проект (https://github.com/*)	LOC	Коммит
tailwarden/komiser	23914	5d9cc2
nanovms/ops	24103	bd7b45
anacrolix/torrent	24764	133cc1
jesseduffield/lazygit	28714	b9a75e
caddyserver/caddy	49929	db9d16
pdfcpu/pdfcpu	53076	4adf70c
XTLS/Xray-core	84377	9a619f
prometheus/prometheus	109681	4e1dac
ovh/cds	208697	884969
pingcap/tidb	555626	f4591b
Суммарно	1162881	

На данных проектах реализованный детектор обнаружил 73 ошибки разыменования нулевого указателя, полученного из отображения. 54.8% этих предупреждений являются истинными.

Ложные предупреждения связаны с недостаточной точностью моделирования условий достижимости (анализатор не смог определить все зависимости между переменными) или нетривиальной логикой программы (итерация по множеству всех ключей без проверки наличия ключа в этом отображении).

```
1  func (h *Handle) handleSingleHistogramUpdate(
2    is infoschema.InfoSchema, rows []chunk.Row) (err error) {
3      ...
4      idx, ok := tbl.Indices[histID]
5      statsVer = idx.StatsVer //error
6      if ok && idx.Histogram.Len() > 0 {
7      ...
8  }
```

Листинг 6: Ошибка разыменования нулевого указателя на TiDB
Listing 6: Nil dereference error on TiDB

Пример найденной ошибки на проекте `pingcap/tidb` [15] показан на листинге 6. Переменная `tbl.Indices` является отображением, в котором значением является структура,

[7] Код проектов анализировался с учетом кода зависимых пакетов.

[8] Импортируемые внешние библиотеки (модули).

имеющая поле `StatsVer`. Значение, ассоциированное с ключом `histID`, извлекается и считывается поле `StatsVer`, при этом проверка на нулевое значение проводится после разыменования.

6. Обзор похожих работ

Анализ коллекций используется в разных видах статического анализа: символьном выполнении, абстрактной интерпретации; не только чтобы находить ошибки при работе с коллекциями, но и повышать точность других анализов и детекторов. Один из подходов к анализу массивов [16] заключается в том, что либо все ячейки массива представлены одним абстрактным значением, либо массив из N элементов представлен N независимыми ячейками. В используемом нами подходе содержимое массивов (как обычных, так и ассоциативных) представлены путями доступа, которые могут быть алиасами друг друга. При этом разрешение алиасов является задачей анализа.

Статический анализатор *Nilaway* [17] позволяет находить ситуации разыменования нулевых указателей в языке Go. Инструмент базируется на существующем стандартном детекторе ошибок *Nilness* [18] стандартного легковесного анализатора `go vet` [19]. Данные инструменты сделаны в рамках инфраструктуры пакетов *golang.org/x/tools* [20]. *Nilaway* использует больше паттернов для поиска ситуации разыменования нулевых указателей, также может находить межпроцедурные ошибки. Мы сравнили наш подход с данным анализатором на основе открытых тестовых наборов *Nilaway*: Svace покрывает все ситуации, предложенные в данном наборе. Также мы сравнили *Nilaway* с нашими синтетическими тестами. В силу отсутствия в *Nilaway* символьного выполнения и SMT-решателя для поиска сложных ошибок, он выдает ложные срабатывания в более сложных случаях, например, на листинг 7 на строке 16 из точки вызова функции `PrintProcessInfo()` на 23 строке. Также *Nilaway* не находит срабатывания с использованием замыканий и инструкций `defer`.

```
1  type Processors struct {
2          processors map[string]*ProcessorInfo
3  }
4
3  func (pr *Processors) GetProcessors(name string) (*ProcessorInfo, bool) {
4          processorInfo, ok := pr.processors[name]
5          return processorInfo, ok
8  }
9
10 type ProcessorInfo struct {
11         threads int
12         Meta    string
13 }
14
15 func (pi *ProcessorInfo) PrintProcessInfo() {
16         fmt.Println(pi.threads, pi.meta)
17 }
18
19 func (pr *Processors) Process() {
20         name := "test"
21
22         if processorInfo, ok := pr.GetProcessors(name); ok {
23                 processorInfo.PrintProcessInfo() //no error
24         }
25 }
```

Листинг 7. Синтетический тест
Listing 7. Synthetic test

В работе [21] описывается способ моделирования библиотечных функций для языка C#, что позволило промоделировать методы `LINQ` и поведение стандартных коллекций языка C#, чтобы описать побочные эффекты функций. Для тестирования алгоритмов, описанных в статье, были созданы модели библиотечных функций для класса `List<T>` (для конструкторов, методов добавления элементов, подсчета количества элементов), а также для некоторых методов операторов запросов `LINQ` (получение первого и последнего элемента – `First/FirstOrDefault`, `Last/LastOrDefault`, `Single/SingleOrDefault` и их перегрузки). Моделирование всего нескольких библиотечных функций позволило избавиться от некоторых ложных срабатываний и получить новые истинные предупреждения, повысив точность анализа на 0,5%. В данном подходе не моделируются значения элементов в коллекции. Написания моделей библиотечных функций недостаточно, чтобы промоделировать более сложное поведение функции, как изменение значений элементов коллекций, без изменений в анализаторе.

