Learning CodeQL

Release 1.24

Jul 27, 2021

CONTENTS

CodeQL is the code analysis platform used by security researchers to automate variant analysis. You can use CodeQL queries to explore code and quickly find variants of security vulnerabilities and bugs. These queries are easy to write and share–visit the topics below and our open source repository on GitHub to learn more. You can also try out CodeQL in the query console on LGTM.com. Here, you can query open source projects directly, without having to download CodeQL databases and libraries.

CodeQL is based on a powerful query language called QL. The following topics help you understand QL in general, as well as how to use it when analyzing code with CodeQL.

> Important
>
> If youve previously used QL, you may notice slight changes in terms we use to describe some important concepts. For more information, see our note about *Recent terminology changes*.

CHAPTER

ONE

QL TUTORIALS

Solve puzzles to learn the basics of QL before you analyze code with CodeQL. The tutorials teach you how to write queries and introduce you to key logic concepts along the way.

1.1 Introduction to QL

Work through some simple exercises and examples to learn about the basics of QL and CodeQL.

1.1.1 Basic syntax

The basic syntax of QL will look familiar to anyone who has used SQL, but it is used somewhat differently.

QL is a logic programming language, so it is built up of logical formulas. QL uses common logical connectives (such as `and`, `or`, and `not`), quantifiers (such as `forall` and `exists`), and other important logical concepts such as predicates.

QL also supports recursion and aggregates. This allows you to write complex recursive queries using simple QL syntax and directly use aggregates such as `count`, `sum`, and `average`.

1.1.2 Running a query

You can try out the following examples and exercises using CodeQL for VS Code, or you can run them in the query console on LGTM.com. Before you can run a query on LGTM.com, you need to select a language and project to query (for these logic examples, any language and project will do).

Once you have selected a language, the query console is populated with the query:

```
import <language>

select "hello world"
```

This query returns the string `"hello world"`.

More complicated queries typically look like this:

```
from /* ... variable declarations ... */
where /* ... logical formulas ... */
select /* ... expressions ... */
```

For example, the result of this query is the number 42:

```
from int x, int y
where x = 6 and y = 7
select x * y
```

Note that `int` specifies that the **type** of `x` and `y` is integer. This means that `x` and `y` are restricted to integer values. Some other common types are: `boolean` (`true` or `false`), `date`, `float`, and `string`.

1.1.3 Simple exercises

You can write simple queries using the some of the basic functions that are available for the `int`, `date`, `float`, `boolean` and `string` types. To apply a function, append it to the argument. For example, `1.toString()` converts the value `1` to a string. Notice that as you start typing a function, a pop-up is displayed making it easy to select the function that you want. Also note that you can apply multiple functions in succession. For example, `100.log().sqrt()` first takes the natural logarithm of 100 and then computes the square root of the result.

Exercise 1

Write a query which returns the length of the string `"lgtm"`. (Hint: here is the list of the functions that can be applied to strings.)

See answer in the query console on LGTM.com

There is often more than one way to define a query. For example, we can also write the above query in the shorter form:

```
select "lgtm".length()
```

Exercise 2

Write a query which returns the sine of the minimum of `3^5` (3 raised to the power 5) and `245.6`.

See answer in the query console on LGTM.com

Exercise 3

Write a query which returns the opposite of the boolean `false`.

See answer in the query console on LGTM.com

Exercise 4

Write a query which computes the number of days between June 10 and September 28, 2017.

See answer in the query console on LGTM.com

1.1.4 Example query with multiple results

The exercises above all show queries with exactly one result, but in fact many queries have multiple results. For example, the following query computes all Pythagorean triples between 1 and 10:

```
from int x, int y, int z
where x in [1..10] and y in [1..10] and z in [1..10] and
      x*x + y*y = z*z
select x, y, z
```

See this in the query console on LGTM.com

To simplify the query, we can introduce a class `SmallInt` representing the integers between 1 and 10. We can also define a predicate `square()` on integers in that class. Defining classes and predicates in this way makes it easy to reuse code without having to repeat it every time.

```
class SmallInt extends int {
  SmallInt() { this in [1..10] }
  int square() { result = this*this }
}

from SmallInt x, SmallInt y, SmallInt z
where x.square() + y.square() = z.square()
select x, y, z
```

See this in the query console on LGTM.com

1.1.5 Example CodeQL queries

The previous examples used the primitive types built in to QL. Although we chose a project to query, we didnt use the information in that projects database. The following example queries *do* use these databases and give you an idea of how to use CodeQL to analyze projects.

Queries using the CodeQL libraries can find errors and uncover variants of important security vulnerabilities in codebases. Visit GitHub Security Lab to read about examples of vulnerabilities that we have recently found in open source projects.

To import the CodeQL library for a specific programming language, type `import <language>` at the start of the query.

```
import python

from Function f
where count(f.getAnArg()) > 7
select f
```

See this in the query console on LGTM.com. The `from` clause defines a variable `f` representing a Python function. The `where` part limits the functions `f` to those with more than 7 arguments. Finally, the `select` clause lists these functions.

```
import javascript

from Comment c
where c.getText().regexpMatch("(?si).*\\bTODO\\b.*")
select c
```

See this in the query console on LGTM.com. The `from` clause defines a variable `c` representing a JavaScript comment. The `where` part limits the comments `c` to those containing the word `"TODO"`. The `select` clause lists these comments.

```
import java

from Parameter p
where not exists(p.getAnAccess())
select p
```

See this in the query console on LGTM.com. The `from` clause defines a variable `p` representing a Java parameter. The `where` clause finds unused parameters by limiting the parameters `p` to those which are not accessed. Finally, the `select` clause lists these parameters.

1.1.6 Further reading

- To find out more about how to write your own queries, try working through the *QL tutorials*.
- For an overview of the other available resources, see *Learning CodeQL*.
- For a more technical description of the underlying language, see the QL language reference.

1.2 Find the thief

Take on the role of a detective to find the thief in this fictional village. You will learn how to use logical connectives, quantifiers, and aggregates in QL along the way.

1.2.1 Introduction

There is a small village hidden away in the mountains. The village is divided into four partsnorth, south, east, and westand in the center stands a dark and mysterious castle Inside the castle, locked away in the highest tower, lies the kings valuable golden crown. One night, a terrible crime is committed. A thief breaks into the tower and steals the crown!

You know that the thief must live in the village, since nobody else knew about the crown. After some expert detective work, you obtain a list of all the people in the village and some of their personal details.

Name	Age	Hair color	Height	Location

Sadly, you still have no idea who could have stolen the crown so you walk around the village to find clues. The villagers act very suspiciously and you are convinced they have information about the thief. They refuse to share their knowledge with you directly, but they reluctantly agree to answer questions. They are still not very talkative and **only answer questions with yes or no**.

You start asking some creative questions and making notes of the answers so you can compare them with your information later:

	Question	Answer
1.	Is the thief taller than 150 cm?	yes
2.	Does the thief have blond hair?	no
3.	Is the thief bald?	no
4.	Is the thief younger than 30?	no
5.	Does the thief live east of the castle?	yes
6.	Does the thief have black or brown hair?	yes
7.	Is the thief taller than 180cm and shorter than 190cm?	no
8.	Is the thief the tallest person in the village?	no
9.	Is the thief shorter than the average villager?	yes
10.	Is the thief the oldest person in the eastern part of the village?	yes

There is too much information to search through by hand, so you decide to use your newly acquired QL skills to help you with your investigation

1. Open the query console on LGTM.com to get started.
2. Select a language and a demo project. For this tutorial, any language and project will do.
3. Delete the default code `import <language> select "hello world"`.

1.2.2 QL libraries

Weve defined a number of QL predicates to help you extract data from your table. A QL predicate is a mini-query that expresses a relation between various pieces of data and describes some of their properties. In this case, the predicates give you information about a person, for example their height or age.

Predicate	Description
`getAge()`	returns the age of the person (in years) as an `int`
`getHairColor()`	returns the hair color of the person as a `string`
`getHeight()`	returns the height of the person (in cm) as a `float`
`getLocation()`	returns the location of the persons home (north, south, east or west) as a `string`

Weve stored these predicates in the QL library `tutorial.qll`. To access this library, type `import tutorial` in

the query console.

Libraries are convenient for storing commonly used predicates. This saves you from defining a predicate every time you need it. Instead you can just `import` the library and use the predicate directly. Once you have imported the library, you can apply any of these predicates to an expression by appending it.

For example, `t.getHeight()` applies `getHeight()` to `t` and returns the height of `t`.

1.2.3 Start the search

The villagers answered yes to the question Is the thief taller than 150cm? To use this information, you can write the following query to list all villagers taller than 150cm. These are all possible suspects.

```
from Person t
where t.getHeight() > 150
select t
```

The first line, `from Person t`, declares that `t` must be a `Person`. We say that the type of `t` is `Person`.

Before you use the rest of your answers in your QL search, here are some more tools and examples to help you write your own QL queries:

1.2.4 Logical connectives

Using logical connectives, you can write more complex queries that combine different pieces of information.

For example, if you know that the thief is older than 30 *and* has brown hair, you can use the following `where` clause to link two predicates:

```
where t.getAge() > 30 and t.getHairColor() = "brown"
```

> Note
>
> The predicate `getHairColor()` returns a `string`, so we need to include quotation marks around the result `"brown"`.

If the thief does *not* live north of the castle, you can use:

```
where not t.getLocation() = "north"
```

If the thief has brown hair *or* black hair, you can use:

```
where t.getHairColor() = "brown" or t.getHairColor() = "black"
```

You can also combine these connectives into longer statements:

```
where t.getAge() > 30
  and (t.getHairColor() = "brown" or t.getHairColor() = "black")
  and not t.getLocation() = "north"
```

> Note
>
> Weve placed parentheses around the `or` clause to make sure that the query is evaluated as intended. Without parentheses, the connective `and` takes precedence over `or`.

Predicates dont always return exactly one value. For example, if a person `p` has black hair which is turning gray, `p.getHairColor()` will return two values: black and gray.

What if the thief is bald? In that case, the thief has no hair, so the `getHairColor()` predicate simply doesnt return any results!

If you know that the thief definitely isnt bald, then there must be a color that matches the thiefs hair color. One way to express this in QL is to introduce a new variable `c` of type `string` and select those `t` where `t.getHairColor()` matches a value of `c`.

```
from Person t, string c
where t.getHairColor() = c
select t
```

Notice that we have only temporarily introduced the variable `c` and we didnt need it at all in the `select` clause. In this case, it is better to use `exists`:

```
from Person t
where exists(string c | t.getHairColor() = c)
select t
```

`exists` introduces a temporary variable `c` of type `string` and holds only if there is at least one `string c` that satisfies `t.getHairColor() = c`.

> Note
>
> If you are familiar with logic, you may notice that `exists` in QL corresponds to the existential quantifier in logic. QL also has a universal quantifier `forall(vars | formula 1 | formula 2)` which is logically equivalent to `not exists(vars | formula 1 | not formula 2)`.

1.2.5 The real investigation

You are now ready to track down the thief! Using the examples above, write a query to find the people who satisfy the answers to the first eight questions:

	Question	Answer
1	Is the thief taller than 150 cm?	yes
2	Does the thief have blond hair?	no
3	Is the thief bald?	no
4	Is the thief younger than 30?	no
5	Does the thief live east of the castle?	yes
6	Does the thief have black or brown hair?	yes
7	Is the thief taller than 180cm and shorter than 190cm?	no
8	Is the thief the oldest person in the village?	no

Hints

1. Dont forget to `import tutorial`!
2. Translate each question into QL separately. Look at the examples above if you get stuck.
3. For question 3, remember that a bald person does not have a hair color.

4. For question 8, note that if a person is *not* the oldest, then there is at least one person who is older than them.
5. Combine the conditions using logical connectives to get a query of the form:

```
import tutorial

from Person t
where <condition 1> and
  not <condition 2> and
  ...
select t
```

Once you have finished, you will have a list of possible suspects. One of those people must be the thief!

See the answer in the query console on LGTM.com

> Note
>
> In the answer, we used `/*` and `*/` to label the different parts of the query. Any text surrounded by `/*` and `*/` is not evaluated as part of the QL code, but is just a *comment*.

You are getting closer to solving the mystery! Unfortunately, you still have quite a long list of suspects To find out which of your suspects is the thief, you must gather more information and refine your query in the next step.

1.2.6 More advanced queries

What if you want to find the oldest, youngest, tallest, or shortest person in the village? As mentioned in the previous topic, you can do this using `exists`. However, there is also a more efficient way to do this in QL using functions like `max` and `min`. These are examples of aggregates.

In general, an aggregate is a function that performs an operation on multiple pieces of data and returns a single value as its output. Common aggregates are `count`, `max`, `min`, `avg` (average) and `sum`. The general way to use an aggregate is:

```
<aggregate>(<variable declarations> | <logical formula> | <expression>)
```

For example, you can use the `max` aggregate to find the age of the oldest person in the village:

```
max(int i | exists(Person p | p.getAge() = i) | i)
```

This aggregate considers all integers `i`, limits `i` to values that match the ages of people in the village, and then returns the largest matching integer.

But how can you use this in an actual query?

If the thief is the oldest person in the village, then you know that the thiefs age is equal to the maximum age of the villagers:

```
from Person t
where t.getAge() = max(int i | exists(Person p | p.getAge() = i) | i)
select t
```

This general aggregate syntax is quite long and inconvenient. In most cases, you can omit certain parts of the aggregate. A particularly helpful QL feature is *ordered aggregation*. This allows you to order the expression using `order by`.

For example, selecting the oldest villager becomes much simpler if you use an ordered aggregate.

```
select max(Person p | | p order by p.getAge())
```

The ordered aggregate considers every person `p` and selects the person with the maximum age. In this case, there are no restrictions on what people to consider, so the `<logical formula>` clause is empty. Note that if there are several people with the same maximum age, the query lists all of them.

Here are some more examples of aggregates:

Example	Result
`min(Person p | p.getLocation() = "east" | p order by p.getHeight())`	shortest person in the east of the village
`count(Person p | p.getLocation() = "south" | p)`	number of people in the south of the village
`avg(Person p | | p.getHeight())`	average height of the villagers
`sum(Person p | p.getHairColor() = "brown" | p.getAge())`	combined age of all the villagers with brown hair

1.2.7 Capture the culprit

You can now translate the remaining questions into QL:

	Question	Answer
9	Is the thief the tallest person in the village?	no
10	Is the thief shorter than the average villager?	yes
11	Is the thief the oldest person in the eastern part of the village?	yes

Have you found the thief?

See the answer in the query console on LGTM.com

1.2.8 Further reading

- QL language reference
- CodeQL tools

1.3 Catch the fire starter

LearnăaboutăQLăpredicatesăandăclassesătoăsolveăyouräsecondămysteryăasăaăQLădetective.

Just as youve successfully found the thief and returned the golden crown to the castle, another terrible crime is committed. Early in the morning, a few people start a fire in a field in the north of the village and destroy all the crops!

You now have the reputation of being an expert QL detective, so you are once again asked to find the culprits.

This time, you have some additional information. There is a strong rivalry between the north and south of the village and you know that the criminals live in the south.

Read the examples below to learn how to define predicates and classes in QL. These make the logic of your queries easier to understand and will help simplify your detective work.

1.3.1 Select the southerners

This time you only need to consider a specific group of villagers, namely those living in the south of the village. Instead of writing `getLocation() = "south"` in all your queries, you could define a new predicate `isSouthern`:

```
predicate isSouthern(Person p) {
  p.getLocation() = "south"
}
```

The predicate `isSouthern(p)` takes a single parameter `p` and checks if `p` satisfies the property `p.getLocation() = "south"`.

Note

- The name of a predicate always starts with a lowercase letter.
- You can also define predicates with a result. In that case, the keyword `predicate` is replaced with the type of the result. This is like introducing a new argument, the special variable `result`. For example, `int getAge() { result = ... }` returns an `int`.

You can now list all southerners using:

```
/* define predicate `isSouthern` as above */

from Person p
where isSouthern(p)
select p
```

This is already a nice way to simplify the logic, but we could be more efficient. Currently, the query looks at every `Person p`, and then restricts to those who satisfy `isSouthern(p)`. Instead, we could define a new class `Southerner` containing precisely the people we want to consider.

```
class Southerner extends Person {
  Southerner() { isSouthern(this) }
}
```

A class in QL represents a logical property: when a value satisfies that property, it is a member of the class. This means that a value can be in many classesbeing in a particular class doesnt stop it from being in other classes too.

The expression `isSouthern(this)` defines the logical property represented by the class, called its *characteristic predicate*. It uses a special variable `this` and indicates that a `Person this` is a `Southerner` if the property `isSouthern(this)` holds.

Note

If you are familiar with object-oriented programming languages, you might be tempted to think of the characteristic predicate as a *constructor*. However, this is **not** the caseit is a logical property which

does not create any objects.

You always need to define a class in QL in terms of an existing (larger) class. In our example, a `Southerner` is a special kind of `Person`, so we say that `Southerner` *extends* (is a subset of) `Person`.

Using this class you can now list all people living in the south simply as:

```
from Southerner s
select s
```

You may have noticed that some predicates are appended, for example `p.getAge()`, while others are not, for example `isSouthern(p)`. This is because `getAge()` is a member predicate, that is, a predicate that only applies to members of a class. You define such a member predicate inside a class. In this case, `getAge()` is defined inside the class `Person`. In contrast, `isSouthern` is defined separately and is not inside any classes. Member predicates are especially useful because you can chain them together easily. For example, `p.getAge().sqrt()` first gets the age of `p` and then calculates the square root of that number.

1.3.2 Travel restrictions

Another factor you want to consider is the travel restrictions imposed following the theft of the crown. Originally there were no restrictions on where villagers could travel within the village. Consequently the predicate `isAllowedIn(string region)` held for any person and any region. The following query lists all villagers, since they could all travel to the north:

```
from Person p
where p.isAllowedIn("north")
select p
```

However, after the recent theft, the villagers have become more anxious of criminals lurking around the village and they no longer allow children under the age of 10 to travel out of their home region.

This means that `isAllowedIn(string region)` no longer holds for all people and all regions, so you should temporarily *override* the original predicate if `p` is a child.

Start by defining a class `Child` containing all villagers under 10 years old. Then you can redefine `isAllowedIn(string region)` as a member predicate of `Child` to allow children only to move within their own region. This is expressed by `region = this.getLocation()`.

```
class Child extends Person {
  /* the characteristic predicate */
  Child() { this.getAge() < 10 }

  /* a member predicate */
  override predicate isAllowedIn(string region) {
    region = this.getLocation()
  }
}
```

Now try applying `isAllowedIn(string region)` to a person `p`. If `p` is not a child, the original definition is used, but if `p` is a child, the new predicate definition overrides the original.

You know that the fire starters live in the south *and* that they must have been able to travel to the north. Write a query to find the possible suspects. You could also extend the `select` clause to list the age of the suspects. That

way you can clearly see that all the children have been excluded from the list.

See the answer in the query console on LGTM.com

You can now continue to gather more clues and find out which of your suspects started the fire

1.3.3 Identify the bald bandits

You ask the northerners if they have any more information about the fire starters. Luckily, you have a witness! The farmer living next to the field saw two people run away just after the fire started. He only saw the tops of their heads, and noticed that they were both bald.

This is a very helpful clue. Remember that you wrote a QL query to select all bald people:

```
from Person p
where not exists (string c | p.getHairColor() = c)
select p
```

To avoid having to type `not exists (string c | p.getHairColor() = c)` every time you want to select a bald person, you can instead define another new predicate `isBald`.

```
predicate isBald(Person p) {
  not exists (string c | p.getHairColor() = c)
}
```

The property `isBald(p)` holds whenever `p` is bald, so you can replace the previous query with:

```
from Person p
where isBald(p)
select p
```

The predicate `isBald` is defined to take a `Person`, so it can also take a `Southerner`, as `Southerner` is a subtype of `Person`. It cant take an `int` for examplethat would cause an error.

You can now write a query to select the bald southerners who are allowed into the north.

See the answer in the query console on LGTM.com

You have found the two fire starters! They are arrested and the villagers are once again impressed with your work.

1.3.4 Further reading

- QL language reference
- CodeQL tools

1.4 Crown the rightful heir

This is a QL detective puzzle that shows you how to use recursion in QL to write more complex queries.

1.4.1 King Basils heir

Phew! No more crimes in the villageyou can finally leave the village and go home.

But then During your last night in the village, the old kingthe great King Basildies in his sleep and there is chaos everywhere!

The king never married and he had no children, so nobody knows who should inherit the kings castle and fortune. Immediately, lots of villagers claim that they are somehow descended from the kings family and that they are the true heir. People argue and fight and the situation seems hopeless.

Eventually you decide to stay in the village to resolve the argument and find the true heir to the throne.

You want to find out if anyone in the village is actually related to the king. This seems like a difficult task at first, but you start work confidently. You know the villagers quite well by now, and you have a list of all the parents in the village and their children.

To find out more about the king and his family, you get access to the castle and find some old family trees. You also include these relations in your database to see if anyone in the kings family is still alive.

The following predicate is useful to help you access the data:

Predicate	Description
`parentOf(Person p)`	returns a parent of `p`

For example, you can list all children `p` together with their parents:

```
from Person p
select parentOf(p) + " is a parent of " + p
```

There is too much information to search through by hand, so you write a QL query to help you find the kings heir.

We know that the king has no children himself, but perhaps he has siblings. Write a query to find out:

```
from Person p
where parentOf(p) = parentOf("King Basil") and
  not p = "King Basil"
select p
```

He does indeed have siblings! But you need to check if any of them are alive Here is one more predicate you might need:

Predicate	Description
`isDeceased()`	holds if the person is deceased

Use this predicate to see if the any of the kings siblings are alive.

```
from Person p
where parentOf(p) = parentOf("King Basil") and
  not p = "King Basil"
  and not p.isDeceased()
select p
```

Unfortunately, none of King Basils siblings are alive. Time to investigate further. It might be helpful to define a predicate `childOf()` which returns a child of the person. To do this, the `parentOf()` predicate can be used inside the definition of `childOf()`. Remember that someone is a child of `p` if and only if `p` is their parent:

```
Person childOf(Person p) {
  p = parentOf(result)
}
```

> Note
>
> As illustrated by the example above, you dont have to directly write `result = <expression involving p>` in the predicate definition. Instead you can also express the relation between `p` and `result` backwards by writing `p` in terms of `result`.

Try to write a query to find out if any of the kings siblings have children:

```
from Person p
where parentOf(p) = parentOf("King Basil") and
  not p = "King Basil"
select childOf(p)
```

The query returns no results, so they have no children. But perhaps King Basil has a cousin who is alive or has children, or a second cousin, or

This is getting complicated. Ideally, you want to define a predicate `relativeOf(Person p)` that lists all the relatives of `p`.

How could you do that?

It helps to think of a precise definition of *relative*. A possible definition is that two people are related if they have a common ancestor.

You can introduce a predicate `ancestorOf(Person p)` that lists all ancestors of `p`. An ancestor of `p` is just a parent of `p`, or a parent of a parent of `p`, or a parent of a parent of a parent of `p`, and so on. Unfortunately, this leads to an endless list of parents. You cant write an infinite QL query, so there must be an easier approach.

Aha, you have an idea! You can say that an ancestor is either a parent, or a parent of someone you already know to be an ancestor.

You can translate this into QL as follows:

```
Person ancestorOf(Person p) {
  result = parentOf(p) or
  result = parentOf(ancestorOf(p))
}
```

As you can see, you have used the predicate `ancestorOf()` inside its own definition. This is an example of recursion.

This kind of recursion, where the same operation (in this case `parentOf()`) is applied multiple times, is very common in QL, and is known as the *transitive closure* of the operation. There are two special symbols `+` and `*` that are extremely useful when working with transitive closures:

- `parentOf+(p)` applies the `parentOf()` predicate to `p` one or more times. This is equivalent to `ancestorOf(p)`.

