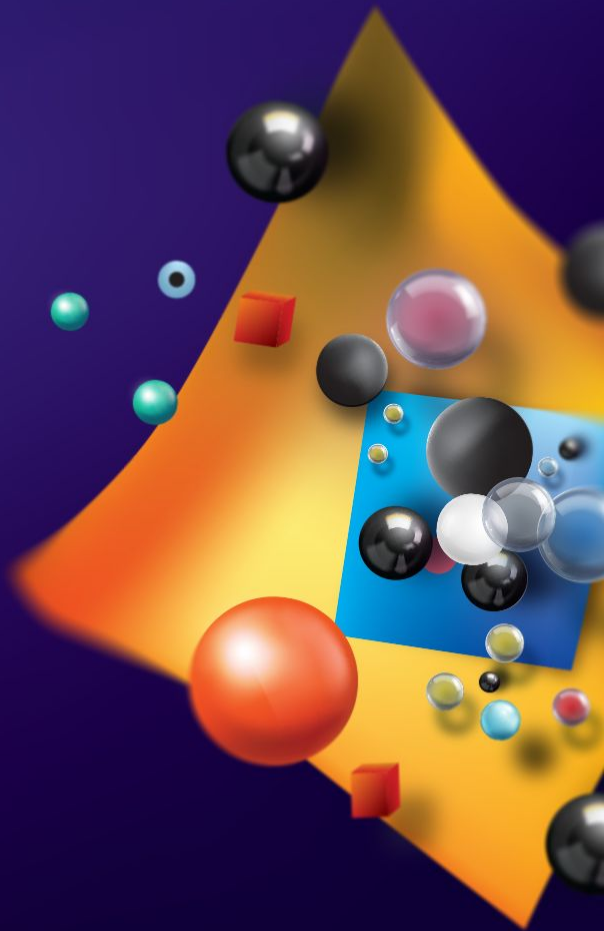
Ruleguard
vs
CodeQL
vs
Semgrep

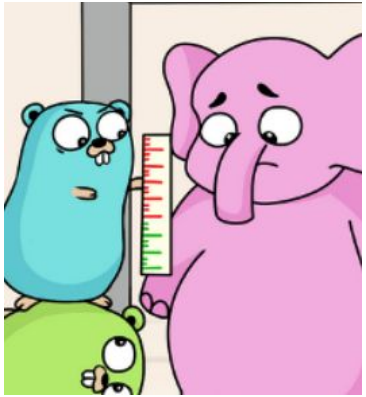
Iskander (pronounced as “Alex”) @quasilyte

Golang
Live 2020



Me & static analysis

go-critic



NoVerify

Ruleguard

Our starting point

We assume that:

- You know that static analysis is cool

Our starting point

We assume that:

- You know that static analysis is cool
- You're using *golangci-lint*

Our starting point

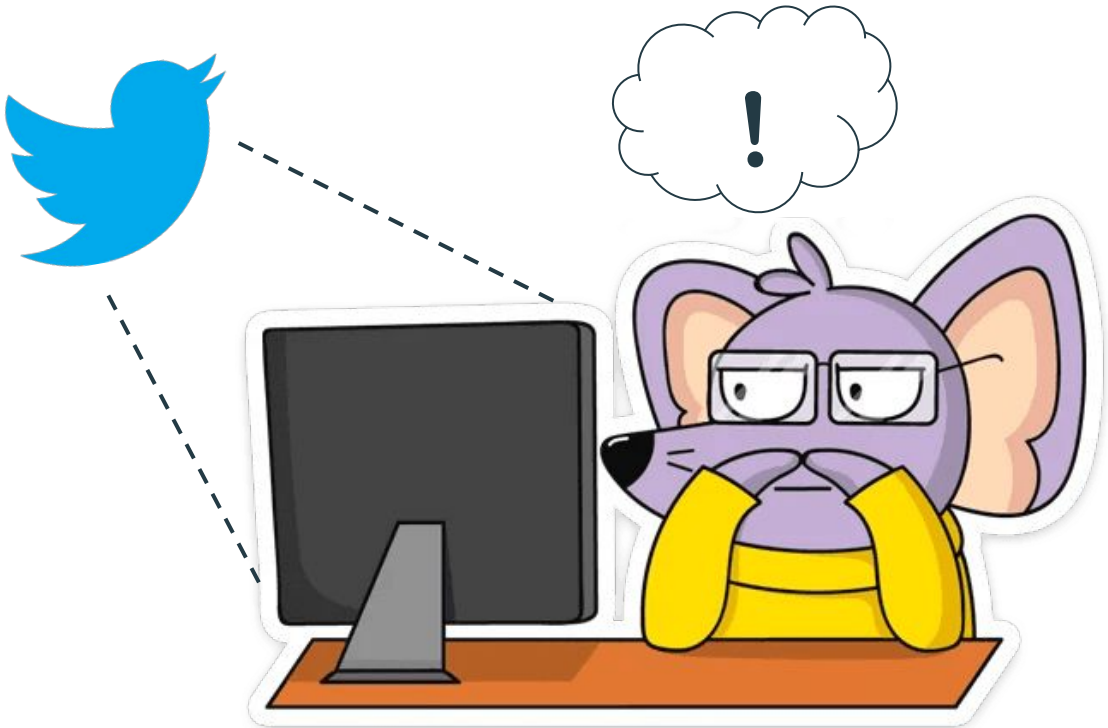
We assume that:

- You know that static analysis is cool
- You're using *golangci-lint*
- You want to create custom code checkers

Trying to come up with linting idea...

/browsing memes/

Golang Live 2020



Golang Live 2020

Damian Gryski
@dgryski

New semgrep check:

Flag this construct:

io.WriteString(w, string(b))

Somewhere on Twitter...

```
  func f(w io.Writer, b []byte) {
-   io.WriteString(w, string(b))
+   w.Write(b)
  }
```

Bad code example

6 hours later...