7. Заключение

Мы описали подход к анализу ассоциативных массивов в языке Go для использования в статических анализаторах на основе символьного выполнения. Наш анализ может использоваться и для других языков, имеющих ассоциативные массивы.

Моделирование ассоциативных массивов позволяет более точно отслеживать поток данных внутри программы и обнаруживать множество видов ошибок. Также в работе был описан детектор для поиска разыменования нулевых указателей с использованием разработанного анализа.

В дальнейшем мы планируем повышать точность моделирования условий достижимости, улучшать анализатор для обнаружения более сложных видов ошибок, а также расширять используемый подход на анализ ассоциативных массивов в других языках.

Список литературы / References

[1]. A. Belevantsev, A. Borodin, I. Dudina, V. Ignatiev, A. Izbyshev, S. Polyakov, D. Zhurikhin. Design and development of Svace static analyzers. In 2018 Ivannikov Memorial Workshop (IVMEM):3—9, 2018.

[2]. 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.

[3]. A. Borodin, A. Goremykin, S. Vartanov и A. Belevancev. Searching for tainted vulnerabilities in static analysis tool svace. Proceedings of the Institute for System Programming of the RAS, 33(1):7—32, 2021.

[4]. Ssadump: инструмент для вывода и интерпретации формы ssa программ на go. https://pkg.go.dev/golang.org/x/tools@v0.19.0/cmd/ssadump. Дата обращения: 2024-02-01.

[5]. R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman и F. K. Zadeck. An efficient method of computing static single assignment form. В Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages, страницы 25—35, 1989.

[6]. Пакет ssa. https://godoc.org/golang.org/x/tools/go/ssa. Дата обращения: 2024-02-01.

[7]. А. Е. Бородин и И. А. Дудина. Внутрипроцедурный анализ для поиска ошибок на основе символьного выполнения. Труды ИСП РАН, 1:3—4, 2020.

[8]. V. B. Livshits и M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. В Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering, страницы 317—326, 2003.

[9]. R. Ghiya и L. J. Hendren. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. В Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, страницы 1—15, 1996.

[10]. U. P. Khedker, A. Sanyal и A. Karkare. Heap reference analysis using access graphs. ACM Transactions on Programming Languages and Systems (TOPLAS), 30(1):1—es, 2007.

[11]. E. Bodden. The secret sauce in efficient and precise static analysis: the beauty of distributive, summary-based static analyses (and how to master them). В Companion Proceedings for the ISSTA/ECOOP 2018 Workshops, страницы 85—93, 2018.

[12]. А. Е. Бородин. Межпроцедурный контекстно-чувствительный статический анализ для поиска ошибок в исходном коде программ на языках Си и Си++. Дис.канд.физ.-мат.наук, Москва, 2016 г.
[13]. 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.
[14]. L. De Moura и N. Bjørner. Z3: an efficient smt solver, 2008.
[15]. Tidb open-source distributed sql database. https://github.com/pingcap/tidb. Дата обращения::2024-02-10
[16]. F. Alberti, S. Ghilardi и N. Sharygina. Decision procedures for flat array properties. Journal of Automated Reasoning, 54:327—352, 2015.
[17]. Nilaway инструмент статического анализа. https://github.com/uber-go/nilaway. Дата обращения: 2024-01-10.
[18]. Nilness инструмент статического анализа. https://pkg.go.dev/ golang.org/x/tools/go/analysis/passes/nilness. Дата обращения: 2024-01-10.
[19]. Go vet main page. https://golang.org/cmd/vet/. Дата обращения: 2023-10-01.
[20]. Golang.org/x/tools модуль для статического анализа. https://pkg.go.dev/golang.org/x/tools. Дата обращения: 2024-02-10.
[21]. W. G. Biktimirov, V. N. Ignatyev и M. V. Belyaev. Improving the accuracy of library function modeling in the static analyzer. В 2023 Ivannikov Ispras Open Conference (ISPRAS). IEEE.

Информация об авторах / Information about authors

Даниил Николаевич СУББОТИН – сотрудник ИСП РАН, студент магистратуры факультета ВМК МГУ. Сфера научных интересов: компиляторные технологии, статический анализ, анализ языков Go и Python, универсальное абстрактное синтаксическое дерево.

Daniil Nikolaevich SUBBOTIN – ISP RAS researcher, graduate student at the Faculty of Computational Mathematics and Cybernetics of Moscow State University. Research interests: compiler technologies, static analysis, Go and Python languages, universal AST.

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.), researcher. Research interests: static analysis for finding errors in source code.

Варвара Викторовна ДВОРЦОВА – сотрудник ИСП РАН, студентка магистратуры факультета ВМК МГУ. Сфера научных интересов: компиляторные технологии, статический анализ, анализ Golang.

Varvara Viktorovna DVORTSOVA – ISP RAS researcher, graduate student at the Faculty of Computational Mathematics and Cybernetics of Moscow State University. Research interests: compiler technologies, static analysis, Golang analysis.