- `parentOf*(p)` applies the `parentOf()` predicate to `p` zero or more times, so it returns an ancestor of `p` or `p` itself.

Try using this new notation to define a predicate `relativeOf()` and use it to list all living relatives of the king.

Hint:

Here is one way to define `relativeOf()`:

```
Person relativeOf(Person p) {
  parentOf*(result) = parentOf*(p)
}
```

Dont forget to use the predicate `isDeceased()` to find relatives that are still alive.

See the answer in the query console on LGTM.com

1.4.2 Select the true heir

At the next village meeting, you announce that there are two living relatives.

To decide who should inherit the kings fortune, the villagers carefully read through the village constitution:

The heir to the throne is the closest living relative of the king. Any person with a criminal record will not be considered. If there are multiple candidates, the oldest person is the heir.

As your final challenge, define a predicate `hasCriminalRecord` so that `hasCriminalRecord(p)` holds if `p` is any of the criminals you unmasked earlier (in the *Find the thief* and *Catch the fire starter* tutorials).

See the answer in the query console on LGTM.com

1.4.3 Experimental explorations

Congratulations! You have found the heir to the throne and restored peace to the village. However, you dont have to leave the villagers just yet. There are still a couple more questions about the village constitution that you could answer for the villagers, by writing QL queries:

- Which villager is next in line to the throne? Could you write a predicate to determine how closely related the remaining villagers are to the new monarch?
- How would you select the oldest candidate using a QL query, if multiple villagers have the same relationship to the monarch?

You could also try writing more of your own QL queries to find interesting facts about the villagers. You are free to investigate whatever you like, but here are some suggestions:

- What is the most common hair color in the village? And in each region?
- Which villager has the most children? Who has the most descendants?
- How many people live in each region of the village?
- Do all villagers live in the same region of the village as their parents?
- Find out whether there are any time travelers in the village! (Hint: Look for impossible family relations.)

1.4.4 Further reading

- QL language reference
- CodeQL tools

1.5 Cross the river

Use common QL features to write a query that finds a solution to the River crossing logic puzzle.

1.5.1 Introduction

> River crossing puzzle
>
> A man is trying to ferry a goat, a cabbage, and a wolf across a river. His boat can only take himself and at most one item as cargo. His problem is that if the goat is left alone with the cabbage, it will eat it. And if the wolf is left alone with the goat, it will eat it. How does he get everything across the river?

A solution should be a set of instructions for how to ferry the items, such as First ferry the goat across the river, and come back with nothing. Then ferry the cabbage across, and come back with

There are lots of ways to approach this problem and implement it in QL. Before you start, make sure that you are familiar with how to define classes and predicates in QL. The following walkthrough is just one of many possible implementations, so have a go at writing your own query too! To find more example queries, see the list *below*.

1.5.2 Walkthrough

Model the elements of the puzzle

The basic components of the puzzle are the cargo items and the shores on either side of the river. Start by modeling these as classes.

First, define a class `Cargo` containing the different cargo items. Note that the man can also travel on his own, so it helps to explicitly include `"Nothing"` as a piece of cargo.

Show/hide code

```
/** A possible cargo item. */
class Cargo extends string {
  Cargo() {
    this = "Nothing" or
    this = "Goat" or
    this = "Cabbage" or
    this = "Wolf"
  }
}
```

Second, any item can be on one of two shores. Lets call these the left shore and the right shore. Define a class `Shore` containing `"Left"` and `"Right"`.

It would be helpful to express the other shore to model moving from one side of the river to the other. You can do this by defining a member predicate `other` in the class `Shore` such that `"Left".other()` returns `"Right"` and vice versa.

Show/hide code

```
/** One of two shores. */
class Shore extends string {
  Shore() {
    this = "Left" or
    this = "Right"
  }

  /** Returns the other shore. */
  Shore other() {
    this = "Left" and result = "Right"
    or
    this = "Right" and result = "Left"
  }
}
```

We also want a way to keep track of where the man, the goat, the cabbage, and the wolf are at any point. We can call this combined information the state. Define a class `State` that encodes the location of each piece of cargo. For example, if the man is on the left shore, the goat on the right shore, and the cabbage and wolf on the left shore, the state should be `Left, Right, Left, Left`.

You may find it helpful to introduce some variables that refer to the shore on which the man and the cargo items are. These temporary variables in the body of a class are called fields.

Show/hide code

```
/** A record of where everything is. */
class State extends string {
  Shore manShore;
  Shore goatShore;
  Shore cabbageShore;
  Shore wolfShore;

  State() { this = manShore + "," + goatShore + "," + cabbageShore + "," + wolfShore }
}
```

We are interested in two particular states, namely the initial state and the goal state, which we have to achieve to solve the puzzle. Assuming that all items start on the left shore and end up on the right shore, define `InitialState` and `GoalState` as subclasses of `State`.

Show/hide code

```
/** The initial state, where everything is on the left shore. */
class InitialState extends State {
  InitialState() { this = "Left" + "," + "Left" + "," + "Left" + "," + "Left" }
}

/** The goal state, where everything is on the right shore. */
class GoalState extends State {
  GoalState() { this = "Right" + "," + "Right" + "," + "Right" + "," + "Right" }
}
```

Note

> To avoid typing out the lengthy string concatenations, you could introduce a helper predicate `renderState` that renders the state in the required form.

Using the above note, the QL code so far looks like this:

Show/hide code

```
/** A possible cargo item. */
class Cargo extends string {
  Cargo() {
    this = "Nothing" or
    this = "Goat" or
    this = "Cabbage" or
    this = "Wolf"
  }
}

/** One of two shores. */
class Shore extends string {
  Shore() {
    this = "Left" or
    this = "Right"
  }

  /** Returns the other shore. */
  Shore other() {
    this = "Left" and result = "Right"
    or
    this = "Right" and result = "Left"
  }
}

/** Renders the state as a string. */
string renderState(Shore manShore, Shore goatShore, Shore cabbageShore, Shore wolfShore) {
  result = manShore + "," + goatShore + "," + cabbageShore + "," + wolfShore
}

/** A record of where everything is. */
class State extends string {
  Shore manShore;
  Shore goatShore;
  Shore cabbageShore;
  Shore wolfShore;

  State() { this = renderState(manShore, goatShore, cabbageShore, wolfShore) }
}

/** The initial state, where everything is on the left shore. */
class InitialState extends State {
  InitialState() { this = renderState("Left", "Left", "Left", "Left") }
}

/** The goal state, where everything is on the right shore. */
```

(continues on next page)

(continued from previous page)

```
class GoalState extends State {
  GoalState() { this = renderState("Right", "Right", "Right", "Right") }
}
```

Model the action of ferrying

The basic act of ferrying moves the man and one cargo item to the other shore, resulting in a new state.

Write a member predicate (of `State`) called `ferry`, that specifies what happens to the state after ferrying a particular cargo. (Hint: Use the predicate `other`.)

Show/hide code

```
  /** Returns the state that is reached after ferrying a particular cargo item. */
  State ferry(Cargo cargo) {
    cargo = "Nothing" and
    result = renderState(manShore.other(), goatShore, cabbageShore, wolfShore)
    or
    cargo = "Goat" and
    result = renderState(manShore.other(), goatShore.other(), cabbageShore, wolfShore)
    or
    cargo = "Cabbage" and
    result = renderState(manShore.other(), goatShore, cabbageShore.other(), wolfShore)
    or
    cargo = "Wolf" and
    result = renderState(manShore.other(), goatShore, cabbageShore, wolfShore.other())
  }
```

Of course, not all ferrying actions are possible. Add some extra conditions to describe when a ferrying action is safe. That is, it doesnt lead to a state where the goat or the cabbage get eaten. For example, follow these steps:

1. Define a predicate `isSafe` that holds when the state itself is safe. Use this to encode the conditions for when nothing gets eaten.
2. Define a predicate `safeFerry` that restricts `ferry` to only include safe ferrying actions.

Show/hide code

```
  /**
   * Holds if the state is safe. This occurs when neither the goat nor the cabbage
   * can get eaten.
   */
  predicate isSafe() {
    // The goat can't eat the cabbage.
    (goatShore != cabbageShore or goatShore = manShore) and
    // The wolf can't eat the goat.
    (wolfShore != goatShore or wolfShore = manShore)
  }

  /** Returns the state that is reached after safely ferrying a cargo item. */
  State safeFerry(Cargo cargo) { result = this.ferry(cargo) and result.isSafe() }
```

Find paths from one state to another

The main aim of this query is to find a path, that is, a list of successive ferrying actions, to get from the initial state to the goal state. You could write this list by separating each item by a newline (`"\n"`).

When finding the solution, you should be careful to avoid infinite paths. For example, the man could ferry the goat back and forth any number of times without ever reaching an unsafe state. Such a path would have an infinite number of river crossings without ever solving the puzzle.

One way to restrict our paths to a finite number of river crossings is to define a member predicate `State reachesVia(string path, int steps)`. The result of this predicate is any state that is reachable from the current state (`this`) via the given path in a specified finite number of steps.

You can write this as a recursive predicate, with the following base case and recursion step:

- If `this` *is* the result state, then it (trivially) reaches the result state via an empty path in zero steps.
- Any other state is reachable if `this` can reach an intermediate state (for some value of `path` and `steps`), and there is a `safeFerry` action from that intermediate state to the result state.

To ensure that the predicate is finite, you should restrict `steps` to a particular value, for example `steps <= 7`.

Show/hide code

```
/**
 * Returns all states that are reachable via safe ferrying.
 * `path` keeps track of how it is achieved and `steps` keeps track of the number of steps it␣
↪takes.
 */
State reachesVia(string path, int steps) {
  // Trivial case: a state is always reachable from itself
  steps = 0 and this = result and path = ""
  or
  // A state is reachable using pathSoFar and then safely ferrying cargo.
  exists(int stepsSoFar, string pathSoFar, Cargo cargo |
    result = this.reachesVia(pathSoFar, stepsSoFar).safeFerry(cargo) and
    steps = stepsSoFar + 1 and
    // We expect a solution in 7 steps, but you can choose any value here.
    steps <= 7 and
    path = pathSoFar + "\n Ferry " + cargo
  )
}
```

However, although this ensures that the solution is finite, it can still contain loops if the upper bound for `steps` is large. In other words, you could get an inefficient solution by revisiting the same state multiple times.

Instead of picking an arbitrary upper bound for the number of steps, you can avoid counting steps altogether. If you keep track of states that have already been visited and ensure that each ferrying action leads to a new state, the solution certainly wont contain any loops.

To do this, change the member predicate to `State reachesVia(string path, string visitedStates)`. The result of this predicate is any state that is reachable from the current state (`this`) via the given path without revisiting any previously visited states.

- As before, if `this` *is* the result state, then it (trivially) reaches the result state via an empty path and an empty string of visited states.

- Any other state is reachable if `this` can reach an intermediate state via some path, without revisiting any previous states, and there is a `safeFerry` action from the intermediate state to the result state. (Hint: To check whether a state has previously been visited, you could check if there is an index of `visitedStates` at which the state occurs.)

Show/hide code

```
/**
 * Returns all states that are reachable via safe ferrying.
 * `path` keeps track of how it is achieved.
 * `visitedStates` keeps track of previously visited states and is used to avoid loops.
 */
State reachesVia(string path, string visitedStates) {
  // Trivial case: a state is always reachable from itself.
  this = result and
  visitedStates = this and
  path = ""
  or
  // A state is reachable using pathSoFar and then safely ferrying cargo.
  exists(string pathSoFar, string visitedStatesSoFar, Cargo cargo |
    result = this.reachesVia(pathSoFar, visitedStatesSoFar).safeFerry(cargo) and
    // The resulting state has not yet been visited.
    not exists(int i | i = visitedStatesSoFar.indexOf(result)) and
    visitedStates = visitedStatesSoFar + "/" + result and
    path = pathSoFar + "\n Ferry " + cargo
  )
}
```

Display the results

Once youve defined all the necessary classes and predicates, write a select clause that returns the resulting path.

Show/hide code

```
from string path
where any(InitialState i).reachesVia(path, _) = any(GoalState g)
select path
```

The dont-care expression (`_`), as the second argument to the `reachesVia` predicate, represents any value of `visitedStates`.

For now, the path defined in `reachesVia` just lists the order of cargo items to ferry. You could tweak the predicate and the select clause to make the solution clearer. Here are some suggestions:

- Display more information, such as the direction in which the cargo is ferried, for example `"Goat to the left shore"`.
- Fully describe the state at every step, for example `"Goat: Left, Man: Left, Cabbage: Right, Wolf: Right"`.
- Display the path in a more visual way, for example by using arrows to display the transitions between states.

1.5.3 Alternative solutions

Here are some more example queries that solve the river crossing puzzle:

1. This query uses a modified `path` variable to describe the resulting path in more detail.

   See solution in the query console on LGTM.com

2. This query models the man and the cargo items in a different way, using an abstract class and predicate. It also displays the resulting path in a more visual way.

   See solution in the query console on LGTM.com

3. This query introduces algebraic datatypes to model the situation, instead of defining everything as a subclass of `string`.

   See solution in the query console on LGTM.com

1.5.4 Further reading

- QL language reference
- CodeQL tools
- *Introduction to QL*: Work through some simple exercises and examples to learn about the basics of QL and CodeQL.
- *Find the thief*: Take on the role of a detective to find the thief in this fictional village. You will learn how to use logical connectives, quantifiers, and aggregates in QL along the way.
- *Catch the fire starter*: LearnăaboutăQLăpredicatesăandăclassesătoăsolveăyourăsecondămysteryăasăaăQLădetective.
- *Crown the rightful heir*: This is a QL detective puzzle that shows you how to use recursion in QL to write more complex queries.
- *Cross the river*: Use common QL features to write a query that finds a solution to the River crossing logic puzzle.

CHAPTER

TWO

CODEQL QUERIES

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

2.1 About CodeQL queries

CodeQL queries are used to analyze code for issues related to security, correctness, maintainability, and readability.

2.1.1 Overview

CodeQL includes queries to find the most relevant and interesting problems for each supported language. You can also write custom queries to find specific issues relevant to your own project. The important types of query are:

- **Alert queries**: queries that highlight issues in specific locations in your code.
- **Path queries**: queries that describe the flow of information between a source and a sink in your code.

You can add custom queries to custom query packs to analyze your projects in LGTM, use them to analyze a database with the CodeQL CLI, or you can contribute to the standard CodeQL queries in our open source repository on GitHub.

This topic is a basic introduction to query files. You can find more information on writing queries for specific programming languages here, and detailed technical information about QL in the QL language reference. For more information on how to format your code when contributing queries to the GitHub repository, see the CodeQL style guide.

2.1.2 Basic query structure

Queries written with CodeQL have the file extension `.ql`, and contain a `select` clause. Many of the existing queries include additional optional information, and have the following structure:

```
/**
 *
 * Query metadata
 *
 */

import /* ... CodeQL libraries or modules ... */
```

(continues on next page)

(continued from previous page)

```
/* ... Optional, define CodeQL classes and predicates ... */

from /* ... variable declarations ... */
where /* ... logical formula ... */
select /* ... expressions ... */
```

The following sections describe the information that is typically included in a query file for alerts. Path queries are discussed in more detail in *Creating path queries*.

Query metadata

Query metadata is used to identify your custom queries when they are added to the GitHub repository or used in your analysis. Metadata provides information about the querys purpose, and also specifies how to interpret and display the query results. For a full list of metadata properties, see *Metadata for CodeQL queries*. The exact metadata requirement depends on how you are going to run your query:

- If you are contributing a query to the GitHub repository, please read the query metadata style guide.
- If you are adding a custom query to a query pack for analysis using LGTM , see Writing custom queries to include in LGTM analysis.
- If you are analyzing a database using the CodeQL CLI, your query metadata must contain `@kind`.
- If you are running a query in the query console on LGTM or with the CodeQL extension for VS Code, metadata is not mandatory. However, if you want your results to be displayed as either an alert or a path, you must specify the correct `@kind` property, as explained below. For more information, see Using the query console on LGTM.com and Analyzing your projects in the CodeQL for VS Code help.

  Note

  Queries that are contributed to the open source repository, added to a query pack in LGTM, or used to analyze a database with the CodeQL CLI must have a query type (`@kind`) specified. The `@kind` property indicates how to interpret and display the results of the query analysis:

  - Alert query metadata must contain `@kind problem`.
  - Path query metadata must contain `@kind path-problem`.

  When you define the `@kind` property of a custom query you must also ensure that the rest of your query has the correct structure in order to be valid, as described below.

Import statements

Each query generally contains one or more `import` statements, which define the libraries or modules to import into the query. Libraries and modules provide a way of grouping together related types, predicates, and other modules. The contents of each library or module that you import can then be accessed by the query. Our open source repository on GitHub contains the standard CodeQL libraries for each supported language.

When writing your own alert queries, you would typically import the standard library for the language of the project that you are querying, using `import` followed by a language:

- C/C++: `cpp`
- C#: `csharp`
- Go: `go`

- Java: `java`
- JavaScript/TypeScript: `javascript`
- Python: `python`

There are also libraries containing commonly used predicates, types, and other modules associated with different analyses, including data flow, control flow, and taint-tracking. In order to calculate path graphs, path queries require you to import a data flow library into the query file. For more information, see *Creating path queries*.

You can explore the contents of all the standard libraries in the CodeQL library reference documentation or in the GitHub repository.

Optional CodeQL classes and predicates

You can customize your analysis by defining your own predicates and classes in the query. For further information, see Defining a predicate and Defining a class.

From clause

The `from` clause declares the variables that are used in the query. Each declaration must be of the form `<type> <variable name>`. For more information on the available types, and to learn how to define your own types using classes, see the QL language reference.

Where clause

The `where` clause defines the logical conditions to apply to the variables declared in the `from` clause to generate your results. This clause uses aggregations, predicates, and logical formulas to limit the variables of interest to a smaller set, which meet the defined conditions. The CodeQL libraries group commonly used predicates for specific languages and frameworks. You can also define your own predicates in the body of the query file or in your own custom modules, as described above.

Select clause

The `select` clause specifies the results to display for the variables that meet the conditions defined in the `where` clause. The valid structure for the select clause is defined by the `@kind` property specified in the metadata.

Select clauses for alert queries (`@kind problem`) consist of two columns, with the following structure:

```
select element, string
```

- `element`: a code element that is identified by the query, which defines where the alert is displayed.
- `string`: a message, which can also include links and placeholders, explaining why the alert was generated.

You can modify the alert message defined in the final column of the `select` statement to give more detail about the alert or path found by the query using links and placeholders. For further information, see *Defining the results of a query*.

Select clauses for path queries (`@kind path-problem`) are crafted to display both an alert and the source and sink of an associated path graph. For more information, see *Creating path queries*.

2.1.3 Viewing the standard CodeQL queries

One of the easiest ways to get started writing your own queries is to modify an existing query. To view the standard CodeQL queries, or to try out other examples, visit the CodeQL and CodeQL for Go repositories on GitHub.

You can also find examples of queries developed to find security vulnerabilities and bugs in open source software projects on the GitHub Security Lab website and in the associated repository.

2.1.4 Contributing queries

Contributions to the standard queries and libraries are very welcome. For more information, see our contributing guidelines. If you are contributing a query to the open source GitHub repository, writing a custom query for LGTM, or using a custom query in an analysis with the CodeQL CLI, then you need to include extra metadata in your query to ensure that the query results are interpreted and displayed correctly. See the following topics for more information on query metadata:

- *Metadata for CodeQL queries*
- Query metadata style guide on GitHub

Query contributions to the open source GitHub repository may also have an accompanying query help file to provide information about their purpose for other users. For more information on writing query help, see the Query help style guide on GitHub and the *Query help files*.

2.1.5 Query help files

When you write a custom query, we also recommend that you write a query help file to explain the purpose of the query to other users. For more information, see the Query help style guide on GitHub, and the *Query help files*.

2.2 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.

2.2.1 About query metadata

Any query that is run as part of an analysis includes a number of properties, known as query metadata. Metadata is included at the top of each query file as the content of a QLDoc comment. This metadata tells LGTM and the CodeQL extension for VS Code how to handle the query and display its results correctly. It also gives other users information about what the query results mean. For further information on query metadata, see the query metadata style guide in our open source repository on GitHub.

> Note
>
> The exact metadata requirement depends on how you are going to run your query. For more information, see the section on query metadata in *About CodeQL queries*.

2.2.2 Metadata properties

The following properties are supported by all query files:

Property	Value	Description
`@description`	`<text>`	A sentence or short paragraph to describe the purpose of the query and *why* the result is useful or important. The description is written in plain text, and uses single quotes (`'`) to enclose code elements.
`@id`	`<text>`	A sequence of words composed of lowercase letters or digits, delimited by `/` or `-`, identifying and classifying the query. Each query must have a **unique** ID. To ensure this, it may be helpful to use a fixed structure for each ID. For example, the standard LGTM queries have the following format: `<language>/<brief-description>`.
`@kind`	`problem` `path-problem`	Identifies the query is an alert (`@kind problem`) or a path (`@kind path-problem`). For further information on these query types, see *About CodeQL queries*.
`@name`	`<text>`	A statement that defines the label of the query. The name is written in plain text, and uses single quotes (`'`) to enclose code elements.

`@tags`	`correctness`
	`maintainability`
	`readability`
	`security`
		These tags group queries together in broad categories to make it easier to search for them and identify them. In addition to the common tags listed here, there are also a number of more specific categories. For more information, see the Query metadata style guide.

`@precision`	`medium`
	`high`
	`very-high`
		Indicates the percentage of query results that are true positives (as opposed to false positive results).ăThis, along with the `@problem.severity` property, determines whether the results are displayed by default on LGTM.

`@problem.severity`	`error`
	`warning`
	`recommendation`
		Defines the level of severity of any alerts generated by the query. This, along with the `@precision` property, determines whether the results are displayed by default on LGTM.

2.2.3 Additional properties for filter queries

Filter queries are used to define additional constraints to limit the results that are returned by other queries. A filter query must have the same `@kind` property as the query whose results it is filtering. No additional metadata properties are required.

2.2.4 Example

Here is the metadata for one of the standard Java queries:

```
/**
 * @name Type mismatch on container modification
 * @description Calling container modification methods such as 'Collection.remove'
 *              or 'Map.remove' with an object of a type that is incompatible with
 *              the corresponding container element type is unlikely to have any effect.
 * @kind problem
 * @problem.severity error
 * @precision very-high
 * @id java/type-mismatch-modification
 * @tags reliability
 *       correctness
 *       logic
 */
```

For more examples of query metadata, see the standard CodeQL queries in our GitHub repository.

2.3 Query help files

Query help files tell users the purpose of a query, and recommend how to solve the potential problem the query finds.

This topic provides detailed information on the structure of query help files. For more information about how to write useful query help in a style that is consistent with the standard CodeQL queries, see the Query help style guide on GitHub.

> Note
>
> You can access the query help for CodeQL queries by visiting the Built-in query pages. You can also access the raw query help files in the GitHub repository. For example, see the JavaScript security queries and C/C++ critical queries.
>
> For queries run by default on LGTM, there are several different ways to access the query help. For further information, see Where do I see the query help for a query on LGTM? in the LGTM user help.

2.3.1 Overview

Each query help file provides detailed information about the purpose and use of a query. When you write your own queries, we recommend that you also write query help files so that other users know what the queries do, and how they work.

2.3.2 Structure

Query help files are written using a custom XML format, and stored in a file with a `.qhelp` extension. Query help files must have the same base name as the query they describe, and must be located in the same directory. The basic structure is as follows:

```
<!DOCTYPE qhelp SYSTEM "qhelp.dtd">
<qhelp>
    CONTAINS one or more section-level elements
</qhelp>
```

The header and single top-level `qhelp` element are both mandatory. The following sections explain additional elements that you may include in your query help files.