No results on stackoverflow?!

WDYM AST type types are not “types”?!

How?!

Why?!

6 hours later...

Golang Live 2020

Let’s create our own linter!

We’ll use a fancy *go/analysis* framework

```go
var analyzer = &analysis.Analyzer{
  Name: "writestring",
  Doc:  "find sloppy io.WriteString() usages",
  Run:  run,
}

func run(pass *analysis.Pass) (interface{}, error) {
  // Analyzer implementation...
  return nil, nil
}
```

Analyzer definition

```go
for _, f := range pass.Files {
  ast.Inspect(f, func(n ast.Node) bool {
    // Check n node...
  })
}
```

Analyzer implementation

```go
// 1. Is it a call expression?
call, ok := n.(*ast.CallExpr)
if !ok || len(call.Args) != 2 {
  return true
}
```

Check n node: part 1

```go
// 2. Is it io.WriteString() call?
fn, ok := call.Fun.(*ast.SelectorExpr)
if !ok || fn.Sel.Name != "WriteString" {
  return true
}
pkg, ok := fn.X.(*ast.Ident)
if !ok || pkg.Name != "io" {
  return true
}
```

Check n node: part 2

```go
// 3. Is second arg a string(b) expr?
stringCall, ok := call.Args[1].(*ast.CallExpr)
if !ok || len(stringCall.Args) != 1 {
  return true
}
stringFn, ok := stringCall.Fun.(*ast.Ident)
if !ok || stringFn.Name != "string" {
  return true
}
```

Check n node: part 3

```go
// 4. Does b has a type of []byte?
b := stringCall.Args[0]
if pass.TypesInfo.TypeOf(b).String() != "[]byte" {
  return true
}
```

Check n node: part 4

```go
// 5. Report the issue
msg := "io.WriteString(w, string(b)) -> w.Write(b)"
pass.Reportf(call.Pos(), msg)
```

Check n node: part 5

```go
func main() {
  singlechecker.Main(analyzer)
}
```

Main function definition

It works

But not without problems...

```go
func f(io InputController, b []byte) {
  io.WriteString(w, string(b))
}
```

io could be something else!

```go
func f(io InputController, b []byte) {
  io.WriteString(w, string(b))
}
```

Need to check that io is a package

io could be something else!

```go
import "github.com/quasilyte/io" // not stdlib!

func f(b []byte) {
  io.WriteString(w, string(b))
}
```

But even if it is a package we can get confused

io could be something else!

```
isharipov@lispbook:analysis-test$ go run main.go -c 1 -- test.go
/home/isharipov/CODE/go/analysis-test/test.go:6:2: io.WriteString(w, string(b)) -> w.Write(b)
5       func f(w io.Writer, x []byte, s string) {
6               io.WriteString(w, string(x))
7               io.WriteString(w, s)
```

The warning message is not perfect

Golang Live 2020

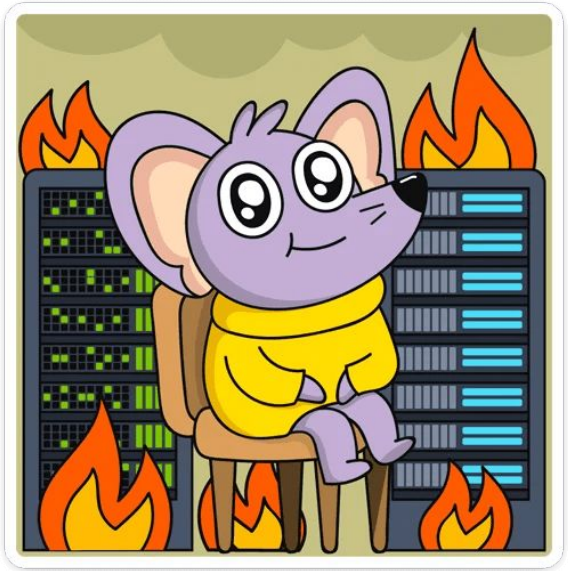
```
isharipov@lispbook:analysis-test$ go run main.go -c 1 -- test.go
/home/isharipov/CODE/go/analysis-test/test.go:6:2: io.WriteString(w, string(b)) -> w.Write(b)
5       func f(w io.Writer, x []byte, s string) {
6               io.WriteString(w, string(x))
7               io.WriteString(w, s)
```

[ ]byte variable is called “x”, not “b”

The warning message is not perfect

Golang Live 2020

It could be worse



Golang Live 2020

Let’s try again

Now with ruleguard

```go
func writeString(m fluent.Matcher) {
  m.Match(`io.WriteString($w, string($b))`).
    Where(m["b"].Type.Is("[]byte")).
    Report("$$ -> $w.Write($b)")
}
```

writeString rule

```
func writeString(m fluent.Matcher) {
  m.Match(`io.WriteString($w, string($b))`).
    Where(m["b"].Type.Is("[]byte")).
    Report("$$ -> $w.Write($b)")
}
```

A rules group named writeString

(May include several rules)

writeString rule

Golang Live 2020

```
func writeString(m fluent.Matcher) {
  m.Match(`io.WriteString($w, string($b))`).
    Where(m["b"].Type.Is("[]byte")).
    Report("$$ -> $w.Write($b)")
}
```

AST pattern

writeString rule

```
func writeString(m fluent.Matcher) {
  m.Match(`io.WriteString($w, string($b))`).
   Where(m["b"].Type.Is("[]byte")).
   Report("$$ -> $w.Write($b)")
}
```

Result filter

writeString rule

Golang Live 2020

```
func writeString(m fluent.Matcher) {
  m.Match(`io.WriteString($w, string($b))`).
    Where(m["b"].Type.Is("[]byte")).
    Report("$$ -> $w.Write($b)")
}
```

Warning message template

writeString rule

Golang Live 2020

```
isharipov@lispbook:ruleguardtest$ ruleguard -c 1 -rules rules.go test.go
/home/isharipov/CODE/go/ruleguardtest/test.go:6:2: io.WriteString(w, string(x)) -> w.Write(x)
5       func f(w io.Writer, x []byte, s string) {
6               io.WriteString(w, string(x))
7               io.WriteString(w, s)
```

The warning message is perfect!