2.3.3 Section-level elements

Section-level elements are used to group the information in the help file into sections. Many sections have a heading, either defined by a `title` attribute or a default value. The following section-level elements are optional child elements of the `qhelp` element.

Element	Attributes	Children	Purpose of section
`example`	None	Any block element	Demonstrate an example of code that violates the rule implemented by the query with guidance on how to fix it. Default heading.
`fragment`	None	Any block element	See *Query help inclusion* below. No heading.
`hr`	None	None	A horizontal rule. No heading.
`include`	`src` The query help file to include.	None	Include a query help file at the location of this element. See *Query help inclusion* below. No heading.
`overview`	None	Any block element	Overview of the purpose of the query. Typically this is the first section in a query document. No heading.
`recommendation`	None	Any block element	Recommend how to address any alerts that this query identifies. Default heading.
`references`	None	`li` elements	Reference list. Typically this is the last section in a query document. Default heading.
`section`	`title` Title of the section	Any block element	General-purpose section with a heading defined by the `title` attribute.
`semmleNotes`	None	Any block element	Implementation notes about the query. This section is used only for queries that implement a rule defined by a third party. Default heading.

2.3.4 Block elements

The following elements are optional child elements of the `section`, `example`, `fragment`,ărecommendation, `overview`, and `semmleNotes` elements.

Element	Attributes	Children	Purpose of block
`blockquote`	None	Any block element	Display a quoted paragraph.
`img`	`src` The image file to include. `alt` Text for the images alt text. `height` Optional, height of the image. `width` Optional, the width of the image.	None	Display an image. The content of the image is in a separate image file.
`include`	`src` The query help file to include.	None	Include a query help file at the location of this element. See *Query help inclusion* below for more information.
`ol`	None	`li`	Display an ordered list. See List elements below.
`p`	None	Any inline content	Display a paragraph, used as in HTML files.
`pre`	None	Text	Display text in a monospaced font with preformatted whitespace.
`sample`	`language` The language of the in-line code sample. `src` Optional, the file containing the sample code.	Text	Display sample code either defined as nested text in the `sample` element or defined in the `src` file specified. When `src` is specified, the language is inferred from the file extension. If `src` is omitted, then language must be provided and the sample code provided as nested text.
`table`	None	`tbody`	Display a table. See Tables below.
`ul`	None	`li`	Display an unordered list. See List elements below.
`warning`	None	Text	Display a warning that will be displayed very visibly on the resulting page. Such warnings are sometimes used on queries that are known to have low precision for many code bases; such queries are often disabled by default.

2.3.5 List elements

Query help files support two types of block elements for lists: `ul` and `ol`. Both block elements support only one child elements of the type `li`. Each `li` element contains either inline content or a block element.

2.3.6 Table elements

The `table` block element is used to include a table in a query help file. Each table includes a number of rows, each of which includes a number of cells. The data in the cells will be rendered as a grid.

Element	Attributes	Children	Purpose
`tbody`	None	`tr`	Defines the top-level element of a table.
`tr`	None	`th` `td`	Defines one row of a table.
`td`	None	Any inline content	Defines one cell of a table row.
`th`	None	Any inline content	Defines one header cell of a table row.

2.3.7 Inline content

Inline content is used to define the content for paragraphs, list items, table cells, and similar elements. Inline content includes text in addition to the inline elements defined below:

Element	Attributes	Children	Purpose
`a`	`href` The URL of the link.	text	Defines hyperlink. When a user selects the child text, they will be redirected to the given URL.
`b`	None	Inline content	Defines content that should be displayed as bold face.
`code`	None	Inline content	Defines content representing code. It is typically shown in a monospace font.
`em`	None	Inline content	Defines content that should be emphasized, typically by italicizing it.
`i`	None	Inline content	Defines content that should be displayed as italics.

`img`	`src`
	`alt`
	`height`
	`width`
		None	Display an image. See the description above in Block elements.

`strong`	None	Inline content	Defines content that should be rendered more strongly, typically using bold face.
`sub`	None	Inline content	Defines content that should be rendered as subscript.
`sup`	None	Inline content	Defines content that should be rendered asăsuperscript.
`tt`	None	Inline content	Defines content that should be displayed with a monospace font.

2.3.8 Query help inclusion

To reuse content between different help topics, you can store shared content in one query help file and then include it in a number of other query help files using the`ăinclude` element. The shared content can be stored either in the same directory as the including files, or in`ăSEMMLE_DIST/docs/include`.

The`ăinclude` element can be used as a section or block element. The content of the query help file defined by the`ăsrc` attribute must contain elements that are appropriate to the location of the`ăinclude` element.

Section-level include elements

Section-level `include` elements can be located beneath the top-level `qhelp` element. For example, in StoredXSS.qhelp, a full query help file is reused:

```
<qhelp>
    <include src="XSS.qhelp" />
</qhelp>
```

In this example, theăXSS.qhelp file must conform to the standard for a full query help file as described above. That is, theăqhelp element may only contain non-`fragment`, section-level elements.

Block-level include elements

Block-level `include` elements can be included beneath section-level elements. For example, an `include` element is used beneath the `overview` section in ThreadUnsafeICryptoTransform.qhelp:

```
<qhelp>
    <overview>
        <include src="ThreadUnsafeICryptoTransformOverview.qhelp" />
    </overview>
    ...
</qhelp>
```

Theăincluded file, ThreadUnsafeICryptoTransformOverview.qhelp, may only contain one or more `fragment` sections. For example:

```
<!DOCTYPE qhelp SYSTEM "qhelp.dtd">
<qhelp>
   <fragment>
      <p>
         ...
      </p>
   </fragment>
</qhelp>
```

2.4 Defining the results of a query

You can control how analysis results are displayed in source code by modifying a querys `select` statement.

2.4.1 About query results

The information contained in the results of a query is controlled by the `select` statement. Part of the process of developing a useful query is to make the results clear and easy for other users to understand. When you write your own queries in the query console or in the CodeQL extension for VS Code there are no constraints on what can be selected. However, if you want to use a query to create alerts in LGTM or generate valid analysis results using the CodeQL CLI, youll need to make the `select` statement report results in the required format. You must also ensure that the query has the appropriate metadata properties defined. This topic explains how to write your select statement to generate helpful analysis results.

2.4.2 Overview

Alert queries must have the property `@kind problem` defined in their metadata. For further information, see *Metadata for CodeQL queries*. In their most basic form, the `select` statement must select two columns:

- **Element**a code element thats identified by the query. This defines the location of the alert.
- **String**a message to display for this code element, describing why the alert was generated.

If you look at some of the LGTM queries, youll see that they can select extra element/string pairs, which are combined with `$@` placeholder markers in the message to form links. For example, Dereferenced variable may be null (Java), or Duplicate switch case (JavaScript).

> Note
>
> An in-depth discussion of `select` statements for path queries is not included in this topic. However, you can develop the string column of the `select` statement in the same way as for alert queries. For more specific information about path queries, see *Creating path queries*.

2.4.3 Developing a select statement

Heres a simple query that uses the standard CodeQL `CodeDuplication.qll` library to identify similar files.

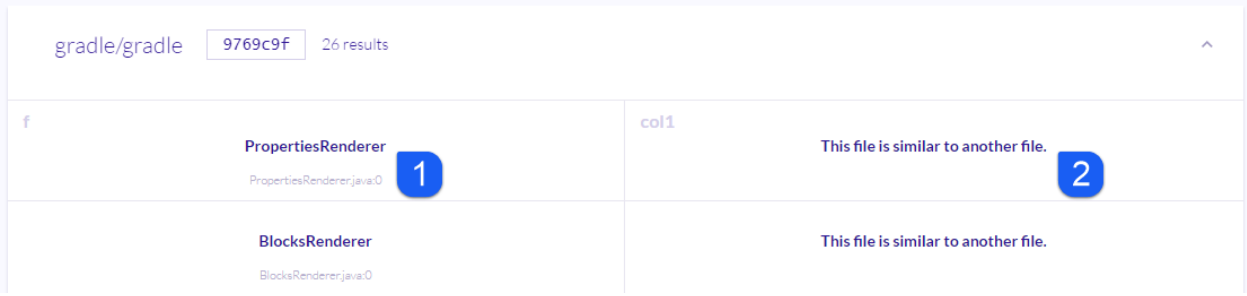
Basic select statement

```
import java
import external.CodeDuplication

from File f, File other, int percent
where similarFiles(f, other, percent)
select f, "This file is similar to another file."
```

This basic select statement has two columns:

1. Element to display the alert on: `f` corresponds to `File`.
2. String message to display: `"This file is similar to another file."`

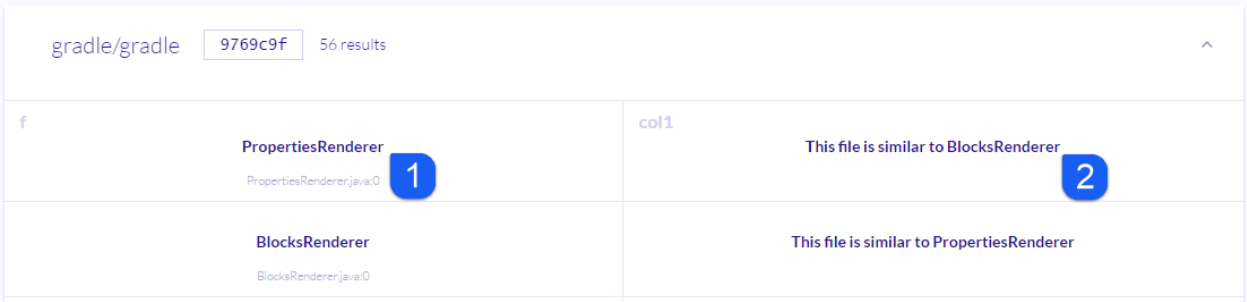


Including the name of the similar file

The alert message defined by the basic select statement is constant and doesnt give users much information. Since the query identifies the similar file (`other`), its easy to extend the `select` statement to report the name of the similar file. For example:

```
select f, "This file is similar to " + other.getBaseName()
```

1. Element: `f` as before.
2. String message: `"This file is similar to "`the string text is combined with the file name for the `other`, similar file, returned by `getBaseName()`.



While this is more informative than the original select statement, the user still needs to find the other file manually.

Adding a link to the similar file

You can use placeholders in the text of alert messages to insert additional information, such as links to the similar file. Placeholders are defined using `$@`, and filled using the information in the next two columns of the select statement. For example, this select statement returns four columns:

```
select f, "This file is similar to $@.", other, other.getBaseName()
```

1. Element: `f` as before.
2. String message: `"This file is similar to $@."`the string text now includes a placeholder, which will display the combined content of the next two columns.
3. Element for placeholder: `other` corresponds to the similar file.
4. String text for placeholder: the short file name returned by `other.getBaseName()`.

When the alert message is displayed, the `$@` placeholder is replaced by a link created from the contents of the third and fourth columns defined by the `select` statement.

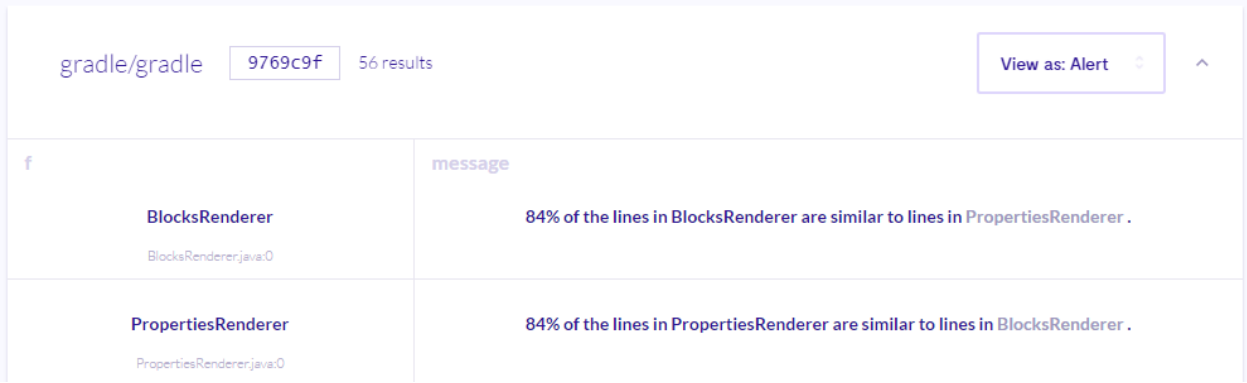
If you use the `$@` placeholder marker multiple times in the description text, then the `N`th use is replaced by a link formed from columns `2N+2` and `2N+3`. If there are more pairs of additional columns than there are placeholder markers, then the trailing columns are ignored. Conversely, if there are fewer pairs of additional columns than there are placeholder markers, then the trailing markers are treated as normal text rather than placeholder markers.

Adding details of the extent of similarity

You could go further and change the `select` statement to report on the similarity of content in the two files, since this information is already available in the query. For example:

```
select f, percent + "% of the lines in " + f.getBaseName() + " are similar to lines in $@.", other,
↪ other.getBaseName()
```

The new elements added here dont need to be clickable, so we added them directly to the description string.



2.4.4 Further reading

- CodeQL repository

2.5 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.

2.5.1 About locations

When displaying information to the user, LGTM needs to be able to extract location information from the results of a query. In order to do this, all QL classes which can provide location information should do this by using one of the following mechanisms:

- *Providing URLs*
- *Providing location information*
- *Using extracted location information*

This list is in priority order, so that the first available mechanism is used.

> Note
>
> Since QL is a relational language, there is nothing to enforce that each entity of a QL class is mapped to precisely one location. This is the responsibility of the designer of the library (or the extractor, in the case of the third option below). If entities are assigned no location at all, users will not be able to click through from query results to the source code viewer. If multiple locations are assigned, results may be duplicated.

Providing URLs

A custom URL can be provided by defining a QL predicate returning `string` with the name `getURL` – note that capitalization matters, and no arguments are allowed. For example:

```
class JiraIssue extends ExternalData {
    JiraIssue() {
```

(continues on next page)

(continued from previous page)

```
        getDataPath() = "JiraIssues.csv"
    }

    string getKey() {
        result = getField(0)
    }

    string getURL() {
        result = "http://mycompany.com/jira/" + getKey()
    }
}
```

File URLs

LGTM supports the display of URLs which define a line and column in a source file.

The schema is `file://`, which is followed by the absolute path to a file, followed by four numbers separated by colons. The numbers denote start line, start column, end line and end column. Both line and column numbers are **1-based**, for example:

- `file://opt/src/my/file.java:0:0:0:0` is used to link to an entire file.
- `file:///opt/src/my/file.java:1:1:2:1` denotes the location that starts at the beginning of the file and extends to the first character of the second line (the range is inclusive).
- `file:///opt/src/my/file.java:1:0:1:0` is taken, by convention, to denote the entire first line of the file.

By convention, the location of an entire file may also be denoted by a `file://` URL without trailing numbers. Optionally, the location within a file can be denoted using three numbers to define the start line number, character offset and character length of the location respectively. Results of these types are not displayed in LGTM.

Other types of URL

The following, less-common types of URL are valid but are not supported by LGTM and will be omitted from any results:

- **HTTP URLs** are supported in some client applications. For an example, see the code snippet above.
- **Folder URLs** can be useful, for example to provide folder-level metrics. They may use a file URL, for example `file:///opt/src:0:0:0:0`, but they may also start with a scheme of `folder://`, and no trailing numbers, for example `folder:///opt/src`.
- **Relative file URLs** are like normal file URLs, but start with the scheme `relative://`. They are typically only meaningful in the context of a particular database, and are taken to be implicitly prefixed by the databases source location. Note that, in particular, the relative URL of a file will stay constant regardless of where the database is analyzed. It is often most convenient to produce these URLs as input when importing external information; selecting one from a QL class would be unusual, and client applications may not handle it appropriately.

Providing location information

If no `getURL()` member predicate is defined, a QL class is checked for the presence of a member predicate called `hasLocationInfo(..)`. This can be understood as a convenient way of providing file URLs (see above) without constructing the long URL string in QL. `hasLocationInfo(..)` should be a predicate, its first column must be `string`-typed (it corresponds to the path portion of a file URL), and it must have an additional 3 or 4 `int`-typed columns, which are interpreted like a trailing group of three or four numbers on a file URL.

For example, let us imagine that the locations for methods provided by the extractor extend from the first character of the method name to the closing curly brace of the method body, and we want to fix them to ensure that only the method name is selected. The following code shows two ways of achieving this:

```
class MyMethod extends Method {
    // The locations from the database, which we want to modify.
    Location getLocation() { result = super.getLocation() }

    /* First member predicate: Construct a URL for the desired location. */
    string getURL() {
        exists(Location loc | loc = this.getLocation() |
            result = "file://" + loc.getFile().getFullName() +
                ":" + loc.getStartLine() +
                ":" + loc.getStartColumn() +
                ":" + loc.getStartLine() +
                ":" + (loc.getStartColumn() + getName().length() - 1)
        )
    }

    /* Second member predicate: Define hasLocationInfo. This will be more
       efficient (it avoids constructing long strings), and will
       only be used if getURL() is not defined. */
    predicate hasLocationInfo(string path, int sl, int sc, int el, int ec) {
        exists(Location loc | loc = this.getLocation() |
            path = loc.getFile().getFullName() and
            sl = loc.getStartLine() and
            sc = loc.getStartColumn() and
            el = sl and
            ec = sc + getName().length() - 1
        )
    }
}
```

Using extracted location information

Finally, if the above two predicates fail, client applications will attempt to call a predicate called `getLocation()` with no parameters, and try to apply one of the above two predicates to the result. This allows certain locations to be put into the database, assigned identifiers, and picked up.

By convention, the return value of the `getLocation()` predicate should be a class called `Location`, and it should define a version of `hasLocationInfo(..)` (or `getURL()`, though the former is preferable). If the `Location` class does not provide either of these member predicates, then no location information will be available.

2.5.2 The `toString()` predicate

All classes except those that extend primitive types, must provide a `string toString()` member predicate. The query compiler will complain if you dont. The uniqueness warning, noted above for locations, applies here too.

2.5.3 Further reading

- CodeQL repository

2.6 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.

2.6.1 Overview

Many CodeQL security queries implement data flow analysis, which can highlight the fate of potentially malicious or insecure data that can cause vulnerabilities in your code base. These queries help you understand if data is used in an insecure way, whether dangerous arguments are passed to functions, or whether sensitive data can leak. As well as highlighting potential security issues, you can also use data flow analysis to understand other aspects of how a program behaves, by finding, for example, uses of uninitialized variables and resource leaks.

The following sections provide a brief introduction to data flow analysis with CodeQL.

See the following tutorials for more information about analyzing data flow in specific languages:

- *Analyzing data flow in C/C++*
- *Analyzing data flow in C#*
- *Analyzing data flow in Java*
- *Analyzing data flow in JavaScript/TypeScript*
- *Analyzing data flow and tracking tainted data in Python*

  Note

  Data flow analysis is used extensively in path queries. To learn more about path queries, see *Creating path queries*.

2.6.2 Data flow graph

The CodeQL data flow libraries implement data flow analysis on a program or function by modeling its data flow graph. Unlike the abstract syntax tree, the data flow graph does not reflect the syntactic structure of the program, but models the way data flows through the program at runtime. Nodes in the abstract syntax tree represent syntactic elements such as statements or expressions. Nodes in the data flow graph, on the other hand, represent semantic elements that carry values at runtime.

Some AST nodes (such as expressions) have corresponding data flow nodes, but others (such as `if` statements) do not. This is because expressions are evaluated to a value at runtime, whereas `if` statements are purely a control-flow construct and do not carry values. There are also data flow nodes that do not correspond to AST nodes at all.

Edges in the data flow graph represent the way data flows between program elements. For example, in the expression `x || y` there are data flow nodes corresponding to the sub-expressions `x` and `y`, as well as a data flow

node corresponding to the entire expression `x || y`. There is an edge from the node corresponding to `x` to the node corresponding to `x || y`, representing the fact that data may flow from `x` to `x || y` (since the expression `x || y` may evaluate to `x`). Similarly, there is an edge from the node corresponding to `y` to the node corresponding to `x || y`.

Local and global data flow differ in which edges they consider: local data flow only considers edges between data flow nodes belonging to the same function and ignores data flow between functions and through object properties. Global data flow, however, considers the latter as well. Taint tracking introduces additional edges into the data flow graph that do not precisely correspond to the flow of values, but model whether some value at runtime may be derived from another, for instance through a string manipulating operation.

The data flow graph is computed using classes to model the program elements that represent the graphs nodes. The flow of data between the nodes is modeled using predicates to compute the graphs edges.

Computing an accurate and complete data flow graph presents several challenges:

- It isnt possible to compute data flow through standard library functions, where the source code is unavailable.
- Some behavior isnt determined until run time, which means that the data flow library must take extra steps to find potential call targets.
- Aliasing between variables can result in a single write changing the value that multiple pointers point to.
- The data flow graph can be very large and slow to compute.

To overcome these potential problems, two kinds of data flow are modeled in the libraries:

- Local data flow, concerning the data flow within a single function. When reasoning about local data flow, you only consider edges between data flow nodes belonging to the same function. It is generally sufficiently fast, efficient and precise for many queries, and it is usually possible to compute the local data flow for all functions in a CodeQL database.
- Global data flow, effectively considers the data flow within an entire program, by calculating data flow between functions and through object properties. Computing global data flow is typically more time and energy intensive than local data flow, therefore queries should be refined to look for more specific sources and sinks.

Many CodeQL queries contain examples of both local and global data flow analysis. See the built-in queries for details.

2.6.3 Normal data flow vs taint tracking

In the standard libraries, we make a distinction between normal data flow and taint tracking. The normal data flow libraries are used to analyze the information flow in which data values are preserved at each step.

For example, if you are tracking an insecure object `x` (which might be some untrusted or potentially malicious data), a step in the program may change its value. So, in a simple process such as `y = x + 1`, a normal data flow analysis will highlight the use of `x`, but not `y`. However, since `y` is derived from `x`, it is influenced by the untrusted or tainted information, and therefore it is also tainted. Analyzing the flow of the taint from `x` to `y` is known as taint tracking.

In QL, taint tracking extends data flow analysis by including steps in which the data values are not necessarily preserved, but the potentially insecure object is still propagated. These flow steps are modeled in the taint-tracking library using predicates that hold if taint is propagated between nodes.

2.6.4 Further reading

- Exploring data flow with path queries

2.7 Creating path queries

You can create path queries to visualize the flow of information through a codebase.

2.7.1 Overview

Security researchers are particularly interested in the way that information flows in a program. Many vulnerabilities are caused by seemingly benign data flowing to unexpected locations, and being used in a malicious way. Path queries written with CodeQL are particularly useful for analyzing data flow as they can be used to track the path taken by a variable from its possible starting points (`source`) to its possible end points (`sink`). To model paths, your query must provide information about the `source` and the `sink`, as well as the data flow steps that link them.

This topic provides information on how to structure a path query file so you can explore the paths associated with the results of data flow analysis.

> Note
>
> The alerts generated by path queries are displayed by default in LGTM and included in the results generated using the CodeQL CLI. You can also view the path explanations generated by your path query directly in LGTM or in the CodeQL extension for VS Code.

To learn more about modeling data flow with CodeQL, see *Introduction to data flow*. For more language-specific information on analyzing data flow, see:

- *Analyzing data flow in C/C++*
- *Analyzing data flow in C#*
- *Analyzing data flow in Java*
- *Analyzing data flow in JavaScript/TypeScript*
- *Analyzing data flow and tracking tainted data in Python*

Path query examples

The easiest way to get started writing your own path query is to modify one of the existing queries. Visit the links below to see all the built-in path queries:

- C/C++ path queries
- C# path queries
- Java path queries
- JavaScript path queries
- Python path queries

The Security Lab researchers have used path queries to find security vulnerabilities in various open source projects. To see articles describing how these queries were written, as well as other posts describing other aspects of security research such as exploiting vulnerabilities, see the GitHub Security Lab website.

2.7.2 Constructing a path query

Path queries require certain metadata, query predicates, and `select` statement structures. Many of the built-in path queries included in CodeQL follow a simple structure, which depends on how the language you are analyzing is modeled with CodeQL.

For C/C++, C#, Java, and JavaScript you should use the following template:

```
/**
 * ...
 * @kind path-problem
 * ...
 */

import <language>
import DataFlow::PathGraph
...

from Configuration config, DataFlow::PathNode source, DataFlow::PathNode sink
where config.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "<message>"
```

Where:

- `DataFlow::Pathgraph` is the path graph module you need to import from the standard CodeQL libraries.
- `source` and `sink` are nodes on the path graph, and `DataFlow::PathNode` is their type.
- `Configuration` is a class containing the predicates which define how data may flow between the `source` and the `sink`.

For Python you should use a slightly different template:

```
/**
 * ...
 * @kind path-problem
 * ...
 */

import python
import semmle.python.security.Paths
...

from TaintedPathSource source, TaintedPathSink sink
where source.flowsTo(sink)
select sink.getNode(), source, sink, "<message>"
```

Where:

- `semmle.python.security.Paths` is the path graph module imported from the standard CodeQL libraries.
- `source` and `sink` are nodes on the path graph, `TaintedPathSource source` and `TaintedPathSink` are their respective types. Note, you do not need to declare a configuration class to define the data flow from the `source` to the `sink` in a Python path query.

The following sections describe the main requirements for a valid path query.

Path query metadata

Path query metadata must contain the property `@kind path-problem`–this ensures that query results are interpreted and displayed correctly. The other metadata requirements depend on how you intend to run the query. For more information, see Query metadata.

Generating path explanations

In order to generate path explanations, your query needs to compute a path graph. To do this you need to define a query predicate called `edges` in your query. This predicate defines the edge relations of the graph you are computing, and it is used to compute the paths related to each result that your query generates. You can import a predefined `edges` predicate from a path graph module in one of the standard data flow libraries. In addition to the path graph module, the data flow libraries contain the other `classes`, `predicates`, and `modules` that are commonly used in data flow analysis. The import statement to use depends on the language that you are analyzing.

For C/C++, C#, Java, and JavaScript you would use:

```
import DataFlow::PathGraph
```

This statement imports the `PathGraph` module from the data flow library (`DataFlow.qll`), in which `edges` is defined.

For Python, the `Paths` module contains the `edges` predicate:

```
import semmle.python.security.Paths
```

You can also import libraries specifically designed to implement data flow analysis in various common frameworks and environments, and many additional libraries are included with CodeQL. To see examples of the different libraries used in data flow analysis, see the links to the built-in queries above or browse the standard libraries.

For all languages, you can also optionally define a `nodes` query predicate, which specifies the nodes of the path graph that you are interested in. If `nodes` is defined, only edges with endpoints defined by these nodes are selected. If `nodes` is not defined, you select all possible endpoints of `edges`.

Defining your own `edges` predicate

You can also define your own `edges` predicate in the body of your query. It should take the following form:

```
query predicate edges(PathNode a, PathNode b) {
/** Logical conditions which hold if `(a,b)` is an edge in the data flow graph */
}
```

For more examples of how to define an `edges` predicate, visit the standard CodeQL libraries and search for `edges`.

Declaring sources and sinks

You must provide information about the `source` and `sink` in your path query. These are objects that correspond to the nodes of the paths that you are exploring. The name and the type of the `source` and the `sink` must be declared in the `from` statement of the query, and the types must be compatible with the nodes of the graph computed by the `edges` predicate.

If you are querying C/C++, C#, Java, or JavaScript code (and you have used `import DataFlow::PathGraph` in your query), the definitions of the `source` and `sink` are accessed via the `Configuration` class in the data flow library. You should declare all three of these objects in the `from` statement. For example:

```
from Configuration config, DataFlow::PathNode source, DataFlow::PathNode sink
```

The configuration class is accessed by importing the data flow library. This class contains the predicates which define how data flow is treated in the query:

- `isSource()` defines where data may flow from.
- `isSink()` defines where data may flow to.

For further information on using the configuration class in your analysis see the sections on global data flow in *Analyzing data flow in C/C++* and *Analyzing data flow in C#*.

You can also create a configuration for different frameworks and environments by extending the `Configuration` class. For further information, see defining a class.

If you are querying Python code (and you have used `import semmle.python.security.Paths` in your query) you should declare `TaintedPathSource source, TaintedPathSink sink` in your `from` statement. You do not need to declare a `Configuration` class as the definitions of the `TaintedPathSource` and `TaintedPathSink` contain all of the type information that is required:

```
from TaintedPathSource source, TaintedPathSink sink
```

You can extend your query by adding different sources and sinks by either defining them in the query, or by importing predefined sources and sinks for specific frameworks and libraries. See the Python path queries for further details.

Defining flow conditions

The `where` clause defines the logical conditions to apply to the variables declared in the `from` clause to generate your results. This clause can use aggregations, predicates, and logical formulas to limit the variables of interest to a smaller set which meet the defined conditions.

When writing a path queries, you would typically include a predicate that holds only if data flows from the `source` to the `sink`.

For C/C++, C#, Java or JavaScript, you would use the `hasFlowPath` predicate to define flow from the `source` to the `sink` for a given `Configuration`:

```
where config.hasFlowPath(source, sink)
```

For Python, you would simply use the `flowsTo` predicate to define flow from the `source` to the `sink`:

```
where source.flowsTo(sink)
```

Select clause

Select clauses for path queries consist of four columns, with the following structure:

```
select element, source, sink, string
```

The `element` and `string` columns represent the location of the alert and the alert message respectively, as explained in *Introduction to writing queries*. The second and third columns, `source` and `sink`, are nodes on the path graph selected by the query. Each result generated by your query is displayed at a single location in the same way as an alert query. Additionally, each result also has an associated path, which can be viewed in LGTM or in the CodeQL extension for VS Code.

The `element` that you select in the first column depends on the purpose of the query and the type of issue that it is designed to find. This is particularly important for security issues. For example, if you believe the `source` value to be globally invalid or malicious it may be best to display the alert at the `source`. In contrast, you should consider displaying the alert at the `sink` if you believe it is the element that requires sanitization.

The alert message defined in the final column in the `select` statement can be developed to give more detail about the alert or path found by the query using links and placeholders. For more information, see *Defining the results of a query*.

Further reading

- Exploring data flow with path queries
- CodeQL repository

2.8 Troubleshooting query performance

Improve the performance of your CodeQL queries by following a few simple guidelines.

2.8.1 About query performance

This topic offers some simple tips on how to avoid common problems that can affect the performance of your queries. Before reading the tips below, it is worth reiterating a few important points about CodeQL and the QL language:

- CodeQL predicates and classes are evaluated to database tables. Large predicates generate large tables with many rows, and are therefore expensive to compute.
- The QL language is implemented using standard database operations and relational algebra (such as join, projection, and union). For further information about query languages and databases, see About the QL language.
- Queries are evaluated *bottom-up*, which means that a predicate is not evaluated until *all* of the predicates that it depends on are evaluated. For more information on query evaluation, see Evaluation of QL programs.

2.8.2 Performance tips

Follow the guidelines below to ensure that you dont get tripped up by the most common CodeQL performance pitfalls.

Eliminate cartesian products

The performance of a predicate can often be judged by considering roughly how many results it has. One way of creating badly performing predicates is by using two variables without relating them in any way, or only relating them using a negation. This leads to computing the Cartesian product between the sets of possible values for each variable, potentially generating a huge table of results. This can occur if you dont specify restrictions on

your variables. For instance, consider the following predicate that checks whether a Java method `m` may access a field `f`:

```
predicate mayAccess(Method m, Field f) {
  f.getAnAccess().getEnclosingCallable() = m
  or
  not exists(m.getBody())
}
```

The predicate holds if `m` contains an access to `f`, but also conservatively assumes that methods without bodies (for example, native methods) may access *any* field.

However, if `m` is a native method, the table computed by `mayAccess` will contain a row `m, f` for *all* fields `f` in the codebase, making it potentially very large.

This example shows a similar mistake in a member predicate:

```
class Foo extends Class {
  ...
  // BAD! Does not use this
  Method getToString() {
    result.getName() = "ToString"
  }
  ...
}
```

Note that while `getToString()` does not declare any parameters, it has two implicit parameters, `result` and `this`, which it fails to relate. Therefore, the table computed by `getToString()` contains a row for every combination of `result` and `this`. That is, a row for every combination of a method named `"ToString"` and an instance of `Foo`. To avoid making this mistake, `this` should be restricted in the member predicate `getToString()` on the class `Foo`.

Use specific types

Types provide an upper bound on the size of a relation. This helps the query optimizer be more effective, so its generally good to use the most specific types possible. For example:

```
predicate foo(LoggingCall e)
```

is preferred over:

```
predicate foo(Expr e)
```

From the type context, the query optimizer deduces that some parts of the program are redundant and removes them, or *specializes* them.

Determine the most specific types of a variable

If you are unfamiliar with the library used in a query, you can use CodeQL to determine what types an entity has. There is a predicate called `getAQlClass()`, which returns the most specific QL types of the entity that it is called on.

For example, if you were working with a Java database, you might use `getAQlClass()` on every `Expr` in a callable called `c`:

```
import java

from Expr e, Callable c
where
    c.getDeclaringType().hasQualifiedName("my.namespace.name", "MyClass")
    and c.getName() = "c"
    and e.getEnclosingCallable() = c
select e, e.getAQlClass()
```

The result of this query is a list of the most specific types of every `Expr` in that function. You will see multiple results for expressions that are represented by more than one type, so it will likely return a very large table of results.

Use `getAQlClass()` as a debugging tool, but dont include it in the final version of your query, as it slows down performance.

Avoid complex recursion

Recursion is about self-referencing definitions. It can be extremely powerful as long as it is used appropriately. On the whole, you should try to make recursive predicates as simple as possible. That is, you should define a *base case* that allows the predicate to *bottom out*, along with a single *recursive call*:

```
int depth(Stmt s) {
  exists(Callable c | c.getBody() = s | result = 0) // base case
  or
  result = depth(s.getParent()) + 1 // recursive call
}
```

> Note
>
> The query optimizer has special data structures for dealing with transitive closures. If possible, use a transitive closure over a simple recursive predicate, as it is likely to be computed faster.

Fold predicates

Sometimes you can assist the query optimizer by folding parts of large predicates out into smaller predicates.

The general principle is to split off chunks of work that are:

- **linear**, so that there is not too much branching.
- **tightly bound**, so that the chunks join with each other on as many variables as possible.

In the following example, we explore some lookups on two `Elements`:

```
predicate similar(Element e1, Element e2) {
  e1.getName() = e2.getName() and
  e1.getFile() = e2.getFile() and
  e1.getLocation().getStartLine() = e2.getLocation().getStartLine()
}
```

Going from `Element -> File` and `Element -> Location -> StartLine` is linear–that is, there is only one `File`, `Location`, etc. for each `Element`.

However, as written it is difficult for the optimizer to pick out the best ordering. Joining first and then doing the linear lookups later would likely result in poor performance. Generally, we want to do the quick, linear parts first, and then join on the resultant larger tables. We can initiate this kind of ordering by splitting the above predicate as follows:

```
predicate locInfo(Element e, string name, File f, int startLine) {
  name = e.getName() and
  f = e.getFile() and
  startLine = e.getLocation().getStartLine()
}

predicate sameLoc(Element e1, Element e2) {
  exists(string name, File f, int startLine |
    locInfo(e1, name, f, startLine) and
    locInfo(e2, name, f, startLine)
  )
}
```

Now the structure we want is clearer. Weve separated out the easy part into its own predicate `locInfo`, and the main predicate `sameLoc` is just a larger join.

2.8.3 Further reading

- QL language reference
- CodeQL tools
- *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 querys `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.

CHAPTER

THREE

CODEQL FOR C AND C++

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from C and C++ codebases.

3.1 Basic query for C and C++ code

Learn to write and run a simple CodeQL query using LGTM.

3.1.1 About the query

The query were going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they have an empty then branch. For example, code such as:

```
if (error) { }
```

3.1.2 Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **C/C++** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import cpp

from IfStmt ifstmt, Block block
where ifstmt.getThen() = block and
  block.getNumStmt() = 0
select ifstmt, "This 'if' statement is redundant."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis () at the bottom of the table indicates that the entire list is not displayedclick it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import cpp`	Imports the standard CodeQL libraries for C/C++.	Every query begins with one or more `import` statements.
`from IfStmt ifstmt, Block block`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `IfStmt` variable for `if` statements • a `Block` variable for the statement block
`where ifstmt.getThen() = block and block.getNumStmt() = 0`	Defines a condition on the variables.	`ifstmt.getThen() = block` relates the two variables. The block must be the `then` branch of the `if` statement. `block.getNumStmt() = 0` states that the block must be empty (that is, it contains no statements).
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

3.1.3 Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if (...) {
  ...
} else if (!strcmp(option, "-verbose") {
  // nothing to do - handled earlier
} else {
  error("unrecognized option");
}
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to ignore empty `then` branches if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Extend the `where` clause to include the following extra condition:

```
and not ifstmt.hasElse()
```

The `where` clause is now:

```
where ifstmt.getThen() = block and
  block.getNumStmt() = 0 and
  not ifstmt.hasElse()
```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer reported.

See this in the query console

3.1.4 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.2 CodeQL library for C and C++

When analyzing C or C++ code, you can use the large collection of classes in the CodeQL library for C and C++.

3.2.1 About the CodeQL library for C and C++

There is an extensive library for analyzing CodeQL databases extracted from C/C++ projects. The classes in this library present the data from a database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks. The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `cpp.qll` imports all the core C/C++ library modules, so you can include the complete library by beginning your query with:

```
import cpp
```

The rest of this topic summarizes the available CodeQL classes and corresponding C/C++ constructs.

3.2.2 Commonly-used library classes

The most commonly used standard library classes are listed below. The listing is broken down by functionality. Each library class is annotated with a C/C++ construct it corresponds to.

Declaration classes

This table lists Declaration classes representing C/C++ declarations.

Example syntax	CodeQL class	Remarks
`int var ;`	GlobalVariable	
`namespace N { float var ; }`	NamespaceVariable	
`int func ( void ) { float var ; }`	LocalVariable	See also Initializer
`class C { int var ; }`	MemberVariable	
`int func (const char param );`	Function	
`template < typename T > void func ( T param);`	TemplateFunction	
`int func (const char* format , ...) { }`	FormattingFunction	
`func < int, float > ( );`	FunctionTemplateInstantiation	
`template < typename T > func < int, T > ( ) { }`	FunctionTemplateSpecialization	
`class C { int func ( float param ); };`	MemberFunction	
`class C { int func ( float param ) const; };`	ConstMemberFunction	
`class C { virtual int func ( ) { } };`	VirtualFunction	
`class C { C ( ) { } };`	Constructor	
`C::operator float () const;`	ConversionOperator	
`class C { ~C ( void ) { } };`	Destructor	
`class C { C ( const D & d ) { } };`	ConversionConstructor	
`C & C :: operator= (const C & );`	CopyAssignmentOperator	
`C & C :: operator= ( C && );`	MoveAssignmentOperator	
`C :: C (const C & );`	CopyConstructor	
`C :: C ( C && );`	MoveConstructor	
`C :: C (void);`	NoArgConstructor	Default constructor

Continued on next page

Table 1 – continued from previous page

Example syntax	CodeQL class	Remarks
`enum en { val1 , val2 }`	EnumConstant	
`friend void func ( int );` `friend class B ;`	FriendDecl	
`int func ( void ) {` `  enum en { val1 , val2 }; }`	LocalEnum	
`class C {` `  enum en { val1 , val2 } }`	NestedEnum	
`enum class en : short { val1 , val2 }`	ScopedEnum	
`class C {` `  virtual void func ( int ) = 0; };`	AbstractClass	
`template < int , float > class C { };`	ClassTemplateInstantiation	
`template < > class C < Type > { };`	FullClassTemplateSpecialization	
`template < typename T >` `class C < T , 5 > { };`	PartialClassTemplateSpecialization	
`int func ( void ) { class C { }; }`	LocalClass	
`class C { class D { }; };`	NestedClass	

`class C {`
`  Type var ;`
`  Type func ( Parameter ) { } };`
	Class	

`struct S {`
`  Type var ;`
`  Type func ( Parameter ) { } };`
	Struct
	Class	

Continued on next page

Table 1 – continued from previous page

Example syntax	CodeQL class	Remarks
`union U { Type var1 ; Type var2 ; };`	Union Struct Class	
`template < typename T > struct C : T { };`	ProxyClass	Appears only in *uninstantiated* templates
`int func ( void ) { struct S { }; }`	LocalStruct	
`class C { struct S { }; };`	NestedStruct	
`int * func ( void ) { union U { }; }`	LocalUnion	
`class C { union U { }; };`	NestedUnion	
`typedef int T ;`	TypedefType	
`int func ( void ) { typedef int T ; }`	LocalTypedefType	
`class C { typedef int T ; };`	NestedTypedefType	
`class V : public B { };`	ClassDerivation	
`class V : virtual B { };`	VirtualClassDerivation	
`template < typename T > class C { };`	TemplateClass	
`int foo ( Type param1 , Type param2 );`	Parameter	
`template <typename T > T t ;`	TemplateVariable	Since C++14

Statement classes

This table lists subclasses of Stmt representing C/C++ statements.

Example syntax	CodeQL class	Remarks
`__asm__ (" movb %bh, (%eax) ");`	AsmStmt	Specific to a given CPU instruction set
`{` Stmt `}`	Block	
`catch (` Parameter `)` Block	CatchBlock	
`catch ( ... )` Block	CatchAnyBlock	
`goto *` *labelptr* `;`	ComputedGotoStmt	GNU extension; use with LabelLiteral
Type *i* `,` *j* `;`	DeclStmt	
`if (` Expr `)` Stmt `else` Stmt	IfStmt	
`switch (` Expr `) {` SwitchCase `}`	SwitchStmt	
`do` Stmt `while (` Expr `)`	DoStmt	
`for (` DeclStmt `;` Expr `;` Expr `)` Stmt	ForStmt	
`for (` DeclStmt `:` Expr `)` Stmt	RangeBasedForStmt	
`while (` Expr `)` Stmt	WhileStmt	
Expr `;`	ExprStmt	
`__try { } __except (` Expr `) { }`	MicrosoftTryExceptStmt	Structured exception handling (SEH) under Windows
`__try { } __finally { }`	MicrosoftTryFinallyStmt	Structured exception handling (SEH) under Windows
`return` Expr `;`	ReturnStmt	
`case` Expr `:`	SwitchCase	
`try {` Stmt `}` CatchBlock CatchAnyBlock	TryStmt	
`void` *func* `(void) try {` Stmt `}` CatchBlock CatchAnyBlock	FunctionTryStmt	
`;`	EmptyStmt	
`break;`	BreakStmt	
`continue;`	ContinueStmt	
`goto` LabelStmt `;`	GotoStmt	
slabel `:`	LabelStmt	
`float` *arr* `[` Expr `] [` Expr `];`	VlaDeclStmt	C99 variable-length array

Expression classes

This table lists subclasses of Expr representing C/C++ expressions.

Example syntax	CodeQL class(es)	Remarks
`{` Expr `}`	ArrayAggregateLiteral ClassAggregateLiteral	

Continued on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
`alignof ( Expr )`	AlignofExprOperator	
`alignof ( Type )`	AlignofTypeOperator	
Expr `[ Expr ]`	ArrayExpr	
`__assume ( Expr )`	AssumeExpr	Microsoft extension
`static_assert ( Expr , StringLiteral ) _Static_assert ( Expr , StringLiteral )`	StaticAssert	C++11 C11
`__noop;`	BuiltInNoOp	Microsoft extension
Expr `( Expr )`	ExprCall	
func `( Expr )` *instance* `.` *func* `( Expr )`	FunctionCall	
Expr `,` Expr	CommaExpr	
`if ( Type` *arg* `= Expr )`	ConditionDeclExpr	
`( Type )` Expr	CStyleCast	
`const_cast < Type > ( Expr )`	ConstCast	
`dynamic_cast < Type > ( Expr )`	DynamicCast	
`reinterpret_cast < Type > ( Expr )`	ReinterpretCast	
`static_cast < Type > ( Expr )`	StaticCast	

`template < typename...` *T* `>`
`auto` *sum* `(` *T* *t* `)`
`{ return (` *t* `+ ... + 0 ); }`
	FoldExpr	Appears only in *uninstantiated* templates

`int` *func* `(` *format* `, ... );`	FormattingFunctionCall	
`[ = ] ( float` *b* `) -> float` `{ return` *captured* `*` *b* `; }`	LambdaExpression	C++11
`^ int ( int` *x* `, int` *y* `) {` `{` Stmt `; return` *x* `+` *y* `; }`	BlockExpr	Apple extension
`void *` *labelptr* `= &&` *label* `;`	LabelLiteral	GNU extension; use with ComputedGotoStmt
%3d %s\n	FormatLiteral	
0xdbceffca	HexLiteral	
0167	OctalLiteral	
c	CharLiteral	
abcdefgh, *Lwide*	StringLiteral	

Continued on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
`new` Type `[` Expr `]`	NewArrayExpr	
`new` Type	NewExpr	
`delete [ ]` Expr `;`	DeleteArrayExpr	
`delete` Expr `;`	DeleteExpr	
`noexcept (` Expr `)`	NoExceptExpr	
Expr `=` Expr	AssignExpr	See also Initializer
Expr `+=` Expr	AssignAddExpr AssignPointerAddExpr	
Expr `/=` Expr	AssignDivExpr	
Expr `*=` Expr	AssignMulExpr	
Expr `%=` Expr	AssignRemExpr	
Expr `-=` Expr	AssignSubExpr AssignPointerSubExpr	
Expr `&=` Expr	AssignAndExpr	
Expr `<<=` Expr	AssignLShiftExpr	
Expr `|=` Expr	AssignOrExpr	
Expr `>>=` Expr	AssignRShiftExpr	
Expr `^=` Expr	AssignXorExpr	
Expr `+` Expr	AddExpr PointerAddExpr ImaginaryRealAddExpr RealImaginaryAddExpr	 C99 C99
Expr `/` Expr	DivExpr ImaginaryDivExpr	 C99
Expr `>?` Expr	MaxExpr	GNU extension
Expr `<?` Expr	MinExpr	GNU extension
Expr `*` Expr	MulExpr ImaginaryMulExpr	 C99
Expr `%` Expr	RemExpr	

Continued on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
Expr - Expr	SubExpr PointerDiffExpr PointerSubExpr ImaginaryRealSubExpr RealImaginarySubExpr	C99 C99
Expr & Expr	BitwiseAndExpr	
Expr | Expr	BitwiseOrExpr	
Expr ^ Expr	BitwiseXorExpr	
Expr << Expr	LShiftExpr	
Expr >> Expr	RShiftExpr	
Expr && Expr	LogicalAndExpr	
Expr || Expr	LogicalOrExpr	
Expr == Expr	EQExpr	
Expr != Expr	NEExpr	
Expr >= Expr	GEExpr	
Expr > Expr	GTExpr	
Expr <= Expr	LEExpr	
Expr < Expr	LTExpr	
Expr ? Expr : Expr	ConditionalExpr	
& Expr	AddressOfExpr	
* Expr	PointerDereferenceExpr	
Expr --	PostfixDecrExpr	
-- Expr	PrefixDecrExpr	
Expr ++	PostfixIncrExpr	
++ Expr	PrefixIncrExpr	
__imag (Expr)	ImaginaryPartExpr	GNU extension
__real (Expr)	RealPartExpr	GNU extension
- Expr	UnaryMinusExpr	
+ Expr	UnaryPlusExpr	
~ Expr	ComplementExpr ConjugationExpr	GNU extension
! Expr	NotExpr	
int *vect* __attribute__ ((vector_size (*16*))) = { *3* , *8* , *32* , *33* };	VectorFillOperation	GNU extension
sizeof (Expr)	SizeofExprOperator	

Continued on next page

Table 2 – continued from previous page

Example syntax	CodeQL class(es)	Remarks
`sizeof ( Type )`	SizeofTypeOperator	

`template < typename... T >`
`int count ( T &&... t )`
`{ return sizeof... ( t ); }`
	SizeofPackOperator	

`( { Stmt ; Expr } )`	StmtExpr	GNU/Clang extension
`this`	ThisExpr	
`throw ( Expr );`	ThrowExpr	
`throw;`	ReThrowExpr	
`typeid ( Expr )` `typeid ( Type )`	TypeidOperator	
`__uuidof ( Expr )`	UuidofOperator	Microsoft extension

Type classes

This table lists subclasses of Type representing C/C++ types.