Golang Live 2020

```
func writeString(m fluent.Matcher) {
  m.Match(`io.WriteString($w, string($b))`).
    Where(m["b"].Type.Is("[]byte")).
    Suggest("$w.Write($b)")
}
```

Auto fix replacement template
(can be combined with Report)

writeString rule

```
isharipov@lispbook:ruleguardtest$ cat test.go
package test

import "io"

func f(w io.Writer, x []byte, s string) {
        io.WriteString(w, string(x))
        io.WriteString(w, s)
        io.WriteString(w, string(s))
}
isharipov@lispbook:ruleguardtest$ ruleguard -c 1 -rules rules.go -fix test.go
isharipov@lispbook:ruleguardtest$ cat test.go
package test

import "io"

func f(w io.Writer, x []byte, s string) {
        w.Write(x)
        io.WriteString(w, s)
        io.WriteString(w, string(s))
}
```

With -fix, suggestions are applied automagically

Golang Live 2020

Let’s try semgrep

```yaml
rules:
  - id: writestring
    patterns:
      - pattern: io.WriteString($W, string($B))
    message: "use $W.Write($B)"
    languages: [go]
    severity: ERROR
```

writestring.yml

Golang Live 2020

```
isharipov@lispbook:semgreptest$ semgrep -f writestring.yml test.go
running 1 rules...
test.go
severity:error rule:writestring: use w.Write(x)
6:      io.WriteString(w, string(x))
severity:error rule:writestring: use w.Write(s)
8:      io.WriteString(w, string(s))
ran 1 rules on 1 files: 2 findings
```

Something went wrong...

Golang Live 2020

```
isharipov@lispbook:semgreptest$ semgrep -f writestring.yml test.go
running 1 rules...
test.go
severity:error rule:writestring: use w.Write(x)
6:      io.WriteString(w, string(x))
severity:error rule:writestring: use w.Write(s)
8:      io.WriteString(w, string(s))
ran 1 rules on 1 files: 2 findings
```

False positive!

Something went wrong...

Golang Live 2020

```yaml
rules:
  - id: writestring
    patterns:
      - pattern: io.WriteString($W, string($B))
    message: "use $W.Write($B)"
    languages: [go]
    severity: ERROR
```

TODO: type filters

writestring.yml

Golang Live 2020

By the way…

Have you heard of *YAML5*?

```
{
  rules: [
    {
      id: 'writestring',
      patterns: [
        {pattern: 'io.WriteString($W, string($B))'},
      ],
      message: 'use $W.Write($B)',
      languages: ['go'],
      severity: 'ERROR',
    },
  ],
}
```

Using YAML5 format for semgrep rules

Let’s try CodeQL

QL language reference

Learn all about QL, the powerful query language that underlies the code scanning tool CodeQL.

- About the QL language
- Predicates
- Queries
- Types
- Modules
- Aliases
- Variables
- Expressions
- Formulas
- Annotations
- Recursion
- Lexical syntax
- Name resolution
- Evaluation of QL programs
- QL language specification
- QLDoc comment specification

QL language reference

Learn all about QL, the powerful query language that underlies the code scanning tool CodeQL.

- About the QL language
- Predicates
- Queries
- Types
- Modules
- Aliases
- Variables
- Expressions
- Formulas
- Annotations
- Recursion
- Lexical syntax
- Name resolution
- Evaluation of QL programs
- QL language specification
- QLDoc comment specification

CodeQL CLI

The CodeQL command-line interface (CLI) is used to create databases for security research. You can query CodeQL databases directly from the command line or using the Visual Studio Code extension.

See the following links to learn how to get set up and run CodeQL commands:

- About the CodeQL CLI: Software developers and security researchers can secure their code using the CodeQL CLI.
- Getting started with the CodeQL CLI: Set up the CodeQL CLI so that you can run CodeQL processes from your command line.
- Creating CodeQL databases: Create relational representations of source code that can be queried like any other database.
- Analyzing CodeQL databases: Analyze your code using queries written in a specially-designed, object-oriented query language.
- Upgrading CodeQL databases: Upgrade your databases so that they can be analyzed using the most up to date CodeQL products.
- Using custom queries with the CodeQL CLI: Use custom queries to extend your analysis or highlight errors that are specific to a particular codebase.
- Creating CodeQL query suites: Define query suite definitions for groups of frequently used queries.
- Testing custom queries: Set up regression testing of custom queries to ensure that they behave as expected in your analysis.
- Testing query help files: Test query help files by rendering them as markdown to ensure they are valid before uploading them to the CodeQL repository or using them in code scanning.

For more information about the CLI commands, see the "CodeQL CLI reference."

QL language reference

Learn all about QL, the powerful query language that underlies the code scanning tool CodeQL.

- About the QL language
- Predicates
- Queries
- Types
- Modules
- Aliases
- Variables
- Expressions
- Formulas
- Annotations
- Recursion
- Lexical syntax
- Name resolution
- Evaluation of QL programs
- QL language specification
- QLDoc comment specification

CodeQL queries

CodeQL queries are used in code scanning analyses to find problems in source code, including potential security vulnerabilities.

- About CodeQL queries: CodeQL queries are used to analyze code for issues related to security, correctness, maintainability, and readability.
- Metadata for CodeQL queries: Metadata tells users important information about CodeQL queries. You must include the correct query metadata in a query to be able to view query results in source code.
- Query help files: Query help files tell users the purpose of a query, and recommend how to solve the potential problem the query finds.
- Defining the results of a query: You can control how analysis results are displayed in source code by modifying a query's `select` statement.
- Providing locations in CodeQL queries: CodeQL includes mechanisms for extracting the location of elements in a codebase. Use these mechanisms when writing custom CodeQL queries and libraries to help display information to users.
- About data flow analysis: Data flow analysis is used to compute the possible values that a variable can hold at various points in a program, determining how those values propagate through the program and where they are used.
- Creating path queries: You can create path queries to visualize the flow of information through a codebase.
- Troubleshooting query performance: Improve the performance of your CodeQL queries by following a few simple guidelines.

CodeQL CLI

The CodeQL command-line interface (CLI) is used to create databases for security research. You can query CodeQL databases directly from the command line or using the Visual Studio Code extension.

See the following links to learn how to get set up and run CodeQL commands:

- About the CodeQL CLI: Software developers and security researchers can secure their code using the CodeQL CLI.
- Getting started with the CodeQL CLI: Set up the CodeQL CLI so that you can run CodeQL processes from your command line.
- Creating CodeQL databases: Create relational representations of source code that can be queried like any other database.
- Analyzing CodeQL databases: Analyze your code using queries written in a specially-designed, object-oriented query language.
- Upgrading CodeQL databases: Upgrade your databases so that they can be analyzed using the most up to date CodeQL products.
- Using custom queries with the CodeQL CLI: Use custom queries to extend your analysis or highlight errors that are specific to a particular codebase.
- Creating CodeQL query suites: Define query suite definitions for groups of frequently used queries.
- Testing custom queries: Set up regression testing of custom queries to ensure that they behave as expected in your analysis.
- Testing query help files: Test query help files by rendering them as markdown to ensure they are valid before uploading them to the CodeQL repository or using them in code scanning.

For more information about the CLI commands, see the "CodeQL CLI reference."

QL language reference

Learn all about QL, the powerful query language that underlies the code scanning tool CodeQL.

- About the QL language
- Predicates
- Queries
- Types
- Modules
- Aliases
- Variables
- Expressions
- Formulas
- Annotations
- Recursion
- Lexical syntax
- Name resolution
- Evaluation of QL programs
- QL language specification
- QLDoc comment specification

CodeQL queries

CodeQL queries are used in code scanning analyses to find problems in source code, including potential security vulnerabilities.

- About CodeQL queries: CodeQL queries are used to analyze code for issues related to security, correctness, maintainability, and readability.
- Metadata for CodeQL queries: Metadata tells users important information about CodeQL queries. You must include the correct query metadata in a query to be able to view query results in source code.
- Query help files: Query help files tell users the purpose of a query, and recommend how to solve the potential problem the query finds.
- Defining the results of a query: You can control how analysis results are displayed in source code by modifying a query's `select` statement.
- Providing locations in CodeQL queries: CodeQL includes mechanisms for extracting the location of elements in a codebase. Use these mechanisms when writing custom CodeQL queries and libraries to help display information to users.
- About data flow analysis: Data flow analysis is used to compute the possible values that a variable can hold at various points in a program, determining how those values propagate through the program and where they are used.
- Creating path queries: You can create path queries to visualize the flow of information through a codebase.
- Troubleshooting query performance: Improve the performance of your CodeQL queries by following a few simple guidelines.

```
 *
 * ```go
 * f(x)
 * g(a, b...)
 * ```
 */
class CallExpr extends CallOrConversionExpr {
  CallExpr() {
    exists(Expr callee | callee = getChildExpr(0) | not isTypeExprBottomUp(callee))
    or
    // only calls can have an ellipsis after their last argument
    has_ellipsis(this)
  }
```

```

/** A byte slice type */
class ByteSliceType extends SliceType {
  ByteSliceType() { this.getElementType() instanceof Uint8Type }
}
```

CodeQL CLI

The CodeQL command-line interface (CLI) is used to create databases for security research. You can query CodeQL databases directly from the command line or using the Visual Studio Code extension.

See the following links to learn how to get set up and run CodeQL commands:

- About the CodeQL CLI: Software developers and security researchers can secure their code using the CodeQL CLI.
- Getting started with the CodeQL CLI: Set up the CodeQL CLI so that you can run CodeQL processes from your command line.
- Creating CodeQL databases: Create relational representations of source code that can be queried like any other database.
- Analyzing CodeQL databases: Analyze your code using queries written in a specially-designed, object-oriented query language.
- Upgrading CodeQL databases: Upgrade your databases so that they can be analyzed using the most up to date CodeQL products.
- Using custom queries with the CodeQL CLI: Use custom queries to extend your analysis or highlight errors that are specific to a particular codebase.
- Creating CodeQL query suites: Define query suite definitions for groups of frequently used queries.
- Testing custom queries: Set up regression testing of custom queries to ensure that they behave as expected in your analysis.
- Testing query help files: Test query help files by rendering them as markdown to ensure they are valid before uploading them to the CodeQL repository or using them in code scanning.

For more information about the CLI commands, see the "CodeQL CLI reference."

QL language reference

Learn all about QL, the powerful query language that underlies the code scanning tool CodeQL

- About the QL language
- Predicates
- Queries
- Types
- Modules
- Aliases
- Variables
- Expressions
- Formulas
- Annotations
- Recursion
- Lexical syntax
- Name resolution
- Evaluation of QL programs
- QL language specification
- QLDoc comment specification

CodeQL queries

CodeQL queries are used in code scanning analyses to find probler
potential security vulnerabilities.

- About CodeQL queries: CodeQL queries are used
  correctness, maintainability, and readability.
- Metadata for CodeQL queries: Metadata
  You must include the correct
  code.
- Query help files: Que
  the potential probl
- Defining the re
  modifying a
- Provid

```
byte slice type */
ss ByteSliceType extends SliceType {
yteSliceType() { this.getElementType() instanceof Uint8Type }
```

-line interface (CLI) is used to create databases for security research. You can query
directly from the command line or using the Visual Studio Code extension.

ks to learn how to get set up and run CodeQL commands:

L CLI: Software developers and security researchers can secure their code using the

d with the CodeQL CLI: Set up the CodeQL CLI so that you can run CodeQL processes from
and line.

ng CodeQL databases: Create relational representations of source code that can be queried like any
other database.