Example syntax	CodeQL class	Remarks
`void`	VoidType	
`_Bool` or `bool`	BoolType	
`char16_t`	Char16Type	C11, C++11
`char32_t`	Char32Type	C11, C++11
`char`	PlainCharType	
`signed char`	SignedCharType	
`unsigned char`	UnsignedCharType	
`int`	IntType	
`long long`	LongLongType	
`long`	LongType	
`short`	ShortType	
`wchar_t`	WideCharType	
`nullptr_t`	NullPointerType	
`double`	DoubleType	
`long double`	LongDoubleType	
`float`	FloatType	
`auto`	AutoType	
`decltype ( Expr )`	Decltype	
Type `[` *n* `]`	ArrayType	
Type `( ^` *blockptr* `)` `(` Parameter `)`	BlockType	Apple extension
Type `( *` *funcptr* `)` `(` Parameter `)`	FunctionPointerType	
Type `( &` *funcref* `)` `(` Parameter `)`	FunctionReferenceType	
Type `__attribute__ ( ( vector_size (` *n* `) ) )`	GNUVectorType	
Type `*`	PointerType	
Type `&`	LValueReferenceType	
Type `&&`	RValueReferenceType	
Type `(` Class `*::` *membptr* `)` `(` Parameter `)`	PointerToMemberType	
`template < template < typename > class` *C* `>`	TemplateTemplateParameter	
`template < typename` *T* `>`	TemplateParameter	

Preprocessor classes

This table lists Preprocessor classes representing C/C++ preprocessing directives.

Example syntax	CodeQL class	Remarks
`#elif` *condition*	PreprocessorElif	
`#if` *condition*	PreprocessorIf	
`#ifdef` *macro*	PreprocessorIfdef	
`#ifndef` *macro*	PreprocessorIfndef	
`#else`	PreprocessorElse	
`#endif`	PreprocessorEndif	
`#line` *line_number file_name*	PreprocessorLine	
`#pragma` *pragma_property*	PreprocessorPragma	
`#undef` *macro*	PreprocessorUndef	
`#warning` *message*	PreprocessorWarning	
`#error` *message*	PreprocessorError	
`#include` *file_name*	Include	
`#import` *file_name*	Import	Apple/NeXT extension
`#include_next` *file_name*	IncludeNext	Apple/NeXT extension
`#define` *macro*	Macro	

3.2.3 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.3 Functions in C and C++

You can use CodeQL to explore functions in C and C++ code.

3.3.1 Overview

The standard CodeQL library for C and C++ represents functions using the `Function` class (see *CodeQL libraries for C and C++*).

The example queries in this topic explore some of the most useful library predicates for querying functions.

3.3.2 Finding all static functions

Using the member predicate `Function.isStatic()` we can list all the static functions in a database:

```
import cpp

from Function f
where f.isStatic()
select f, "This is a static function."
```

This query is very general, so there are probably too many results to be interesting for most nontrivial projects.

3.3.3 Finding functions that are not called

It might be more interesting to find functions that are not called, using the standard CodeQL `FunctionCall` class from the **abstract syntax tree** category (see *CodeQL libraries for C and C++*). The `FunctionCall` class can be used to identify places where a function is actually used, and it is related to `Function` through the `FunctionCall.getTarget()` predicate.

```
import cpp

from Function f
where not exists(FunctionCall fc | fc.getTarget() = f)
select f, "This function is never called."
```

See this in the query console on LGTM.com

The new query finds functions that are not the target of any `FunctionCall`in other words, functions that are never called. You may be surprised by how many results the query finds. However, if you examine the results, you can see that many of the functions it finds are used indirectly. To create a query that finds only unused functions, we need to refine the query and exclude other ways of using a function.

3.3.4 Excluding functions that are referenced with a function pointer

You can modify the query to remove functions where a function pointer is used to reference the function:

```
import cpp

from Function f
where not exists(FunctionCall fc | fc.getTarget() = f)
  and not exists(FunctionAccess fa | fa.getTarget() = f)
select f, "This function is never called, or referenced with a function pointer."
```

See this in the query console on LGTM.com

This query returns fewer results. However, if you examine the results then you can probably still find potential refinements.

For example, there is a more complicated LGTM query that finds unused static functions. To see the code for this query, click **Open in query console** at the top of the page.

> You can explore the definition of an element in the standard libraries and see what predicates are available. Use the keyboard **F3** button to open the definition of any element. Alternatively, hover over the element and click **Jump to definition** in the tooltip displayed. The library file is opened in a new tab with the definition highlighted.

3.3.5 Finding a specific function

This query uses `Function` and `FunctionCall` to find calls to the function `sprintf` that have a variable format stringwhich is potentially a security hazard.

```
import cpp

from FunctionCall fc
where fc.getTarget().getQualifiedName() = "sprintf"
```

(continues on next page)

(continued from previous page)

```
  and not fc.getArgument(1) instanceof StringLiteral
select fc, "sprintf called with variable format string."
```

See this in the query console on LGTM.com

This uses:

- `Declaration.getQualifiedName()` to identify calls to the specific function `sprintf`.
- `FunctionCall.getArgument(1)` to fetch the format string argument.

Note that we could have used `Declaration.getName()`, but `Declaration.getQualifiedName()` is a better choice because it includes the namespace. For example: `getName()` would return `vector` where `getQualifiedName` would return `std::vector`.

The LGTM version of this query is considerably more complicated, but if you look carefully you will find that its structure is the same. See Non-constant format string and click **Open in query console** at the top of the page.

3.3.6 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.4 Expressions, types, and statements in C and C++

You can use CodeQL to explore expressions, types, and statements in C and C++ code to find, for example, incorrect assignments.

3.4.1 Expressions and types in CodeQL

Each part of an expression in C becomes an instance of the `Expr` class. For example, the C code `x = x + 1` becomes an `AssignExpr`, an `AddExpr`, two instances of `VariableAccess` and a `Literal`. All of these CodeQL classes extend `Expr`.

Finding assignments to zero

In the following example we find instances of `AssignExpr` which assign the constant value zero:

```
import cpp

from AssignExpr e
where e.getRValue().getValue().toInt() = 0
select e, "Assigning the value 0 to something."
```

See this in the query console on LGTM.com

The `where` clause in this example gets the expression on the right side of the assignment, `getRValue()`, and compares it with zero. Notice that there are no checks to make sure that the right side of the assignment is an integer or that it has a value (that is, it is compile-time constant, rather than a variable). For expressions where either of these assumptions is wrong, the associated predicate simply does not return anything and the `where` clause will not produce a result. You could think of it as if there is an implicit `exists(e.getRValue().getValue().toInt())` at the beginning of this line.

It is also worth noting that the query above would find this C code:

```
yPtr = NULL;
```

This is because the database contains a representation of the code base after the preprocessor transforms have run. This means that any macro invocations, such as the `NULL` define used here, are expanded during the creation of the database. If you want to write queries about macros then there are some special library classes that have been designed specifically for this purpose (for example, the `Macro`, `MacroInvocation` classes and predicates like `Element.isInMacroExpansion()`). In this case, it is good that macros are expanded, but we do not want to find assignments to pointers. For more information, see Database generation on LGTM.com.

Finding assignments of 0 to an integer

We can make the query more specific by defining a condition for the left side of the expression. For example:

```
import cpp

from AssignExpr e
where e.getRValue().getValue().toInt() = 0
  and e.getLValue().getType().getUnspecifiedType() instanceof IntegralType
select e, "Assigning the value 0 to an integer."
```

See this in the query console on LGTM.com

This checks that the left side of the assignment has a type that is some kind of integer. Note the call to `Type.getUnspecifiedType()`. This resolves `typedef` types to their underlying types so that the query finds assignments like this one:

```
typedef int myInt;
myInt i;

i = 0;
```

3.4.2 Statements in CodeQL

We can refine the query further using statements. In this case we use the class `ForStmt`:

- `Stmt` - C/C++ statements
  - `Loop`
    - `WhileStmt`
    - `ForStmt`
    - `DoStmt`
  - `ConditionalStmt`

IfStmt

SwitchStmt

- TryStmt
- ExprStmt - expressions used as a statement; for example, an assignment
- Block - { } blocks containing more statements

Finding assignments of 0 in for loop initialization

We can restrict the previous query so that it only considers assignments inside `for` statements by adding the `ForStmt` class to the query. Then we want to compare the expression to `ForStmt.getInitialization()`:

```
import cpp

from AssignExpr e, ForStmt f
// the assignment is the for loop initialization
where e = f.getInitialization()
...
```

Unfortunately this would not quite work, because the loop initialization is actually a `Stmt` not an `Exprthe` `AssignExpr` class is wrapped in an `ExprStmt` class. Instead, we need to find the closest enclosing `Stmt` around the expression using `Expr.getEnclosingStmt()`:

```
import cpp

from AssignExpr e, ForStmt f
// the assignment is in the 'for' loop initialization statement
where e.getEnclosingStmt() = f.getInitialization()
  and e.getRValue().getValue().toInt() = 0
  and e.getLValue().getType().getUnspecifiedType() instanceof IntegralType
select e, "Assigning the value 0 to an integer, inside a for loop initialization."
```

See this in the query console on LGTM.com

Finding assignments of 0 within the loop body

We can find assignments inside the loop body using similar code with the predicate `Loop.getStmt()`:

```
import cpp

from AssignExpr e, ForStmt f
// the assignment is in the for loop body
where e.getEnclosingStmt().getParentStmt*() = f.getStmt()
  and e.getRValue().getValue().toInt() = 0
  and e.getLValue().getType().getUnderlyingType() instanceof IntegralType
select e, "Assigning the value 0 to an integer, inside a for loop body."
```

See this in the query console on LGTM.com

Note that we replaced `e.getEnclosingStmt()` with `e.getEnclosingStmt().getParentStmt*()`, to find an assignment expression that is deeply nested inside the loop body. The transitive closure modifier `*` here indicates

that `Stmt.getParentStmt()` may be followed zero or more times, rather than just once, giving us the statement, its parent statement, its parents parent statement etc.

3.4.3 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.5 Conversions and classes in C and C++

You can use the standard CodeQL libraries for C and C++ to detect when the type of an expression is changed.

3.5.1 Conversions

In C and C++, conversions change the type of an expression. They may be implicit conversions generated by the compiler, or explicit conversions requested by the user.

Lets take a look at the Conversion class in the standard library:

- `Expr`
  - `Conversion`
    - `Cast`
      - `CStyleCast`
      - `StaticCast`
      - `ConstCastReinterpretCast`
      - `DynamicCast`
    - `ArrayToPointerConversion`
    - `VirtualMemberToFunctionPointerConversion`

Exploring the subexpressions of an assignment

Let us consider the following C code:

```
typedef signed int myInt;
int main(int argc, char *argv[])
{
    unsigned int i;
    i = (myInt)1;
    return 0;
}
```

And this simple query:

```
import cpp

from AssignExpr a
select a, a.getLValue().getType(), a.getRValue().getType()
```

The query examines the code for assignments, and tells us the type of their left and right subexpressions. In the example C code above, there is just one assignment. Notably, this assignment has two conversions (of type `CStyleCast`) on the right side:

1. Explicit cast of the integer `1` to a `myInt`.
2. Implicit conversion generated by the compiler, in preparation for the assignment, converting that expression into an `unsigned int`.

The query actually reports the result:

```
... = ... | unsigned int | int
```

It is as though the conversions are not there! The reason for this is that `Conversion` expressions do not wrap the objects they convert; instead conversions are attached to expressions and can be accessed using `Expr.getConversion()`. The whole assignment in our example is seen by the standard library classes like this:

```
AssignExpr, i = (myInt)1
 VariableAccess, i
 Literal, 1
      CStyleCast, myInt (explicit)
           CStyleCast, unsigned int (implicit)
```

Accessing parts of the assignment:

- Left sideaccess value using `Assignment.getLValue()`.
- Right sideaccess value using `Assignment.getRValue()`.
- Conversions of the `Literal` on the right sideaccess both using calls to `Expr.getConversion()`. As a shortcut, you can use `Expr.GetFullyConverted()` to follow all the way to the resulting type, or `Expr.GetExplicitlyConverted()` to find the last explicit conversion from an expression.

Using these predicates we can refine our query so that it reports the results that we expected:

```
import cpp

from AssignExpr a
select a, a.getLValue().getExplicitlyConverted().getType(), a.getRValue().getExplicitlyConverted().
↪getType()
```

The result is now:

```
... = ... | unsigned int | myInt
```

We can refine the query further by adding `Type.getUnderlyingType()` to resolve the `typedef`:

```
import cpp

from AssignExpr a
select a, a.getLValue().getExplicitlyConverted().getType().getUnderlyingType(), a.getRValue().
↪getExplicitlyConverted().getType().getUnderlyingType()
```

The result is now:

```
... = ... | unsigned int | signed int
```

If you simply wanted to get the values of all assignments in expressions, regardless of position, you could replace `Assignment.getLValue()` and `Assignment.getRValue()` with `Operation.getAnOperand()`:

```
import cpp

from AssignExpr a
select a, a.getAnOperand().getExplicitlyConverted().getType()
```

Unlike the earlier versions of the query, this query would return each side of the expression as a separate result:

```
... = ... | unsigned int
... = ... | myInt
```

Note

In general, predicates named `getAXxx` exploit the ability to return multiple results (multiple instances of `Xxx`) whereas plain `getXxx` predicates usually return at most one specific instance of `Xxx`.

3.5.2 Classes

Next were going to look at C++ classes, using the following CodeQL classes:

- `Type`
  - `UserType`includes classes, typedefs, and enums

    `Class`a class or struct

    - `Struct`a struct, which is treated as a subtype of `Class`
    - `TemplateClass`a C++ class template

Finding derived classes

We want to create a query that checks for destructors that should be `virtual`. Specifically, when a class and a class derived from it both have destructors, the base class destructor should generally be virtual. This ensures that the derived class destructor is always invoked. In the CodeQL library, `Destructor` is a subtype of `MemberFunction`:

- `Function`
  - `MemberFunction`

    `Constructor`

```
Destructor
```

Our starting point for the query is pairs of a base class and a derived class, connected using `Class.getABaseClass()`:

```
import cpp

from Class base, Class derived
where derived.getABaseClass+() = base
select base, derived, "The second class is derived from the first."
```

See this in the query console on LGTM.com

Note that the transitive closure symbol `+` indicates that `Class.getABaseClass()` may be followed one or more times, rather than only accepting a direct base class.

A lot of the results are uninteresting template parameters. You can remove those results by updating the `where` clause as follows:

```
where derived.getABaseClass+() = base
  and not exists(base.getATemplateArgument())
  and not exists(derived.getATemplateArgument())
```

See this in the query console on LGTM.com

Finding derived classes with destructors

Now we can extend the query to find derived classes with destructors, using the `Class.getDestructor()` predicate:

```
import cpp

from Class base, Class derived, Destructor d1, Destructor d2
where derived.getABaseClass+() = base
  and not exists(base.getATemplateArgument())
  and not exists(derived.getATemplateArgument())
  and d1 = base.getDestructor()
  and d2 = derived.getDestructor()
select base, derived, "The second class is derived from the first, and both have a destructor."
```

See this in the query console on LGTM.com

Notice that getting the destructor implicitly asserts that one exists. As a result, this version of the query returns fewer results than before.

Finding base classes where the destructor is not virtual

Our last change is to use `Function.isVirtual()` to find cases where the base destructor is not virtual:

```
import cpp

from Class base, Destructor d1, Class derived, Destructor d2
where derived.getABaseClass+() = base
```

(continues on next page)

(continued from previous page)

```
  and d1.getDeclaringType() = base
  and d2.getDeclaringType() = derived
  and not d1.isVirtual()
select d1, "This destructor should probably be virtual."
```

See this in the query console on LGTM.com

That completes the query.

There is a similar built-in query on LGTM.com that finds classes in a C/C++ project with virtual functions but no virtual destructor. You can take a look at the code for this query by clicking **Open in query console** at the top of that page.

3.5.3 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.6 Analyzing data flow in C and C++

You can use data flow analysis to track the flow of potentially malicious or insecure data that can cause vulnerabilities in your codebase.

3.6.1 About data flow

Data flow analysis computes 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. In CodeQL, you can model both local data flow and global data flow. For a more general introduction to modeling data flow, see *About data flow analysis*.

3.6.2 Local data flow

Local data flow is data flow within a single function. Local data flow is usually easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. `Nodes` are divided into expression nodes (`ExprNode`) and parameter nodes (`ParameterNode`). It is possible to map between data flow nodes and expressions/parameters using the member predicates `asExpr` and `asParameter`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the parameter corresponding to this node, if any. */
  Parameter asParameter() { ... }

  ...
}
```

or using the predicates `exprNode` and `parameterNode`:

```
/**
 * Gets the node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }

/**
 * Gets the node corresponding to the value of parameter `p` at function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. The predicate can be applied recursively (using the `+` and `*` operators), or through the predefined recursive predicate `localFlow`, which is equivalent to `localFlowStep*`.

For example, finding flow from a parameter `source` to an expression `sink` in zero or more local steps can be achieved as follows:

```
DataFlow::localFlow(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
int i = tainted_user_input();
some_big_struct *array = malloc(i * sizeof(some_big_struct));
```

In this case, the argument to `malloc` is tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint propagation edge from the node `nodeFrom` to the node `nodeTo`. The predicate can be applied recursively (using the `+` and `*` operators), or through the predefined recursive predicate `localTaint`, which is equivalent to `localTaintStep*`.

For example, finding taint propagation from a parameter `source` to an expression `sink` in zero or more local steps can be achieved as follows:

```
TaintTracking::localTaint(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Examples

The following query finds the filename passed to `fopen`.

```
import cpp

from Function fopen, FunctionCall fc
where fopen.hasQualifiedName("fopen")
  and fc.getTarget() = fopen
select fc.getArgument(0)
```

Unfortunately, this will only give the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import cpp
import semmle.code.cpp.dataflow.DataFlow

from Function fopen, FunctionCall fc, Expr src
where fopen.hasQualifiedName("fopen")
  and fc.getTarget() = fopen
  and DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(fc.getArgument(0)))
select src
```

Then we can vary the source, for example an access to a public parameter. The following query finds where a public parameter is used to open a file:

```
import cpp
import semmle.code.cpp.dataflow.DataFlow

from Function fopen, FunctionCall fc, Parameter p
where fopen.hasQualifiedName("fopen")
  and fc.getTarget() = fopen
  and DataFlow::localFlow(DataFlow::parameterNode(p), DataFlow::exprNode(fc.getArgument(0)))
select p
```

The following example finds calls to formatting functions where the format string is not hard-coded.

```
import semmle.code.cpp.dataflow.DataFlow
import semmle.code.cpp.commons.Printf

from FormattingFunction format, FunctionCall call, Expr formatString
where call.getTarget() = format
  and call.getArgument(format.getFormatParameterIndex()) = formatString
  and not exists(DataFlow::Node source, DataFlow::Node sink |
    DataFlow::localFlow(source, sink) and
    source.asExpr() instanceof StringLiteral and
    sink.asExpr() = formatString
  )
select call, "Argument to " + format.getQualifiedName() + " isn't hard-coded."
```

Exercises

Exercise 1: Write a query that finds all hard-coded strings used to create a `host_ent` via `gethostbyname`, using local data flow. (*Answer*)

3.6.3 Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see Creating path queries.

Using global data flow

The global data flow library is used by extending the class `DataFlow::Configuration` as follows:

```
import semmle.code.cpp.dataflow.DataFlow

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

The following predicates are defined in the configuration:

- `isSource`defines where data may flow from
- `isSink`defines where data may flow to
- `isBarrier`optional, restricts the data flow
- `isBarrierGuard`optional, restricts the data flow
- `isAdditionalFlowStep`optional, adds additional flow steps

The characteristic predicate `MyDataFlowConfiguration()` defines the name of the configuration, so `"MyDataFlowConfiguration"` should be replaced by the name of your class.

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Data flow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow as local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. The global taint tracking library is used by extending the class `TaintTracking::Configuration` as follows:

```
import semmle.code.cpp.dataflow.TaintTracking

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

The following predicates are defined in the configuration:

- `isSource`defines where taint may flow from
- `isSink`defines where taint may flow to
- `isSanitizer`optional, restricts the taint flow
- `isSanitizerGuard`optional, restricts the taint flow
- `isAdditionalTaintStep`optional, adds additional taint steps

Similar to global data flow, the characteristic predicate `MyTaintTrackingConfiguration()` defines the unique name of the configuration, so `"MyTaintTrackingConfiguration"` should be replaced by the name of your class.