- Analyzing CodeQL databases: Analyze your code using queries written in a specially-designed, object-oriented query language.
- Upgrading CodeQL databases: Upgrade your databases so that they can be analyzed using the most up to date CodeQL products.
- Using custom queries with the CodeQL CLI: Use custom queries to extend your analysis or highlight errors that are specific to a particular codebase.
- Creating CodeQL query suites: Define query suite definitions for groups of frequently used queries.
- Testing custom queries: Set up regression testing of custom queries to ensure that they behave as expected in your analysis.
- Testing query help files: Test query help files by rendering them as markdown to ensure they are valid before uploading them to the CodeQL repository or using them in code scanning.

For more information about the CLI commands, see the "CodeQL CLI reference."

```
onversionExpr {
  CallExpr() {
    exists(Expr callee | callee = getChildExpr(0) | not isTypeExprBottomUp(callee))
    or
    // only calls can have an ellipsis after their last argument
    has_ellipsis(this)
  }
```

Golang Live 2020

```
import go
from CallExpr c,
     Expr w,
     ConversionExpr conv,
     SelectorExpr fn
where w = c.getArgument(0)
  and conv = c.getArgument(1)
  and fn = c.getCalleeExpr()
  and fn.getSelector().getName() = "WriteString"
  and fn.getBase().toString() = "io"
  and conv.getOperand().getType() instanceof ByteSliceType
  and conv.getType() instanceof StringType
select c, "use " + w + ".Write(" + conv.getOperand() + ")"
```

CodeQL query

Golang Live 2020

How to run?

- Use the *online query console*
- Select *quasilyte/codeql-test* project
- Copy/paste query from the previous slide

Query ran on 1 project Run 3 minutes ago Order query runs by... Share

quasilyte/codeql-test 3123a59 1 result View as: Alert

c	message
call to WriteString test.go:6	use w.Write(x)

CodeQL pros

- SSA support

CodeQL pros

- SSA support
- Taint analysis (source-sink)

CodeQL pros

- SSA support
- Taint analysis (source-sink)
- Not limited by (Go) syntax rules

CodeQL pros

- SSA support
- Taint analysis (source-sink)
- Not limited by (Go) syntax rules
- Real declarative programming language

CodeQL pros

- SSA support
- Taint analysis (source-sink)
- Not limited by (Go) syntax rules
- Real declarative programming language
- Backed by GitHub

CodeQL pros

- SSA support
- Taint analysis (source-sink)
- Not limited by (Go) syntax rules
- Real declarative programming language
- Backed by ~~GitHub~~ Microsoft

CodeQL pros

- SSA support
- Taint analysis (source-sink)
- Not limited by (Go) syntax rules
- Real declarative programming language
- Backed by ~~GitHub~~ Microsoft
- 1st class GitHub integration

CodeQL

Ruleguard
and
Semgrep

Truth be told...

Golang Live 2020

CodeQL cons

The main points that I want to cover:

1. Steep learning curve
2. Simple things are not simple
3. Non-trivial QL may look alien for many people

Why Ruleguard then?

- Very easy to get started (just “go get” it)

Why Ruleguard then?

- Very easy to get started (just “go get” it)
- Rules are written in pure Go

Why Ruleguard then?

- Very easy to get started (just “go get” it)
- Rules are written in pure Go
- Integrated in *golangci-lint* and *go-critic*

Why Ruleguard then?

- Very easy to get started (just “go get” it)
- Rules are written in pure Go
- Integrated in *golangci-lint* and *go-critic*
- Simple things are simple

Why Ruleguard then?

- Very easy to get started (just “go get” it)
- Rules are written in pure Go
- Integrated in *golangci-lint* and *go-critic*
- Simple things are simple
- Very Go-centric (both pro and con)

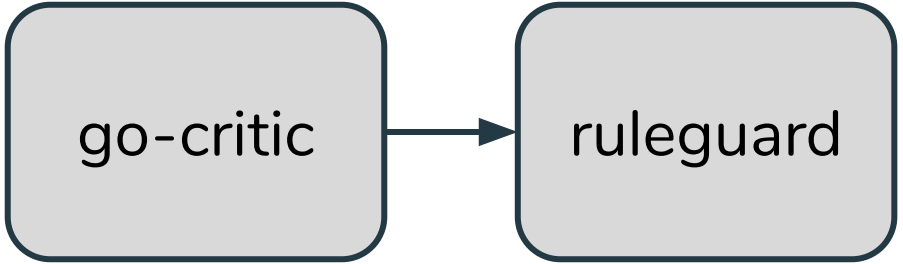
Using ruleguard from golangci

Enabling Ruleguard

1. Install golangci-lint on your pipeline (if not yet)
2. Prepare a rules file (a Go file with ruleguard rules)
3. Enable ruleguard in golangci-lint config

You can also use Ruleguard directly or via go-critic.

ruleguard



```yaml
linters:
  enable:
    - gocritic
linters-settings:
  gocritic:
    enabled-checks:
      - ruleguard
    settings:
      ruleguard:
        rules: "rules.go"
```

.golangci.yml checklist

```yaml
linters:
  enable:
    - gocritic
linters-settings:
  gocritic:
    enabled-checks:
      - ruleguard
    settings:
      ruleguard:
        rules: "rules.go"
```

go-critic linter should be enabled

.golangci.yml checklist

Golang Live 2020

```yaml
linters:
  enable:
    - gocritic
linters-settings:
  gocritic:
    enabled-checks:
      - ruleguard
    settings:
      ruleguard:
        rules: "rules.go"
```

ruleguard checker should be enabled

.golangci.yml checklist

Golang Live 2020

```yaml
linters:
  enable:
    - gocritic
linters-settings:
  gocritic:
    enabled-checks:
      - ruleguard
    settings:
      ruleguard:
        rules: "rules.go"
```

rules param should be set

.golangci.yml checklist

Golang Live 2020

Ruleguard guide

Golang Live 2020

```
m.Match(`pattern1`, `pattern2`)
```

Match() does the syntax matching

```
m.Match(`pattern1`, `pattern2`)
```

Matching alternations:

pattern1|pattern2

Match() does the syntax matching

Golang Live 2020

`$x = $x` pattern string

`$x = $x` pattern string

⇩

Parsed AST

`$x = $x` pattern string

Parsed AST

Modified AST (with meta nodes)

AST matching engine

```
func match(pat, n ast.Node) bool
```

pat is a compiled pattern

n is a node being matched

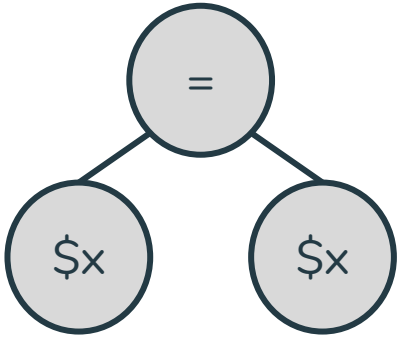
Algorithm

- Both pat and n are traversed
- Non-meta nodes are compared normally
- pat meta nodes are separate cases
- Named matches are collected (capture)
- Some patterns may involve backtracking

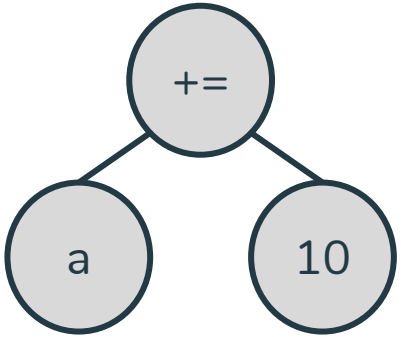
Meta node examples

- $x is a simple “match any” named match
- $_ is a “match any” unnamed match
- $*_ matches zero or more nodes

Pattern $x=$x

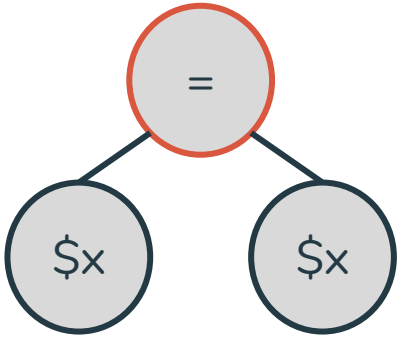


Target a+=10

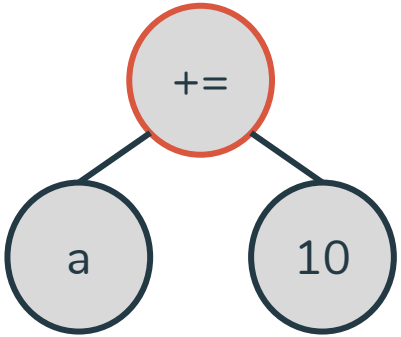


Pattern matching

Pattern $x=$x

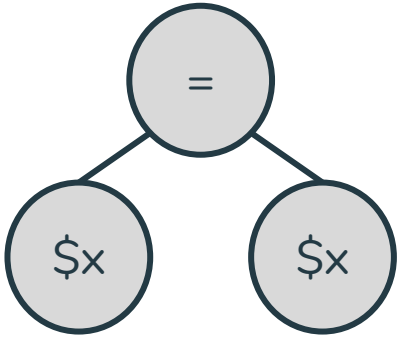


Target a+=10

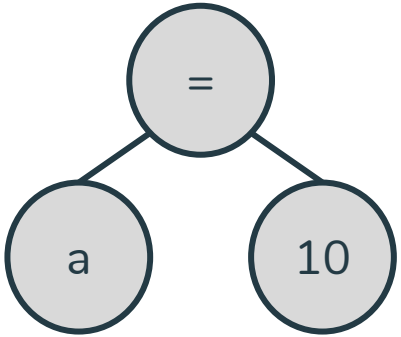


Pattern matching

Pattern $x=$x

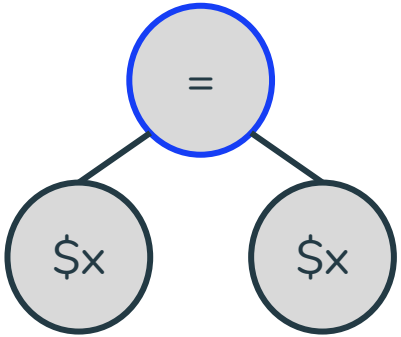


Target a=10

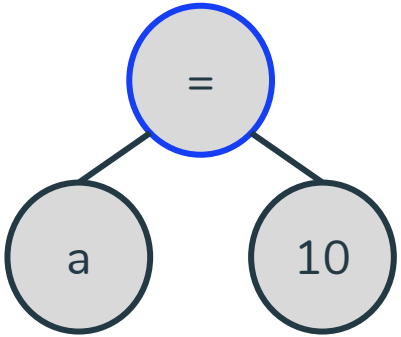


Pattern matching

Pattern $x=$x

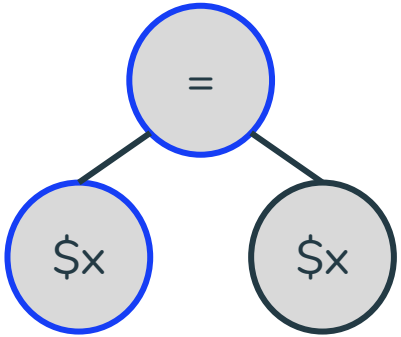


Target a=10

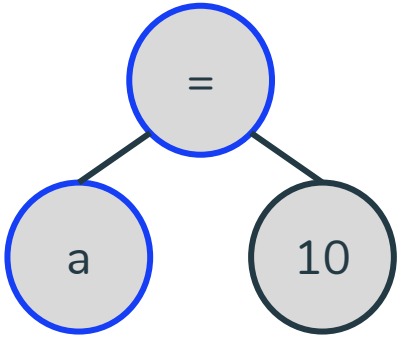


Pattern matching

Pattern $x=$x



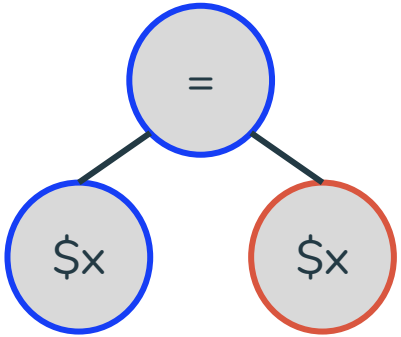
Target a=10



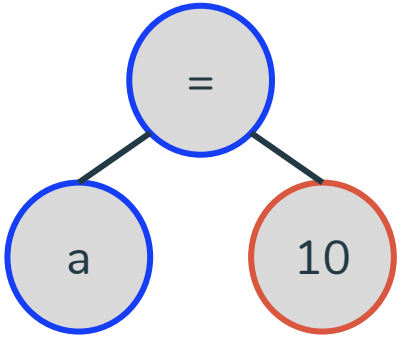
$x is bound to a

Pattern matching

Pattern $x=$x



Target a=10

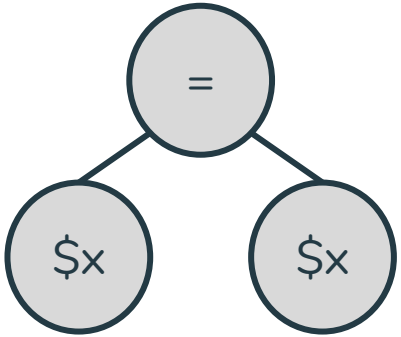


a != 10

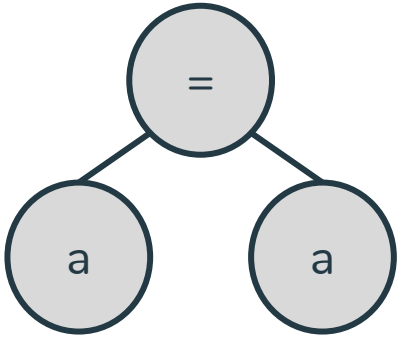
Pattern matching

Golang Live 2020

Pattern $x=$x

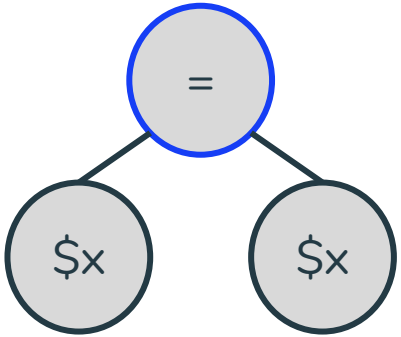


Target a=a

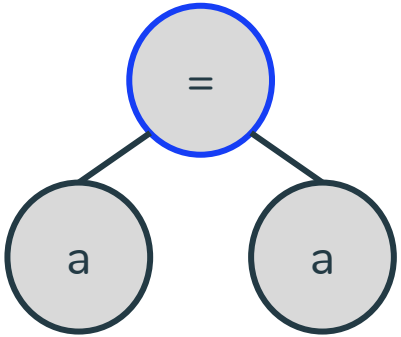


Pattern matching

Pattern $x=$x

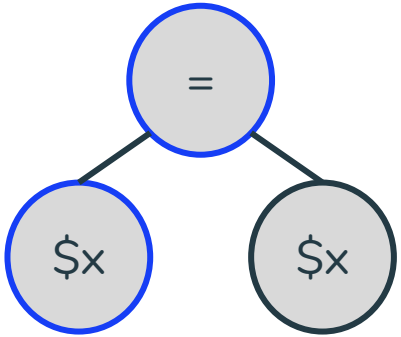


Target a=a

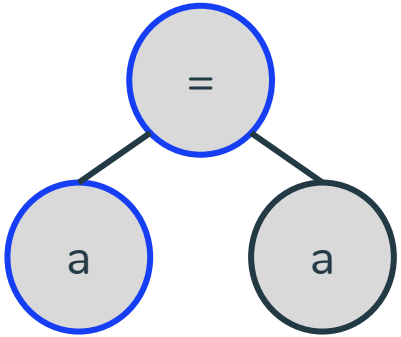


Pattern matching

Pattern $x=$x



Target a=a

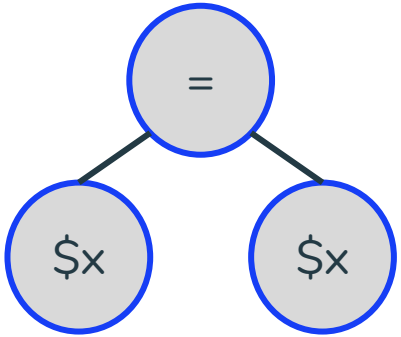


$x is bound to a

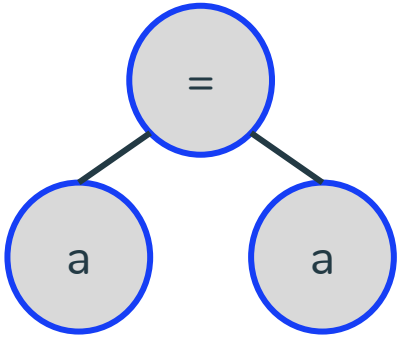
Pattern matching

Golang Live 2020

Pattern $x=$x



Target a=a



a = a, pattern matched

Pattern matching

Golang Live 2020

```
m.Where(cond1 && cond2)
```

Where() is for the match filtering

```
m.Where(cond1 && cond2)
```

Where expression

Where() is for the match filtering

```
m.Where(cond1 && cond2)
```

Where expression operands

Where() is for the match filtering

Golang Live 2020

Where() expression operands

- Matched text predicates

Where() expression operands

- Matched text predicates
- Properties like AssignableTo/ConvertibleTo/Pure

Where() expression operands

- Matched text predicates
- Properties like AssignableTo/ConvertibleTo/Pure
- Check whether a value implements interface

Where() expression operands

- Matched text predicates
- Properties like AssignableTo/ConvertibleTo/Pure
- Check whether a value implements interface
- Type matching expressions

Where() expression operands

- Matched text predicates
- Properties like AssignableTo/ConvertibleTo/Pure
- Check whether a value implements interface
- Type matching expressions
- File-related filters (like “file imports X”)

Type matching examples

`$t`	Arbitrary type
`[]byte`	Byte slice type
`[]$t`	Arbitrary slice type
`map[$t]$t`	Map with $t key and value types
`map[$t]struct{}`	Any set-like map
`func($_) $_`	Any T1->T2 function type

Type matching examples (cont.)

`struct{$*_}`	Arbitrary struct
`struct{$x; $x}`	Struct of 2 $x-typed fields
`struct{$_; $_}`	Struct with any 2 fields
`struct{$x; $*_}`	Struct that starts with $x field
`struct{$*_; $x}`	Struct that ends with $x field
`struct{$*_; $x; $*_}`	Struct that contains $x field

```go
// Just report a message
m.Report("warning message")

// Report + do an auto fix in -fix mode
m.Suggest("autofix template")
```

Report() and Suggest() handle a match

More ruleguard examples

```go
func printFmt(m fluent.Matcher) {
  m.Match(`fmt.Println($s, $*_)`).
    Where(m["s"].Text.Matches(`%[sdv]`)).
    Report("found formatting directives")
}
```

Find formatting directives in a non-formatting fmt calls

```go
func badLock(m fluent.Matcher) {
  m.Match(`$mu.Lock(); $mu.Unlock()`).
    Report(`$mu unlocked immediately`)

  m.Match(`$mu.Lock(); defer $mu.RUnlock()`).
    Report(`maybe $mu.RLock() is intended?`)
}
```

Find mutex usage issues

(*real-world example*)

Golang Live 2020

```go
func sprintErr(m fluent.Matcher) {
  m.Match(`fmt.Sprint($err)`,
          `fmt.Sprintf("%s", $err)`,
          `fmt.Sprintf("%v", $err)`).
    Where(m["err"].Type.Is(`error`)).
    Suggest(`$err.Error()`)
}
```

Suggest error.Error() instead

```
func arrayDeref(m fluent.Matcher) {
  m.Match(`(*$arr)[$i]`).
    Where(m["arr"].Type.Is(`*[$_]$_`)).
    Suggest(`$arr[$i]`)
}
```

Find redundant explicit array dereference expressions

```go
func osFilepath(m fluent.Matcher) {
  m.Match(`os.PathSeparator`).
    Where(m.File().Imports("path/filepath")).
    Suggest(`filepath.Separator`)
}
```

Suggest filepath.Separator instead of os.PathSeparator

```
# -e runs a single inline rule
ruleguard -e 'm.Match(`!($x != $y)`)' file.go
```

Running ruleguard with -e

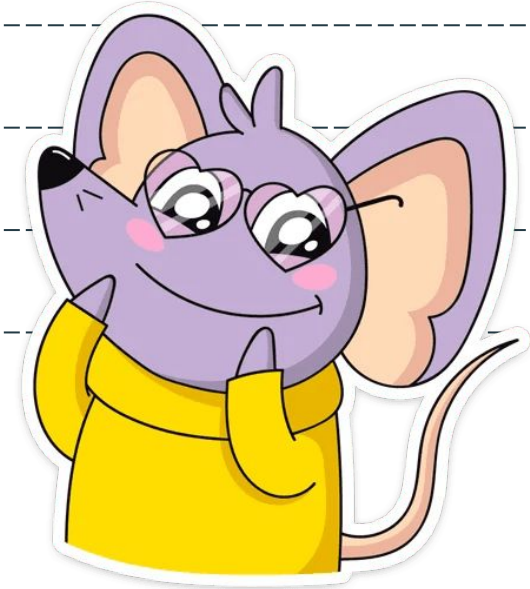
Side-by-side comparison

Written in

go-ruleguard	Go
Semgrep	Mostly OCaml
CodeQL	??? (Compler+Runtime are closed source)

Written in

go-ruleguard	Go
Semgrep	Not Go
CodeQL	Probably not Go



Ruleguard vs Semgrep vs CodeQL

Matching mechanism

go-ruleguard	AST patterns
Semgrep	AST patterns
CodeQL	Dedicated query language

Type matching mechanism

go-ruleguard	Typematch patterns + predicates
Semgrep	N/A (planned, but not implemented yet)
CodeQL	Type assertion-like API

DSL

go-ruleguard	Go
Semgrep	YAML files
CodeQL	Dedicated query language

Supported languages

go-ruleguard	Go
Semgrep	Go + other languages
CodeQL	Go + other languages

How much you can do

go-ruleguard	Simple-medium diagnostics
Semgrep	Simple-medium diagnostics
CodeQL	Almost whatever you want

Links

- Ruleguard quickstart: *EN*, *RU*
- *Ruleguard DSL documentation*
- Ruleguard examples: *one*, *two*
- *gogrep* - AST patterns matching library for Go
- A list of *similar tools*
- *.golangci.yml* from go-critic (uses ruleguard)

Ruleguard
vs
CodeQL
vs
Semgrep

Искандер (Alex) Шарипов @quasilyte

Golang Live 2020