The taint tracking analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Examples

The following data flow configuration tracks data flow from environment variables to opening files in a Unix-like environment:

```
import semmle.code.cpp.dataflow.DataFlow

class EnvironmentToFileConfiguration extends DataFlow::Configuration {
  EnvironmentToFileConfiguration() { this = "EnvironmentToFileConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    exists (Function getenv |
      source.asExpr().(FunctionCall).getTarget() = getenv and
      getenv.hasQualifiedName("getenv")
    )
  }
```

(continues on next page)

(continued from previous page)

```
  override predicate isSink(DataFlow::Node sink) {
    exists (FunctionCall fc |
      sink.asExpr() = fc.getArgument(0) and
      fc.getTarget().hasQualifiedName("fopen")
    )
  }
}

from Expr getenv, Expr fopen, EnvironmentToFileConfiguration config
where config.hasFlow(DataFlow::exprNode(getenv), DataFlow::exprNode(fopen))
select fopen, "This 'fopen' uses data from $@.",
  getenv, "call to 'getenv'"
```

The following taint-tracking configuration tracks data from a call to `ntohl` to an array index operation. It uses the `Guards` library to recognize expressions that have been bounds-checked, and defines `isSanitizer` to prevent taint from propagating through them. It also uses `isAdditionalTaintStep` to add flow from loop bounds to loop indexes.

```
import cpp
import semmle.code.cpp.controlflow.Guards
import semmle.code.cpp.dataflow.TaintTracking

class NetworkToBufferSizeConfiguration extends TaintTracking::Configuration {
  NetworkToBufferSizeConfiguration() { this = "NetworkToBufferSizeConfiguration" }

  override predicate isSource(DataFlow::Node node) {
    node.asExpr().(FunctionCall).getTarget().hasGlobalName("ntohl")
  }

  override predicate isSink(DataFlow::Node node) {
    exists(ArrayExpr ae | node.asExpr() = ae.getArrayOffset())
  }

  override predicate isAdditionalTaintStep(DataFlow::Node pred, DataFlow::Node succ) {
    exists(Loop loop, LoopCounter lc |
      loop = lc.getALoop() and
      loop.getControllingExpr().(RelationalOperation).getGreaterOperand() = pred.asExpr() |
      succ.asExpr() = lc.getVariableAccessInLoop(loop)
    )
  }

  override predicate isSanitizer(DataFlow::Node node) {
    exists(GuardCondition gc, Variable v |
      gc.getAChild*() = v.getAnAccess() and
      node.asExpr() = v.getAnAccess() and
      gc.controls(node.asExpr().getBasicBlock(), _)
    )
  }
}

from DataFlow::Node ntohl, DataFlow::Node offset, NetworkToBufferSizeConfiguration conf
```

(continues on next page)

(continued from previous page)

```
where conf.hasFlow(ntohl, offset)
select offset, "This array offset may be influenced by $@.", ntohl,
  "converted data from the network"
```

Exercises

Exercise 2: Write a query that finds all hard-coded strings used to create a `host_ent` via `gethostbyname`, using global data flow. (*Answer*)

Exercise 3: Write a class that represents flow sources from `getenv`. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query which finds all global data flows from `getenv` to `gethostbyname`. (*Answer*)

3.6.4 Answers

Exercise 1

```
import semmle.code.cpp.dataflow.DataFlow

from StringLiteral sl, FunctionCall fc
where fc.getTarget().hasName("gethostbyname")
  and DataFlow::localFlow(DataFlow::exprNode(sl), DataFlow::exprNode(fc.getArgument(0)))
select sl, fc
```

Exercise 2

```
import semmle.code.cpp.dataflow.DataFlow

class LiteralToGethostbynameConfiguration extends DataFlow::Configuration {
  LiteralToGethostbynameConfiguration() {
    this = "LiteralToGethostbynameConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists (FunctionCall fc |
      sink.asExpr() = fc.getArgument(0) and
      fc.getTarget().hasName("gethostbyname"))
  }
}

from StringLiteral sl, FunctionCall fc, LiteralToGethostbynameConfiguration cfg
where cfg.hasFlow(DataFlow::exprNode(sl), DataFlow::exprNode(fc.getArgument(0)))
select sl, fc
```

Exercise 3

```
import cpp

class GetenvSource extends FunctionCall {
  GetenvSource() {
    this.getTarget().hasQualifiedName("getenv")
  }
}
```

Exercise 4

```
import semmle.code.cpp.dataflow.DataFlow

class GetenvSource extends DataFlow::Node {
  GetenvSource() {
    this.asExpr().(FunctionCall).getTarget().hasQualifiedName("getenv")
  }
}

class GetenvToGethostbynameConfiguration extends DataFlow::Configuration {
  GetenvToGethostbynameConfiguration() {
    this = "GetenvToGethostbynameConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof GetenvSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists (FunctionCall fc |
      sink.asExpr() = fc.getArgument(0) and
      fc.getTarget().hasName("gethostbyname"))
  }
}

from DataFlow::Node getenv, FunctionCall fc, GetenvToGethostbynameConfiguration cfg
where cfg.hasFlow(getenv, DataFlow::exprNode(fc.getArgument(0)))
select getenv.asExpr(), fc
```

3.6.5 Further reading

- Exploring data flow with path queries
- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.7 Refining a query to account for edge cases

You can improve the results generated by a CodeQL query by adding conditions to remove false positive results caused by common edge cases.

3.7.1 Overview

This topic describes how a C++ query was developed. The example introduces recursive predicates and demonstrates the typical workflow used to refine a query. For a full overview of the topics available for learning to write queries for C/C++ code, see *CodeQL for C and C++*.

3.7.2 Finding every private field and checking for initialization

Writing a query to check if a constructor initializes all private fields seems like a simple problem, but there are several edge cases to account for.

3.7.3 Basic query

We can start by looking at every private field in a class and checking that every constructor in that class initializes them. Once you are familiar with the library for C++ this is not too hard to do.

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    and not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

1. `f.getDeclaringType() = c.getDeclaringType()` asserts that the field and constructor are both part of the same class.
2. `f.isPrivate()` checks if the field is private.
3. `not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)` checks that there is no assignment to the field in the constructor.

This code looks fairly complete, but when you test it on a project, there are several results that contain examples that we have overlooked.

3.7.4 Refinement 1excluding fields initialized by lists

You may see that the results contain fields that are initialized by constructor initialization lists, instead of by assignment statements. For example, the following class:

```
class BoxedInt {
public:
  BoxedInt(int value) : m_value(value) {}

private:
  int m_value;
};
```

These can be excluded by adding an extra condition to check for this special constructor-only form of assignment.

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    and not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.getEnclosingFunction() = c)
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

3.7.5 Refinement 2excluding fields initialized by external libraries

When you test the revised query, you may discover that fields from classes in external libraries are over-reported. This is often because a header file declares a constructor that is defined in a source file that is not analyzed (external libraries are often excluded from analysis). When the source code is analyzed, the CodeQL database is populated with a `Constructor` entry with no body. This `constructor` therefore contains no assignments and consequently the query reports that any fields initialized by the constructor are uninitialized. There is no particular reason to be suspicious of these cases, and we can exclude them from the results by defining a condition to exclude constructors that have no body:

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    and not exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.getEnclosingFunction() = c)
    // ignore cases where the constructor source code is not available
    and exists(c.getBlock())
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

This is a reasonably precise querymost of the results that it reports are interesting. However, you could make further refinements.

3.7.6 Refinement 3excluding fields initialized indirectly

You may also wish to consider methods called by constructors that assign to the fields, or even to the methods called by those methods. As a concrete example of this, consider the following class.

```
class BoxedInt {
public:
  BoxedInt(int value) {
    setValue(value);
  }

  void setValue(int value) {
    m_value = value;
  }

private:
```

(continues on next page)

(continued from previous page)

```
  int m_value;
};
```

This case can be excluded by creating a recursive predicate. The recursive predicate is given a function and a field, then checks whether the function assigns to the field. The predicate runs itself on all the functions called by the function that it has been given. By passing the constructor to this predicate, we can check for assignments of a field in all functions called by the constructor, and then do the same for all functions called by those functions all the way down the tree of function calls. For more information, see Recursion in the QL language reference.

```
import cpp

predicate getSubAssignment(Function c, Field f){
    exists(Assignment a | a = f.getAnAssignment() and a.getEnclosingFunction() = c)
    or exists(Function fun | c.calls(fun) and getSubAssignment(fun, f))
}
from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.getEnclosingFunction() = c)
    // check for initializations performed indirectly by methods called
    // as a result of the constructor being called
    and not getSubAssignment(c, f)
    // ignore cases where the constructor source code is not available
    and exists(c.getBlock())
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

3.7.7 Refinement 4simplifying the query

Finally we can simplify the query by using the transitive closure operator. In this final version of the query, `c.calls*(fun)` resolves to the set of all functions that are `c` itself, are called by `c`, are called by a function that is called by `c`, and so on. This eliminates the need to make a new predicate all together. For more information, see Transitive closures in the QL language reference.

```
import cpp

from Constructor c, Field f
where f.getDeclaringType() = c.getDeclaringType() and f.isPrivate()
    // check for constructor initialization lists as well
    and not exists(ConstructorFieldInit i | i.getTarget() = f and i.getEnclosingFunction() = c)
    // check for initializations performed indirectly by methods called
    // as a result of the constructor being called
    and not exists(Function fun, Assignment a |
      c.calls*(fun) and a = f.getAnAssignment() and a.getEnclosingFunction() = fun)
    // ignore cases where the constructor source code is not available
    and exists(c.getBlock())
select c, "Constructor does not initialize fields $@.", f, f.getName()
```

See this in the query console on LGTM.com

3.7.8 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.8 Detecting a potential buffer overflow

You can use CodeQL to detect potential buffer overflows by checking for allocations equal to `strlen` in C and C++. This topic describes how a C/C++ query for detecting a potential buffer overflow was developed.

3.8.1 Problemdetecting memory allocation that omits space for a null termination character

The objective of this query is to detect C/C++ code which allocates an amount of memory equal to the length of a null terminated string, without adding +1 to make room for a null termination character. For example the following code demonstrates this mistake, and results in a buffer overflow:

```
void processString(const char *input)
{
    char *buffer = malloc(strlen(input));

    strcpy(buffer, input);

    ...
}
```

3.8.2 Basic query

Before you can write a query you need to decide what entities to search for and then define how to identify them.

Defining the entities of interest

You could approach this problem either by searching for code similar to the call to `malloc` in line 3 or the call to `strcpy` in line 5 (see example above). For our basic query, we start with a simple assumption: any call to `malloc` with only a `strlen` to define the memory size is likely to cause an error when the memory is populated.

Calls to `strlen` can be identified using the library StrlenCall class, but we need to define a new class to identify calls to `malloc`. Both the library class and the new class need to extend the standard class `FunctionCall`, with the added restriction of the function name that they apply to:

```
import cpp

class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
}
```

Note

You could easily extend this class to include similar functions such as `realloc`, or your own custom allocator. With a little effort they could even include C++ `new` expressions (to do this, `MallocCall` would need to extend a common superclass of both `FunctionCall` and `NewExpr`, such as `Expr`).

Finding the `strlen(string)` pattern

Before we start to write our query, theres one remaining task. We need to modify our new `MallocCall` class, so it returns an expression for the size of the allocation. Currently this will be the first argument to the `malloc` call, `FunctionCall.getArgument(0)`, but converting this into a predicate makes it more flexible for future refinements.

```
class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
    Expr getAllocatedSize() {
        result = this.getArgument(0)
    }
}
```

Defining the basic query

Now we can write a query using these classes:

```
import cpp

class MallocCall extends FunctionCall
{
    MallocCall() { this.getTarget().hasGlobalName("malloc") }
    Expr getAllocatedSize() {
        result = this.getArgument(0)
    }
}

from MallocCall malloc
where malloc.getAllocatedSize() instanceof StrlenCall
select malloc, "This allocation does not include space to null-terminate the string."
```

Note that there is no need to check whether anything is added to the `strlen` expression, as it would be in the corrected C code `malloc(strlen(string) + 1)`. This is because the corrected code would in fact be an `AddExpr` containing a `StrlenCall`, not an instance of `StrlenCall` itself. A side-effect of this approach is that we omit certain unlikely patterns such as `malloc(strlen(string) + 0)`. In practice we can always come back and extend our query to cover this pattern if it is a concern.

Tip

For some projects, this query may not return any results. Possibly the project you are querying does not have any problems of this kind, but it is also important to make sure the query itself is working properly. One solution is to set up a test project with examples of correct and incorrect code to run the query against (the C code at the very top of this page makes a good starting point). Another approach is to test each part of the query individually to make sure everything is working.

When you have defined the basic query then you can refine the query to include further coding patterns or to exclude false positives:

3.8.3 Improving the query using the SSA library

The `SSA` library represents variables in static single assignment (SSA) form. In this form, each variable is assigned exactly once and every variable is defined before it is used. The use of SSA variables simplifies queries considerably as much of the local data flow analysis has been done for us. For more information, see Static single assignment on Wikipedia.

Including examples where the string size is stored before use

The query above works for simple cases, but does not identify a common coding pattern where `strlen(string)` is stored in a variable before being passed to `malloc`, as in the following example:

```
int len = strlen(input);
buffer = malloc(len);
```

To identify this case we can use the standard library `SSA.qll` (imported as `semmle.code.cpp.controlflow.SSA`).

This library helps us identify where values assigned to local variables may subsequently be used.

For example, consider the following code:

```
void myFunction(bool condition)
{
    const char* x = "alpha"; // definition #1 of x

    printf("x = %s\n", x); // use #1 of x

    if (condition)
    {
        x = "beta"; // definition #2 of x
    } else {
        x = "gamma"; // definition #3 of x
    }

    printf("x = %s\n", x); // use #2 of x
}
```

If we run the following query on the code, we get three results:

```
import cpp
import semmle.code.cpp.controlflow.SSA

from Variable var, Expr defExpr, Expr use
where exists(SsaDefinition ssaDef |
    defExpr = ssaDef.getAnUltimateDefiningValue(var)
    and use = ssaDef.getAUse(var))
select var, defExpr.getLocation().getStartLine() as dline, use.getLocation().getStartLine() as␣
↪uline
```

Results:

var	dline	uline
x	3	5
x	9	14
x	11	14

It is often useful to also display the defining expression `defExpr`, if there is one. For example we might adjust the query above as follows:

```
import cpp
import semmle.code.cpp.controlflow.SSA

from Variable var, Expr defExpr, Expr use
where exists(SsaDefinition ssaDef |
    defExpr = ssaDef.getAnUltimateDefiningValue(var)
    and use = ssaDef.getAUse(var))
select var, defExpr.getLocation().getStartLine() as dline, use.getLocation().getStartLine() as␣
↪uline, defExpr
```

Now we can see the assigned expression in our results:

var	dline	uline	defExpr
x	3	5	alpha
x	9	14	beta
x	11	14	gamma

Extending the query to include allocations passed via a variable

Using our experiments above we can expand our simple implementation of `MallocCall.getAllocatedSize()`. With the following refinement, if the argument is an access to a variable, `getAllocatedSize()` returns a value assigned to that variable instead of the variable access itself:

```
Expr getAllocatedSize() {
    if this.getArgument(0) instanceof VariableAccess then
        exists(LocalScopeVariable v, SsaDefinition ssaDef |
                result = ssaDef.getAnUltimateDefiningValue(v)
                and this.getArgument(0) = ssaDef.getAUse(v))
    else
        result = this.getArgument(0)
}
```

The completed query will now identify cases where the result of `strlen` is stored in a local variable before it is used in a call to `malloc`. Here is the query in full:

```
import cpp

class MallocCall extends FunctionCall
{
```

(continues on next page)

(continued from previous page)

```
    MallocCall() { this.getTarget().hasGlobalName("malloc") }

    Expr getAllocatedSize() {
        if this.getArgument(0) instanceof VariableAccess then
            exists(LocalScopeVariable v, SsaDefinition ssaDef |
                result = ssaDef.getAnUltimateDefiningValue(v)
                and this.getArgument(0) = ssaDef.getAUse(v))
        else
            result = this.getArgument(0)
    }
}

from MallocCall malloc
where malloc.getAllocatedSize() instanceof StrlenCall
select malloc, "This allocation does not include space to null-terminate the string."
```

3.8.4 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.9 Using the guards library in C and C++

You can use the CodeQL guards library to identify conditional expressions that control the execution of other parts of a program in C and C++ codebases.

3.9.1 About the guards library

The guards library (defined in `semmle.code.cpp.controlflow.Guards`) provides a class GuardCondition representing Boolean values that are used to make control flow decisions. A `GuardCondition` is considered to guard a basic block if the block can only be reached if the `GuardCondition` is evaluated a certain way. For instance, in the following code, `x < 10` is a `GuardCondition`, and it guards all the code before the return statement.

```
if(x < 10) {
  f(x);
} else if (x < 20) {
  g(x);
} else {
  h(x);
}
return 0;
```

3.9.2 The `controls` predicate

The `controls` predicate helps determine which blocks are only run when the `GuardCondition` evaluates a certain way. `guard.controls(block, testIsTrue)` holds if `block` is only entered if the value of this condition is `testIsTrue`.

In the following code sample, the call to `isValid` controls the calls to `performAction` and `logFailure` but not the return statement.

```
if(isValid(accessToken)) {
  performAction();
  succeeded = 1;
} else {
  logFailure();
  succeeded = 0;
}
return succeeded;
```

In the following code sample, the call to `isValid` controls the body of the `if` statement, and also the code after the `if`.

```
if(!isValid(accessToken)) {
  logFailure();
  return 0;
}
performAction();
return succeeded;
```

3.9.3 The `ensuresEq` and `ensuresLt` predicates

The `ensuresEq` and `ensuresLt` predicates are the main way of determining what, if any, guarantees the `GuardCondition` provides for a given basic block.

The `ensuresEq` predicate

When `ensuresEq(left, right, k, block, true)` holds, then `block` is only executed if `left` was equal to `right + k` at their last evaluation. When `ensuresEq(left, right, k, block, false)` holds, then `block` is only executed if `left` was not equal to `right + k` at their last evaluation.

The `ensuresLt` predicate

When `ensuresLt(left, right, k, block, true)` holds, then `block` is only executed if `left` was strictly less than `right + k` at their last evaluation. When `ensuresLt(left, right, k, block, false)` holds, then `block` is only executed if `left` was greater than or equal to `right + k` at their last evaluation.

In the following code sample, the comparison on the first line ensures that `index` is less than `size` in the then block, and that `index` is greater than or equal to `size` in the else block.

```
if(index < size) {
  ret = array[index];
} else {
  ret = nullptr
```

(continues on next page)

(continued from previous page)

```
}
return ret;
```

3.9.4 The `comparesEq` and `comparesLt` predicates

The `comparesEq` and `comparesLt` predicates help determine if the `GuardCondition` evaluates to true.

The `comparesEq` predicate

`comparesEq(left, right, k, true, testIsTrue)` holds if `left` equals `right + k` when the expression evaluates to `testIsTrue`.

The `comparesLt` predicate

`comparesLt(left, right, k, isLessThan, testIsTrue)` holds if `left < right + k` evaluates to `isLessThan` when the expression evaluates to `testIsTrue`.

3.9.5 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.10 Using range analysis for C and C++

You can use range analysis to determine the upper or lower bounds on an expression, or whether an expression could potentially over or underflow.

3.10.1 About the range analysis library

The range analysis library (defined in `semmle.code.cpp.rangeanalysis.SimpleRangeAnalysis`) provides a set of predicates for determining constant upper and lower bounds on expressions, as well as recognizing integer overflows. For performance, the library performs automatic widening and therefore may not provide the tightest possible bounds.

3.10.2 Bounds predicates

The `upperBound` and `lowerBound` predicates provide constant bounds on expressions. No conversions of the argument are included in the bound. In the common case that your query needs to take conversions into account, call them on the converted form, such as `upperBound(expr.getFullyConverted())`.

3.10.3 Overflow predicates

`exprMightOverflow` and related predicates hold if the relevant expression might overflow, as determined by the range analysis library. The `convertedExprMightOverflow` family of predicates will take conversions into account.

3.10.4 Example

This query uses `upperBound` to determine whether the result of `snprintf` is checked when used in a loop.

```
from FunctionCall call, DataFlow::Node source, DataFlow::Node sink, Expr convSink
where
  // the call is an snprintf with a string format argument
  call.getTarget().getName() = "snprintf" and
  call.getArgument(2).getValue().regexpMatch(".*%s.*") and

  // the result of the call influences its size argument in later iterations
  TaintTracking::localTaint(source, sink) and
  source.asExpr() = call and
  sink.asExpr() = call.getArgument(1) and

  // there is no fixed bound on the snprintf's size argument
  upperBound(convSink) = typeUpperBound(convSink.getType().getUnspecifiedType()) and
  convSink = call.getArgument(1).getFullyConverted()

select call, upperBound(call.getArgument(1).getFullyConverted())
```

3.10.5 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools

3.11 Hash consing and value numbering

You can use specialized CodeQL libraries to recognize expressions that are syntactically identical or compute the same value at runtime in C and C++ codebases.

3.11.1 About the hash consing and value numbering libraries

In C and C++ databases, each node in the abstract syntax tree is represented by a separate object. This allows both analysis and results display to refer to specific appearances of a piece of syntax. However, it is frequently useful to determine whether two expressions are equivalent, either syntactically or semantically.

The hash consing library (defined in `semmle.code.cpp.valuenumbering.HashCons`) provides a mechanism for identifying expressions that have the same syntactic structure. The global value numbering library (defined in

`semmle.code.cpp.valuenumbering.GlobalValueNumbering`) provides a mechanism for identifying expressions that compute the same value at runtime. Both libraries partition the expressions in each function into equivalence classes represented by objects. Each `HashCons` object represents a set of expressions with identical parse trees, while `GVN` objects represent sets of expressions that will always compute the same value. For more information, see Hash consing and Value numbering on Wikipedia.

3.11.2 Example C code

In the following C program, `x + y` and `x + z` will be assigned the same value number but different hash conses.

```
int x = 1;
int y = 2;
int z = y;
if(x + y == x + z) {
  ...
}
```

However, in the next example, the uses of `x + y` will have different value numbers but the same hash cons.

```
int x = 1;
int y = 2;
if(x + y) {
  ...
}

x = 2;

if(x + y) {
  ...
}
```

3.11.3 Value numbering

The value numbering library (defined in `semmle.code.cpp.valuenumbering.GlobalValueNumbering`) provides a mechanism for identifying expressions that compute the same value at runtime. Value numbering is useful when your primary concern is with the values being produced or the eventual machine code being run. For instance, value numbering might be used to determine whether a check is being done against the same value as the operation it is guarding.

The value numbering API

The value numbering library exposes its interface primarily through the `GVN` class. Each instance of `GVN` represents a set of expressions that will always evaluate to the same value. To get an expression in the set represented by a particular `GVN`, use the `getAnExpr()` member predicate.

To get the `GVN` of an `Expr`, use the `globalValueNumber` predicate.

Note: While the `GVN` class has `toString` and `getLocation` methods, these are only provided as debugging aids. They give the `toString` and `getLocation` of an arbitrary `Expr` within the set.

Why not a predicate?

The obvious interface for this library would be a predicate `equivalent(Expr e1, Expr e2)`. However, this predicate would be very large, with a quadratic number of rows for each set of equivalent expressions. By using a class as an intermediate step, the number of rows can be kept linear, and therefore can be cached.

Example query

This query uses the `GVN` class to identify calls to `strncpy` where the size argument is derived from the source rather than the destination

```
from FunctionCall strncpy, FunctionCall strlen
where
  strncpy.getTarget().hasGlobalName("strncpy") and
  strlen.getTarget().hasGlobalName("strlen") and
  globalValueNumber(strncpy.getArgument(1)) = globalValueNumber(strlen.getArgument(0)) and
  strlen = strncpy.getArgument(2)
select ci, "This call to strncpy is bounded by the size of the source rather than the destination"
```

3.11.4 Hash consing

The hash consing library (defined in `semmle.code.cpp.valuenumbering.HashCons`) provides a mechanism for identifying expressions that have the same syntactic structure. Hash consing is useful when your primary concern is with the text of the code. For instance, hash consing might be used to detect duplicate code within a function.

The hash consing API

The hash consing library exposes its interface primarily through the `HashCons` class. Each instance of `HashCons` represents a set of expressions within one function that have the same syntax (including referring to the same variables). To get an expression in the set represented by a particular `HashCons`, use the `getAnExpr()` member predicate.

Note: While the `HashCons` class has `toString` and `getLocation` methods, these are only provided as debugging aids. They give the `toString` and `getLocation` of an arbitrary `Expr` within the set.

To get the `HashCons` of an `Expr`, use the `hashCons` predicate.

Example query

```
import cpp
import semmle.code.cpp.valuenumbering.HashCons

from IfStmt outer, IfStmt inner
where
  outer.getElse+() = inner and
  hashCons(outer.getCondition()) = hashCons(inner.getCondition())
select inner.getCondition(), "The condition of this if statement duplicates the condition of $@",
  outer.getCondition(), "an enclosing if statement"
```

3.11.5 Further reading

- CodeQL queries for C and C++
- Example queries for C and C++
- CodeQL library reference for C and C++
- QL language reference
- CodeQL tools
- *Basic query for C and C++ code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for C and C++*: When analyzing C or C++ code, you can use the large collection of classes in the CodeQL library for C and C++.
- *Functions in C and C++*: You can use CodeQL to explore functions in C and C++ code.
- *Expressions, types, and statements in C and C++*: You can use CodeQL to explore expressions, types, and statements in C and C++ code to find, for example, incorrect assignments.
- *Conversions and classes in C and C++*: You can use the standard CodeQL libraries for C and C++ to detect when the type of an expression is changed.
- *Analyzing data flow in C and C++*: You can use data flow analysis to track the flow of potentially malicious or insecure data that can cause vulnerabilities in your codebase.
- *Refining a query to account for edge cases*: You can improve the results generated by a CodeQL query by adding conditions to remove false positive results caused by common edge cases.
- *Detecting a potential buffer overflow*: You can use CodeQL to detect potential buffer overflows by checking for allocations equal to `strlen` in C and C++.
- *Using the guards library in C and C++*: You can use the CodeQL guards library to identify conditional expressions that control the execution of other parts of a program in C and C++ codebases.
- *Using range analysis for C and C++*: You can use range analysis to determine the upper or lower bounds on an expression, or whether an expression could potentially over or underflow.
- *Hash consing and value numbering*: You can use specialized CodeQL libraries to recognize expressions that are syntactically identical or compute the same value at runtime in C and C++ codebases.

CHAPTER

FOUR

CODEQL FOR C#

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from C# codebases.

4.1 Basic query for C# code

Learn to write and run a simple CodeQL query using LGTM.

4.1.1 About the query

The query were going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they have an empty then branch. For example, code such as:

```
if (error) { }
```

4.1.2 Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **C#** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import csharp

from IfStmt ifstmt, BlockStmt block
where ifstmt.getThen() = block and
  block.isEmpty()
select ifstmt, "This 'if' statement is redundant."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis () at the bottom of the table indicates that the entire list is not displayedclick it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import csharp`	Imports the standard CodeQL libraries for C#.	Every query begins with one or more `import` statements.
`from IfStmt ifstmt, BlockStmt block`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `IfStmt` variable for `if` statements • a `BlockStmt` variable for the then block
`where ifstmt.getThen() = block and block.isEmpty()`	Defines a condition on the variables.	`ifstmt.getThen() = block` relates the two variables. The block must be the `then` branch of the `if` statement. `block.isEmpty()` states that the block must be empty (that is, it contains no statements).
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

4.1.3 Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if (...)
{
    ...
}
else if (option == "-verbose")
{
    // nothing to do - handled earlier
}
else
{
    error("unrecognized option");
}
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to ignore empty `then` branches if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Add the following to the where clause:

```
and not exists(ifstmt.getElse())
```

   The `where` clause is now:

```
where ifstmt.getThen() = block and
  block.isEmpty() and
  not exists(ifstmt.getElse())
```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer included.

See this in the query console

4.1.4 Further reading

- CodeQL queries for C#
- Example queries for C#
- CodeQL library reference for C#
- QL language reference
- CodeQL tools

4.2 CodeQL library for C#

When youre analyzing a C# program, you can make use of the large collection of classes in the CodeQL library for C#.

4.2.1 About the CodeQL libraries for C#

There is an extensive core library for analyzing CodeQL databases extracted from C# projects. The classes in this library present the data from a database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks. The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `csharp.qll` imports all the core C# library modules, so you can include the complete library by beginning your query with:

```
import csharp
```

Since this is required for all C# queries, its omitted from code snippets below.

The core library contains all the program elements, including *files*, *types*, methods, *variables*, *statements*, and *expressions*. This is sufficient for most queries, however additional libraries can be imported for bespoke functionality such as control flow and data flow. For information about these additional libraries, see *CodeQL for C#*.

Class hierarchies

Each section contains a class hierarchy, showing the inheritance structure between CodeQL classes. For example:

- `Expr`
  - `Operation`
    - `ArithmeticOperation`
      - `UnaryArithmeticOperation`
      - `UnaryMinusExpr`, `UnaryPlusExpr`
      - `MutatorOperation`
      - `IncrementOperation`
      - `PreIncrExpr`, `PostIncrExpr`
      - `DecrementOperation`
      - `PreDecrExpr`, `PostDecrExpr`
      - `BinaryArithmeticOperation`
      - `AddExpr`, `SubExpr`, `MulExpr`, `DivExpr`, `RemExpr`

This means that the class `AddExpr` extends class `BinaryArithmeticOperation`, which in turn extends class `ArithmeticOperation` and so on. If you want to query any arithmetic operation, use the class `ArithmeticOperation`, but if you specifically want to limit the query to addition operations, use the class `AddExpr`.

Classes can also be considered to be *sets*, and the `extends` relation between classes defines a subset. Every member of class `AddExpr` is also in the class `BinaryArithmeticOperation`. In general, classes overlap and an entity can be a member of several classes.

This overview omits some of the less important or intermediate classes from the class hierarchy.

Each class has predicates, which are logical propositions about that class. They also define navigable relationships between classes. Predicates are inherited, so for example the `AddExpr` class inherits the predicates `getLeftOperand()` and `getRightOperand()` from `BinaryArithmeticOperation`, and `getType()` from class `Expr`. This is similar to how methods are inherited in object-oriented programming languages.

In this overview, we present the most common and useful predicates. For the complete list of predicates available on each class, you can look in the CodeQL source code, use autocomplete in the editor, or see the C# reference.

Exercises

Each section in this topic contains exercises to check your understanding.

Exercise 1: Simplify this query:

```
from BinaryArithmeticOperation op
where op instanceof AddExpr
select op
```

(*Answer*)

4.2.2 Files

Files are represented by the class File, and directories by the class Folder. The database contains all of the source files and assemblies used during the compilation.

Class hierarchy

- `File` - any file in the database (including source files, XML and assemblies)
  - `SourceFile` - a file containing source code
- `Folder` - a directory

Predicates

- `getName()` - gets the full path of the file (for example, `C:\Temp\test.cs`).
- `getNumberOfLines()` - gets the number of lines (for source files only).
- `getShortName()` - gets the name of the file without the extension (for example, `test`).
- `getBaseName()` - gets the name and extension of the file (for example, `test.cs`).
- `getParent()` - gets the parent directory.

Examples

Count the number of source files:

```
select count(SourceFile f)
```

Count the number of lines of code, excluding the directory `external`:

```
select sum(SourceFile f |
  not exists(Folder external | external.getShortName() = "external" |
             external.getAFolder*().getAFile() = f) |
  f.getNumberOfLines())
```

Exercises

Exercise 2: Write a query to find the source file with the largest number of lines. Hint: Find the source file with the same number of lines as the `max` number of lines in any file. (*Answer*)

4.2.3 Elements

The class Element is the base class for all parts of a C# program, and its the root of the element class hierarchy. All program elements (such as types, methods, statements, and expressions) ultimately derive from this common base class.

`Element` forms a hierarchical structure of the program, which can be navigated using the `getParent()` and `getChild()` predicates. This is much like an abstract syntax tree, and also applies to elements in assemblies.

Predicates

The `Element` class provides common functionality for all program elements, including:

- `getLocation()` - gets the text span in the source code.
- `getFile()` - gets the `File` containing the `Element`.
- `getParent()` - gets the parent `Element`, if any.
- `getAChild()` - gets a child `Element` of this element, if any.

Examples

To list all elements in `Main.cs`, their QL class and location:

```
from Element e
where e.getFile().getShortName() = "Main"
select e, e.getAQlClass(), e.getLocation()
```

Note that `getAQlClass()` is available on all entities and is a useful way to figure out the QL class of something. Often the same element will have several classes which are all returned by `getAQlClass()`.

4.2.4 Locations

Location represents a section of text in the source code, or an assembly. All elements have a `Location` obtained by their `getLocation()` predicate. A `SourceLocation` represents a span of text in source code, whereas an `Assembly` location represents a referenced assembly.

Sometimes elements have several locations, for example if they occur in both source code and an assembly. In this case, only the `SourceLocation` is returned.

Class hierarchy

- `Location`
  - `SourceLocation`
  - `Assembly`

Predicates

Some predicates of `Location` include:

- `getFile()` - gets the `File`.
- `getStartLine()` - gets the first line of the text.
- `getEndLine()` - gets the last line of the text.
- `getStartColumn()` - gets the column of the start of the text.
- `getEndColumn()` - gets the column of the end of the text.

Examples

Find all elements that are one character wide:

```
from Element e, Location l
where l = e.getLocation()
  and l.getStartLine() = l.getEndLine()
  and l.getStartColumn() = l.getEndColumn()
select e, "This element is a single character."
```

4.2.5 Declarations

Declaration is the common class of all entities defined in the program, such as types, methods, variables etc. The database contains all declarations from the source code and all referenced assemblies.

Class hierarchy

- `Element`
  - `Declaration`

    `Callable`

    `UnboundGeneric`

    `ConstructedGeneric`

    `Modifiable` - a declaration which can have a modifier (for example `public`)

    - `Member` - a declaration that is member of a type

    `Assignable` - an element that can be assigned to

    - `Variable`
    - `Property`
    - `Indexer`
    - `Event`

Predicates

Useful member predicates on `Declaration` include:

- `getDeclaringType()` - gets the type containing the declaration, if any.
- `getName()`/`hasName(string)` - gets the name of the declared entity.
- `isSourceDeclaration()` - whether the declaration is source code and is not a constructed type/method.
- `getSourceDeclaration()` - gets the original (unconstructed) declaration.

Examples

Find declarations containing a username:

```
from Declaration decl
where decl.getName().regexpMatch("[uU]ser([Nn]ame)?")
select decl, "A username."
```

4.2.6 Variables

The class Variable represents C# variables, such as fields, parameters and local variables. The database contains all variables from the source code, as well as all fields and parameters from assemblies referenced by the program.

Class hierarchy

- `Element`
  - `Declaration`
    - `Variable` - any type of variable
      - `Field` - a field in a `class`/`struct`
      - `MemberConstant` - a `const` field
      - `EnumConstant` - a field in an `enum`
      - `LocalScopeVariable` - a variable whose scope is limited to a single `Callable`
      - `LocalVariable` - a local variable in a `Callable`
      - `LocalConstant` - a locally defined constant in a `Callable`
      - `Parameter` - a parameter to a `Callable`

Predicates

Some common predicates on `Variable` are:

- `getType()` - gets the `Type` of this variable.
- `getAnAccess()` - gets an expression that accesses (reads or writes) this variable, if any.
- `getAnAssignedValue()` - gets an expression that is assigned to this variable, if any.
- `getInitializer()` - gets the expression used to initialize the variable, if any.

Examples

Find all unused local variables:

```
from LocalVariable v
where not exists(v.getAnAccess())
select v, "This local variable is unused."
```

4.2.7 Types

Types are represented by the CodeQL class Type and consist of builtin types, interfaces, classes, structs, enums, and type parameters. The database contains types from the program and all referenced assemblies including mscorlib and the .NET framework.

The builtin types (`object`, `int`, `double` etc.) have corresponding types (`System.Object`, `System.Int32` etc.) in mscorlib.

Class `ValueOrRefType` represents defined types, such as a `class`, `struct`, `interface` or `enum`.

Class hierarchy

- `Element`
  - `Declaration`

    `Modifiable` - a declaration which can have a modifier (for example `public`)

    - `Member` - a declaration that is member of a type
    - `Type` - all types
    - `ValueOrRefType` - a defined type
    - `ValueType` - a value type (see below for further hierarchy)
    - `RefType` - a reference type (see below for further hierarchy)
    - `NestedType` - a type defined in another type
    - `VoidType` - `void`
    - `PointerType` - a pointer type

The `ValueType` class extends further:

- `ValueType` - a value type
  - `SimpleType` - a simple built-in type

    `BoolType` - `bool`

    `CharType` - `char`

    `IntegralType`

    - `UnsignedIntegralType`
    - `ByteType` - `byte`
    - `UShortType` - `unsigned short`/`System.UInt16`
    - `UIntType` - `unsigned int`/`System.UInt32`
    - `ULongType` - `unsigned long`/`System.UInt64`
    - `SignedIntegralType`
    - `SByteType` - `signed byte`
    - `ShortType` - `short`/`System.Int16`
    - `IntType` - `int`/`System.Int32`
    - `LongType` - `long`/`System.Int64`
    - `FloatingPointType`
    - `FloatType` - `float`/`System.Single`
    - `DoubleType` - `double`/`System.Double`

        · DecimalType - decimal/System.Decimal

    Enum - an enum

    Struct - a struct

    NullableType

    ArrayType

The RefType class extends further:

- RefType
  - Class - a class

    AnonymousClass

    ObjectType - object/System.Object

    StringType - string/System.String
  - Interface - an interface
  - DelegateType
  - NullType - the type of null
  - DynamicType - dynamic
- NestedType - a type defined in another type

These class hierarchies omit generic types for simplicity.

Predicates

Useful members of ValueOrRefType include:

- getQualifiedName()/hasQualifiedName(string) - gets the qualified name of the type (for example, "System.String").
- getABaseInterface() - gets an immediate interface of this type, if any.
- getABaseType() - gets an immediate base class or interface of this type, if any.
- getBaseClass() - gets the immediate base class of this type, if any.
- getASubType() - gets an immediate subtype, a type which directly inherits from this type, if any.
- getAMember() - gets any member (field/method/property etc), if any.
- getAMethod() - gets a method, if any.
- getAProperty() - gets a property, if any.
- getAnIndexer() - gets an indexer, if any.
- getAnEvent() - gets an event, if any.
- getAnOperator() - gets an operator, if any.
- getANestedType() - gets a nested type.
- getNamespace() - gets the enclosing namespace.

Examples

Find all members of `System.Object`:

```
from ObjectType object
select object.getAMember()
```

Find all types which directly implement `System.Collections.IEnumerable`:

```
from Interface ienumerable
where ienumerable.hasQualifiedName("System.Collections.IEnumerable")
select ienumerable.getASubType()
```

List all simple types in the `System` namespace:

```
select any(SimpleType t | t.getNamespace().hasName("System"))
```

Find all variables of type `PointerType`:

```
from Variable v
where v.fromSource()
  and v.getType() instanceof PointerType
select v
```

List all classes in source files:

```
from Class c
where c.fromSource()
select c
```

Exercises

Exercise 3: Write a query to list the methods in `string`. (*Answer*)

Exercise 4: Adapt the example to find all types which indirectly implement `IEnumerable`. (*Answer*)

Exercise 5: Write a query to find all classes starting with the letter `A`. (*Answer*)

4.2.8 Callables

Callables are represented by the class Callable and are anything that can be called independently, such as methods, constructors, destructors, operators, anonymous functions, indexers, and property accessors.

The database contains all of the callables in your program and in all referenced assemblies.

Class hierarchy

- `Element`
  - `Declaration`
    `Callable`
    - `Method`

- ExtensionMethod
- Constructor
- StaticConstructor
- InstanceConstructor
- Destructor
- Operator
- UnaryOperator
- PlusOperator, MinusOperator, NotOperator, ComplementOperator, IncrementOperator, DecrementOperator, FalseOperator, TrueOperator
- BinaryOperator
- AddOperator, SubOperator, MulOperator, DivOperator, RemOperator, AndOperator, OrOperator, XorOperator, LShiftOperator, RShiftOperator, EQOperator, NEOperator, LTOperator, GTOperator, LEOperator, GEOperator
- ConversionOperator
- ImplicitConversionOperator
- ExplicitConversionOperator
- AnonymousFunctionExpr
- LambdaExpr
- AnonymousMethodExpr
- Accessor
- Getter
- Setter
- EventAccessor
- AddEventAccessor, RemoveEventAccessor

Predicates

Here are a few useful predicates on the `Callable` class:

- `getParameter(int)`/`getAParameter()` - gets a parameter.
- `calls(Callable)` - whether theres a direct call from one callable to another.
- `getReturnType()` - gets the return type.
- `getBody()`/`getExpressionBody()` - gets the body of the callable.

Since `Callable` extends `Declaration`, it also has predicates from `Declaration`, such as:

- `getName()`/`hasName(string)`
- `getSourceDeclaration()`
- `getName()`

- `getDeclaringType()`

Methods have additional predicates, including:

- `getAnOverridee()` - gets a method that is immediately overridden by this method.
- `getAnOverrider()` - gets a method that immediately overrides this method.
- `getAnImplementee()` - gets an interface method that is immediately implemented by this method.
- `getAnImplementor()` - gets a method that immediately implements this interface method.

Examples

List all types which override `ToString`:

```
from Method m
where m.hasName("ToString")
select m
```

Find methods that look like `ToString` methods but dont override `Object.ToString`:

```
from Method toString, Method falseToString
where toString.hasQualifiedName("System.Object.ToString")
 and falseToString.getName().toLowerCase() = "tostring"
 and not falseToString.overrides*(toString)
 and falseToString.getNumberOfParameters() = 0
select falseToString, "This method looks like it overrides Object.ToString but it doesn't."
```

Find all methods which take a pointer type:

```
from Method m
where m.getAParameter().getType() instanceof PointerType
select m, "This method uses pointers."
```

Find all classes which have a destructor but arent disposable:

```
from Class c
where c.getAMember() instanceof Destructor
  and not c.getABaseType*().hasQualifiedName("System.IDisposable")
select c, "This class has a destructor but is not IDisposable."
```

Find `Main` methods which are not `private`:

```
from Method m
where m.hasName("Main")
  and not m.isPrivate()
select m, "Main method should be private."
```

4.2.9 Statements

Statements are represented by the class Stmt and make up the body of methods (and other callables). The database contains all statements in the source code, but does not contain any statements from referenced assemblies where the source code is not available.

Class hierarchy

- Element
  - ControlFlowElement
    - Stmt
      - BlockStmt - { ... }
      - ExprStmt
      - SelectionStmt
      - IfStmt - if
      - SwitchStmt - switch
      - LabeledStmt
      - ConstCase
      - DefaultCase - default
      - LabelStmt
      - LoopStmt
      - WhileStmt - while(...) { ... }
      - DoStmt - do { ... } while(...)
      - ForStmt - for
      - ForEachStmt - foreach
      - JumpStmt
      - BreakStmt - break
      - ContinueStmt - continue
      - GotoStmt - goto
      - GotoLabelStmt
      - GotoCaseStmt
      - GotoDefaultStmt
      - ThrowStmt - throw
      - ReturnStmt - return
      - YieldStmt
      - YieldBreakStmt - yield break
      - YieldReturnStmt - yield return
      - TryStmt - try
      - CatchClause - catch
      - SpecificCatchClause
      - GeneralCatchClause

- CheckedStmt - checked
- UncheckedStmt - unchecked
- LockStmt - lock
- UsingStmt - using
- LocalVariableDeclStmt
- LocalConstantDeclStmt
- EmptyStmt - ;
- UnsafeStmt - unsafe
- FixedStmt - fixed

Examples

Find long methods:

```
from Method m
where m.getBody().(BlockStmt).getNumberOfStmts() >= 100
select m, "This is a long method!"
```

Find `for(;;)`:

```
from ForStmt for
where not exists(for.getAnInitializer())
  and not exists(for.getUpdate(_))
  and not exists(for.getCondition())
select for, "Infinite loop."
```

Find `catch(NullDefererenceException)`:

```
from SpecificCatchClause catch
where catch.getCaughtExceptionType().hasQualifiedName("System.NullReferenceException")
select catch, "Catch NullReferenceException."
```

Find an `if` statement with a constant condition:

```
from IfStmt ifStmt
where ifStmt.getCondition().hasValue()
select ifStmt, "This 'if' statement is constant."
```

Find an `if` statement with an empty then block:

```
from IfStmt ifStmt
where ifStmt.getThen().(BlockStmt).isEmpty()
select ifStmt, "If statement with empty 'then' block."
```

The `(BlockStmt)` is an inline cast, which restricts the query to cases where the result of `getThen()` has the QL class `BlockStmt`, and allows predicates on `BlockStmt` to be used, such as `isEmpty()`.

Exercises

Exercise 6: Write a query to list all empty methods. (*Answer*)

Exercise 7: Modify the last example to also detect empty statements (`;`) in the then block. (*Answer*)

Exercise 8: Modify the last example to exclude chains of `if` statements, where the `else` part is another `if` statement. (*Answer*)

4.2.10 Expressions

The Expr class represents all C# expressions in the program. An expression is something producing a value such as `a+b` or `new List<int>()`. The database contains all expressions from the source code, but no expressions from referenced assemblies where the source code is not available.

The `Access` class represents any use or cross-reference of another `Declaration` such a variable, property, method or field. The `getTarget()` predicate gets the declaration being accessed.

The `Call` class represents a call to a `Callable`, for example to a `Method` or an `Accessor`, and the `getTarget()` method gets the `Callable` being called. The `Operation` class consists of arithmetic, bitwise operations and logical operations.

Some expressions use a qualifier, which is the object on which the expression operates. A typical example is a `MethodCall`. In this case, the `getQualifier()` predicate is used to get the expression on the left of the `.`, and `getArgument(int)` is used to get the arguments of the call.

Class hierarchy

- `Element`
  - `ControlFlowElement`
    - `Expr`
      - `LocalVariableDeclExpr`
      - `LocalConstantDeclExpr`
      - `Operation`
      - `UnaryOperation`
      - `SizeofExpr`, `PointerIndirectionExpr`, `AddressOfExpr`
      - `BinaryOperation`
      - `ComparisonOperation`
      - `EqualityOperation`
      - `EQExpr`, `NEExpr`
      - `RelationalOperation`
      - `GTExpr`, `LTExpr`, `GEExpr`, `LEExpr`
      - `Assignment`
      - `AssignOperation`
      - `AddOrRemoveEventExpr`

- AddEventExpr
- RemoveEventExpr
- AssignArithmeticOperation
- AssignAddExpr, AssignSubExpr, AssignMulExpr, AssignDivExpr, AssignRemExpr
- AssignBitwiseOperation
- AssignAndExpr, AssignOrExpr, AssignXorExpr, AssignLShiftExpr, AssignRShiftExpr
- AssignExpr
- MemberInitializer
- ArithmeticOperation
- UnaryArithmeticOperation
- UnaryMinusExpr, UnaryPlusExpr
- MutatorOperation
- IncrementOperation
- PreIncrExpr, PostIncrExpr
- DecrementOperation
- PreDecrExpr, PostDecrExpr
- BinaryArithmeticOperation
- AddExpr, SubExpr, MulExpr, DivExpr, RemExpr
- BitwiseOperation
- UnaryBitwiseOperation
- ComplementOperation
- BinaryBitwiseOperation
- LShiftExpr, RShiftExpr, BitwiseAndExpr, BitwiseOrExpr, BitwiseXorExpr
- LogicalOperation
- UnaryLogicalOperation
- LogicalNotOperation
- BinaryLogicalOperation
- LogicalAndExpr, LogicalOrExpr, NullCoalescingExpr
- ConditionalExpr
- ParenthesisedExpr, CheckedExpr, UncheckedExpr, IsExpr, AsExpr, CastExpr, TypeofExpr, DefaultValueExpr, AwaitExpr, NameofExpr, InterpolatedStringExpr
- Access
- ThisAccess

- BaseAccess
- MemberAccess
- MethodAccess
- VirtualMethodAccess
- FieldAccess, PropertyAccess, IndexerAccess, EventAccess, MethodAccess
- AssignableAccess
- VariableAccess
- ParameterAccess
- LocalVariableAccess
- LocalScopeVariableAccess
- FieldAccess
- MemberConstantAccess
- PropertyAccess
- TrivialPropertyAccess
- VirtualPropertyAccess
- IndexerAccess
- VirtualIndexerAccess
- EventAccess
- VirtualEventAccess
- TypeAccess
- ArrayAccess
- Call
- PropertyCall
- IndexerCall
- EventCall
- MethodCall
- VirtualMethodCall
- ElementInitializer
- ConstructorInitializer
- OperatorCall
- MutatorOperatorCall
- DelegateCall
- ObjectCreation
- DefaultValueTypeObjectCreation

- TypeParameterObjectCreation
- AnonymousObjectCreation
- ObjectOrCollectionInitializer
- ObjectInitializer
- CollectionInitializer
- DelegateCreation
- ExplicitDelegateCreation, ImplicitDelegateCreation
- ArrayInitializer
- ArrayCreation
- AnonymousFunctionExpr
- LambdaExpr
- AnonymousMethodExpr
- Literal
- BoolLiteral, CharLiteral, IntegerLiteral, IntLiteral, LongLiteral, UIntLiteral, ULongLiteral, RealLiteral, FloatLiteral, DoubleLiteral, DecimalLiteral, StringLiteral, NullLiteral

Predicates

Useful predicates on Expr include:

- getType() - gets the Type of the expression.
- getValue() - gets the compile-time constant, if any.
- hasValue() - whether the expression has a compile-time constant.
- getEnclosingStmt() - gets the statement containing the expression, if any.
- getEnclosingCallable() - gets the callable containing the expression, if any.
- stripCasts() - remove all explicit or implicit casts.
- isImplicit() - whether the expression was implicit, such as an implicit this qualifier (ThisAccess).

Examples

Find calls to String.Format with just one argument:

```
from MethodCall c
where c.getTarget().hasQualifiedName("System.String.Format")
  and c.getNumberOfArguments() = 1
select c, "Missing arguments to 'String.Format'."
```

Find all comparisons of floating point values:

```
from ComparisonOperation cmp
where (cmp instanceof EQExpr or cmp instanceof NEExpr)
  and cmp.getAnOperand().getType() instanceof FloatingPointType
select cmp, "Comparison of floating point values."
```

Find hard-coded passwords:

```
from Variable v, string value
where v.getName().regexpMatch("[pP]ass(word|wd|)")
  and value = v.getAnAssignedValue().getValue()
select v, "Hard-coded password '" + value + "'."
```

Exercises

Exercise 9: Limit the previous query to string types. Exclude empty passwords or null passwords. (*Answer*)

4.2.11 Attributes

C# attributes are represented by the class Attribute. They can be present on many C# elements, such as classes, methods, fields, and parameters. The database contains attributes from the source code and all assembly references.

The attribute of any `Element` can be obtained via `getAnAttribute()`, whereas if you have an attribute, you can find its element via `getTarget()`. These two query fragments are identical:

```
attribute = element.getAnAttribute()
element = attribute.getTarget()
```

Class hierarchy

- `Element`
  - `Attribute`

Predicates

- `getTarget()` - gets the `Element` to which this attribute applies.
- `getArgument(int)` - gets the given argument of the attribute.
- `getType()` - gets the type of this attribute. Note that the class name must end in `"Attribute"`.

Examples

Find all obsolete elements:

```
from Element e, Attribute attribute
where e = attribute.getTarget()
  and attribute.getType().hasName("ObsoleteAttribute")
select e, "This is obsolete because " + attribute.getArgument(0).getValue()
```

Model NUnit test fixtures:

```
class TestFixture extends Class
{
  TestFixture() {
    this.getAnAttribute().getType().hasName("TestFixtureAttribute")
  }

  TestMethod getATest() {
    result = this.getAMethod()
  }
}

class TestMethod extends Method
{
  TestMethod() {
    this.getAnAttribute().getType().hasName("TestAttribute")
  }
}

from TestFixture f
select f, f.getATest()
```

Exercises

Exercise 10: Write a query to find just obsolete methods. (*Answer*)

Exercise 11: Write a query to find all places where the `Obsolete` attribute is used without a reason string (that is, `[Obsolete]`). (*Answer*)

Exercise 12: In the first example, what happens if the `Obsolete` attribute doesnt have a reason string? How could the query be fixed to accommodate this? (*Answer*)

4.2.12 Answers

Exercise 1

```
from AddExpr op
select op
```

or

```
select any(AddExpr op)
```

Exercise 2

```
from File f
where f.getNumberOfLines() = max(any(File g).getNumberOfLines())
select f
```

Exercise 3

```
from StringType s
select s.getAMethod()
```

Exercise 4

```
from Interface ienumerable
where ienumerable.hasQualifiedName("System.Collections.IEnumerable")
select ienumerable.getASubType*()
```

Exercise 5

```
from Class a
where a.getName().toLowerCase().matches("a%")
select a
```

Exercise 6

```
select any(Method m | m.getBody().(BlockStmt).isEmpty())
```

Exercise 7

```
from IfStmt ifStmt
where ifStmt.getThen().(BlockStmt).isEmpty() or ifStmt.getThen() instanceof EmptyStmt
select ifStmt, "If statement with empty 'then' block."
```

Exercise 8

```
from IfStmt ifStmt
where (ifStmt.getThen().(BlockStmt).isEmpty() or ifStmt.getThen() instanceof EmptyStmt)
  and not ifStmt.getElse() instanceof IfStmt
select ifStmt, "If statement with empty 'then' block."
```

Exercise 9

```
from Variable v, StringLiteral value
where v.getName().regexpMatch("[pP]ass(word|wd|)")
  and value = v.getAnAssignedValue()
  and value.getValue() != ""
select v, "Hard-coded password '" + value.getValue() + "'."
```

Exercise 10

```
from Method method, Attribute attribute
where method = attribute.getTarget()
  and attribute.getType().hasName("ObsoleteAttribute")
select method, "This is obsolete because " + attribute.getArgument(0).getValue()
```

Exercise 11

```
from Attribute attribute
where attribute.getType().hasName("ObsoleteAttribute")
  and not exists(attribute.getArgument(0))
select attribute, "Missing reason in 'Obsolete' attribute."
```

Exercise 12

The query does not return results where the argument is missing.

Here is the fixed version:

```
from Element e, Attribute attribute, string reason
where e = attribute.getTarget()
  and attribute.getType().hasName("ObsoleteAttribute")
  and if exists(attribute.getArgument(0))
    then reason = attribute.getArgument(0).getValue()
    else reason = "(not given)"
select e, "This is obsolete because " + reason
```

4.2.13 Further reading

- CodeQL queries for C#
- Example queries for C#
- CodeQL library reference for C#
- QL language reference
- CodeQL tools

4.3 Analyzing data flow in C#

You can use CodeQL to track the flow of data through a C# program to its use.

4.3.1 About this article

This article describes how data flow analysis is implemented in the CodeQL libraries for C# and includes examples to help you write your own data flow queries. The following sections describe how to use the libraries for local data flow, global data flow, and taint tracking. For a more general introduction to modeling data flow, see *About data flow analysis*.

4.3.2 Local data flow

Local data flow is data flow within a single method or callable. Local data flow is easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. `Node`s are divided into expression nodes (`ExprNode`) and parameter nodes (`ParameterNode`). You can map between data flow nodes and expressions/parameters using the member predicates `asExpr` and `asParameter`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the parameter corresponding to this node, if any. */
  Parameter asParameter() { ... }

 ...
}
```

or using the predicates `exprNode` and `parameterNode`:

```
/**
 * Gets the node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }

/**
 * Gets the node corresponding to the value of parameter `p` at function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively, by using the `+` and `*` operators, or you can use the predefined recursive predicate `localFlow`.

For example, you can find flow from a parameter `source` to an expression `sink` in zero or more local steps:

```
DataFlow::localFlow(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
var temp = x;
var y = temp + ", " + temp;
```

If `x` is a tainted string then `y` is also tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint

propagation edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively, by using the `+` and `*` operators, or you can use the predefined recursive predicate `localTaint`.

For example, you can find taint propagation from a parameter `source` to an expression `sink` in zero or more local steps:

```
TaintTracking::localTaint(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Examples

This query finds the filename passed to `System.IO.File.Open`:

```
import csharp

from Method fileOpen, MethodCall call
where fileOpen.hasQualifiedName("System.IO.File.Open")
  and call.getTarget() = fileOpen
select call.getArgument(0)
```

Unfortunately this will only give the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import csharp

from Method fileOpen, MethodCall call, Expr src
where fileOpen.hasQualifiedName("System.IO.File.Open")
  and call.getTarget() = fileOpen
  and DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(call.getArgument(0)))
select src
```

Then we can make the source more specific, for example an access to a public parameter. This query finds instances where a public parameter is used to open a file:

```
import csharp

from Method fileOpen, MethodCall call, Parameter p
where fileOpen.hasQualifiedName("System.IO.File.Open")
  and call.getTarget() = fileOpen
  and DataFlow::localFlow(DataFlow::parameterNode(p), DataFlow::exprNode(call.getArgument(0)))
  and call.getEnclosingCallable().(Member).isPublic()
select p, "Opening a file from a public method."
```

This query finds calls to `String.Format` where the format string isnt hard-coded:

```
import csharp

from Method format, MethodCall call, Expr formatString
where format.hasQualifiedName("System.String.Format")
  and call.getTarget() = format
  and formatString = call.getArgument(0)
  and formatString.getType() instanceof StringType
```

(continues on next page)

(continued from previous page)

```
  and not exists(StringLiteral source | DataFlow::localFlow(DataFlow::exprNode(source),␣
↪DataFlow::exprNode(formatString)))
select call, "Argument to 'string.Format' isn't hard-coded."
```

Exercises

Exercise 1: Write a query that finds all hard-coded strings used to create a `System.Uri`, using local data flow. (*Answer*)

4.3.3 Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

> Note
>
> You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see Creating path queries.

Using global data flow

The global data flow library is used by extending the class `DataFlow::Configuration`:

```
import csharp

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "..." }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource` - defines where data may flow from.
- `isSink` - defines where data may flow to.
- `isBarrier` - optionally, restricts the data flow.
- `isAdditionalFlowStep` - optionally, adds additional flow steps.

The characteristic predicate (`MyDataFlowConfiguration()`) defines the name of the configuration, so `"..."` must be replaced with a unique name.

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguation dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Dataflow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow what local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. The global taint tracking library is used by extending the class `TaintTracking::Configuration`:

```
import csharp

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "..." }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource` - defines where taint may flow from.
- `isSink` - defines where taint may flow to.
- `isSanitizer` - optionally, restricts the taint flow.
- `isAdditionalTaintStep` - optionally, adds additional taint steps.

Similar to global data flow, the characteristic predicate (`MyTaintTrackingConfiguration()`) defines the unique name of the configuration and the taint analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Flow sources

The data flow library contains some predefined flow sources. The class `PublicCallableParameterFlowSource` (defined in module `semmle.code.csharp.dataflow.flowsources.PublicCallableParameter`) represents data flow from public parameters, which is useful for finding security problems in a public API.

The class `RemoteSourceFlow` (defined in module `semmle.code.csharp.dataflow.flowsources.Remote`) represents data flow from remote network inputs. This is useful for finding security problems in networked services.

Example

This query shows a data flow configuration that uses all public API parameters as data sources:

```
import csharp
import semmle.code.csharp.dataflow.flowsources.PublicCallableParameter
```

(continues on next page)

(continued from previous page)

```
class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() {
    this = "..."
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof PublicCallableParameterFlowSource
  }

  ...
}
```

Class hierarchy

- `DataFlow::Configuration` - base class for custom global data flow analysis.
- `DataFlow::Node` - an element behaving as a data flow node.
  - `DataFlow::ExprNode` - an expression behaving as a data flow node.
  - `DataFlow::ParameterNode` - a parameter data flow node representing the value of a parameter at function entry.

    `PublicCallableParameter` - a parameter to a public method/callable in a public class.
  - `RemoteSourceFlow` - data flow from network/remote input.

    `AspNetRemoteFlowSource` - data flow from remote ASP.NET user input.

    - `AspNetQueryStringRemoteFlowSource` - data flow from `System.Web.HttpRequest`.
    - `AspNetUserInputRemoveFlowSource` - data flow from `System.Web.IO.WebControls.TextBox`.

    `WcfRemoteFlowSource` - data flow from a WCF web service.

    `AspNetServiceRemoteFlowSource` - data flow from an ASP.NET web service.
- `TaintTracking::Configuration` - base class for custom global taint tracking analysis.

Examples

This data flow configuration tracks data flow from environment variables to opening files:

```
import csharp

class EnvironmentToFileConfiguration extends DataFlow::Configuration {
  EnvironmentToFileConfiguration() { this = "Environment opening files" }

  override predicate isSource(DataFlow::Node source) {
    exists(Method m |
      m = source.asExpr().(MethodCall).getTarget() and
      m.hasQualifiedName("System.Environment.GetEnvironmentVariable")
    )
```

(continues on next page)

(continued from previous page)

```
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(MethodCall mc |
      mc.getTarget().hasQualifiedName("System.IO.File.Open") and
      sink.asExpr() = mc.getArgument(0)
    )
  }
}

from Expr environment, Expr fileOpen, EnvironmentToFileConfiguration config
where config.hasFlow(DataFlow::exprNode(environment), DataFlow::exprNode(fileOpen))
select fileOpen, "This 'File.Open' uses data from $@.",
  environment, "call to 'GetEnvironmentVariable'"
```

Exercises

Exercise 2: Find all hard-coded strings passed to `System.Uri`, using global data flow. (*Answer*)

Exercise 3: Define a class that represents flow sources from `System.Environment.GetEnvironmentVariable`. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query to find all global data flow from `System.Environment.GetEnvironmentVariable` to `System.Uri`. (*Answer*)

4.3.4 Extending library data flow

Library data flow defines how data flows through libraries where the source code is not available, such as the .NET Framework, third-party libraries or proprietary libraries.

To define new library data flow, extend the class `LibraryTypeDataFlow` from the module `semmle.code.csharp.dataflow.LibraryTypeDataFlow`. Override the predicate `callableFlow` to define how data flows through the methods in the class. `callableFlow` has the signature

```
predicate callableFlow(CallableFlowSource source, CallableFlowSink sink, SourceDeclarationCallable␣
↪callable, boolean preservesValue)
```

- `callable` - the `Callable` (such as a method, constructor, property getter or setter) performing the data flow.
- `source` - the data flow input.
- `sink` - the data flow output.
- `preservesValue` - whether the flow step preserves the value, for example if `x` is a string then `x.ToString()` preserves the value where as `x.ToLower()` does not.

Class hierarchy

- `Callable` - a callable (methods, accessors, constructors etc.)
  - `SourceDeclarationCallable` - an unconstructed callable.
- `CallableFlowSource` - the input of data flow into the callable.

  - `CallableFlowSourceQualifier` - the data flow comes from the object itself.
  - `CallableFlowSourceArg` - the data flow comes from an argument to the call.
- `CallableFlowSink` - the output of data flow from the callable.
  - `CallableFlowSinkQualifier` - the output is to the object itself.
  - `CallableFlowSinkReturn` - the output is returned from the call.
  - `CallableFlowSinkArg` - the output is an argument.
  - `CallableFlowSinkDelegateArg` - the output flows through a delegate argument (for example, LINQ).

Example

This example is adapted from `LibraryTypeDataFlow.qll`. It declares data flow through the class `System.Uri`, including the constructor, the `ToString` method, and the properties `Query`, `OriginalString`, and `PathAndQuery`.

```
import semmle.code.csharp.dataflow.LibraryTypeDataFlow
import semmle.code.csharp.frameworks.System

class SystemUriFlow extends LibraryTypeDataFlow, SystemUriClass {
  override predicate callableFlow(CallableFlowSource source, CallableFlowSink sink,␣
↪SourceDeclarationCallable c, boolean preservesValue) {
    (
      constructorFlow(source, c) and
      sink instanceof CallableFlowSinkQualifier
      or
      methodFlow(c) and
      source instanceof CallableFlowSourceQualifier and
      sink instanceof CallableFlowSinkReturn
      or
      exists(Property p |
        propertyFlow(p) and
        source instanceof CallableFlowSourceQualifier and
        sink instanceof CallableFlowSinkReturn and
        c = p.getGetter()
      )
    )
    and
    preservesValue = false
  }

  private predicate constructorFlow(CallableFlowSourceArg source, Constructor c) {
    c = getAMember()
    and
    c.getParameter(0).getType() instanceof StringType
    and
    source.getArgumentIndex() = 0
  }

  private predicate methodFlow(Method m) {
```

(continues on next page)

(continued from previous page)

```
    m.getDeclaringType() = getABaseType*()
    and
    m = getSystemObjectClass().getToStringMethod().getAnOverrider*()
  }

  private predicate propertyFlow(Property p) {
    p = getPathAndQueryProperty()
    or
    p = getQueryProperty()
    or
    p = getOriginalStringProperty()
  }
}
```

This defines a new class `SystemUriFlow` which extends `LibraryTypeDataFlow` to add another case. It extends `SystemUriClass` (the class representing `System.Uri`, defined in the module `semmle.code.csharp.frameworks.System`) to access methods such as `getQueryProperty`.

The predicate `callableFlow` declares data flow through `System.Uri`. The first case (`constructorFlow`) declares data flow from the first argument of the constructor to the object itself (`CallableFlowSinkQualifier`).

The second case declares data flow from the object (`CallableFlowSourceQualifier`) to the result of calling `ToString` on the object (`CallableFlowSinkReturn`).

The third case declares data flow from the object (`CallableFlowSourceQualifier`) to the return (`CallableFlowSinkReturn`) of the getters for the properties `PathAndQuery`, `Query` and `OriginalString`. Note that the properties (`getPathAndQueryProperty`, `getQueryProperty` and `getOriginalStringProperty`) are inherited from the class `SystemUriClass`.

In all three cases `preservesValue = false`, which means that these steps will only be included in taint tracking, not in (normal) data flow.

Exercises

Exercise 5: In `System.Uri`, what other properties could expose data? How could they be added to `SystemUriFlow`? (*Answer*)

Exercise 6: Implement the data flow for the class `System.Exception`. (*Answer*)

4.3.5 Answers

Exercise 1

```
import csharp

from Expr src, Call c
where DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(c.getArgument(0)))
  and c.getTarget().(Constructor).getDeclaringType().hasQualifiedName("System.Uri")
  and src.hasValue()
select src, "This string constructs 'System.Uri' $@.", c, "here"
```

Exercise 2

```
import csharp

class Configuration extends DataFlow::Configuration {
  Configuration() { this="String to System.Uri" }

  override predicate isSource(DataFlow::Node src) {
    src.asExpr().hasValue()
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call c | c.getTarget().(Constructor).getDeclaringType().hasQualifiedName("System.Uri")
    and sink.asExpr()=c.getArgument(0))
  }
}

from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This string constructs a 'System.Uri' $@.", sink, "here"
```

Exercise 3

```
class EnvironmentVariableFlowSource extends DataFlow::ExprNode {
  EnvironmentVariableFlowSource() {
    this.getExpr().(MethodCall).getTarget().hasQualifiedName("System.Environment.
↪GetEnvironmentVariable")
  }
}
```

Exercise 4

```
import csharp

class EnvironmentVariableFlowSource extends DataFlow::ExprNode {
  EnvironmentVariableFlowSource() {
    this.getExpr().(MethodCall).getTarget().hasQualifiedName("System.Environment.
↪GetEnvironmentVariable")
  }
}

class Configuration extends DataFlow::Configuration {
  Configuration() { this="Environment to System.Uri" }

  override predicate isSource(DataFlow::Node src) {
    src.asExpr() instanceof EnvironmentVariableFlowSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call c | c.getTarget().(Constructor).getDeclaringType().hasQualifiedName("System.Uri")
    and sink.asExpr()=c.getArgument(0))
```

(continues on next page)

(continued from previous page)

```
  }
}

from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This environment variable constructs a 'System.Uri' $@.", sink, "here"
```

Exercise 5

All properties can flow data:

```
private predicate propertyFlow(Property p) {
  p = getAMember()
}
```

Exercise 6

This can be adapted from the `SystemUriFlow` class:

```
import semmle.code.csharp.dataflow.LibraryTypeDataFlow
import semmle.code.csharp.frameworks.System

class SystemExceptionFlow extends LibraryTypeDataFlow, SystemExceptionClass {
  override predicate callableFlow(CallableFlowSource source, CallableFlowSink sink,␣
↪SourceDeclarationCallable c, boolean preservesValue) {
    (
      constructorFlow(source, c) and
      sink instanceof CallableFlowSinkQualifier
      or
      methodFlow(source, sink, c)
      or
      exists(Property p |
        propertyFlow(p) and
        source instanceof CallableFlowSourceQualifier and
        sink instanceof CallableFlowSinkReturn and
        c = p.getGetter()
      )
    )
    and
    preservesValue = false
  }

  private predicate constructorFlow(CallableFlowSourceArg source, Constructor c) {
    c = getAMember()
    and
    c.getParameter(0).getType() instanceof StringType
    and
    source.getArgumentIndex() = 0
  }
```

(continues on next page)

(continued from previous page)

```
  private predicate methodFlow(CallableFlowSourceQualifier source, CallableFlowSinkReturn sink,␣
↪SourceDeclarationMethod m) {
    m.getDeclaringType() = getABaseType*()
    and
    m = getSystemObjectClass().getToStringMethod().getAnOverrider*()
  }

  private predicate propertyFlow(Property p) {
    p = getAProperty() and p.hasName("Message")
  }
}
```

4.3.6 Further reading

- Exploring data flow with path queries
- CodeQL queries for C#
- Example queries for C#
- CodeQL library reference for C#
- QL language reference
- CodeQL tools
- *Basic query for C# code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for C#*: When youre analyzing a C# program, you can make use of the large collection of classes in the CodeQL library for C#.
- *Analyzing data flow in C#*: You can use CodeQL to track the flow of data through a C# program to its use.

CHAPTER

FIVE

CODEQL FOR GO

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Go codebases.

5.1 Basic query for Go code

Learn to write and run a simple CodeQL query using LGTM.

5.1.1 About the query

The query were going to run searches the code for methods defined on value types that modify their receiver by writing a field:

```
func (s MyStruct) valueMethod() { s.f = 1 } // method on value
```

This is problematic because the receiver argument is passed by value, not by reference. Consequently, valueMethod is called with a copy of the receiver object, so any changes it makes to the receiver will be invisible to the caller. To prevent this, the method should be defined on a pointer instead:

```
func (s *MyStruct) pointerMethod() { s.f = 1 } // method on pointer
```

For further information on using methods on values or pointers in Go, see the Go FAQ.

5.1.2 Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Go** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import go

from Method m, Variable recv, Write w, Field f
where
  recv = m.getReceiver() and
  w.writesField(recv.getARead(), f, _) and
  not recv.getType() instanceof PointerType
select w, "This update to " + f + " has no effect, because " + recv + " is not a pointer."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   > Note
   >
   > Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to `w`, which is the location in the source code where the receiver `recv` is modified. The second column is the alert message.

   Example query results

   > Note
   >
   > An ellipsis () at the bottom of the table indicates that the entire list is not displayedclick it to show more results.

6. If any matching code is found, click a link in the `w` column to view it in the code viewer.

   The matching `w` is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import go`	Imports the standard CodeQL libraries for Go.	Every query begins with one or more `import` statements.
`from Method m, Variable recv, Write w, Field f`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We declare: • `m` as a variable for all methods • a `recv` variable, which is the receiver of `m` • `w` as the location in the code where the receiver is modified • `f` as the field that is written when `m` is called
`where recv = m.getReceiver() and w.writesField(recv.getARead(), f, _) and not recv.getType() instanceof PointerType`	Defines a condition on the variables.	`recv = m.getReceiver()` states that `recv` must be the receiver variable of `m`. `w.writesField(recv.getARead(), f, _)` states that `w` must be a location in the code where field `f` of `recv` is modified. We use a dont-care expression `_` for the value that is written to `f`the actual value doesnt matter in this query. `not recv.getType() instanceof PointerType` states that `m` is not a pointer method.
`select w, "This update to " + f + " has no effect, because " + recv + " is not a pointer."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports `w` with a message that explains the potential problem.

5.1.3 Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Among the results generated by the first iteration of this query, you can find cases where a value method is called but the receiver variable is returned. In such cases, the change to the receiver is not invisible to the caller, so a pointer method is not required. These are false positive results and you can improve the query by adding an extra condition to remove them.

To exclude these values:

1. Extend the where clause to include the following extra condition:

```
not exists(ReturnStmt ret | ret.getExpr() = recv.getARead().asExpr())
```

   The `where` clause is now:

```
where e.isPure() and
  recv = m.getReceiver() and
  w.writesField(recv.getARead(), f, _) and
  not recv.getType() instanceof PointerType and
  not exists(ReturnStmt ret | ret.getExpr() = recv.getARead().asExpr())
```

2. Click **Run**.

   There are now fewer results because value methods that return their receiver variable are no longer reported.

See this in the query console

5.1.4 Further reading

- CodeQL queries for Go
- Example queries for Go
- CodeQL library reference for Go
- QL language reference
- CodeQL tools

5.2 CodeQL library for Go

When youre analyzing a Go program, you can make use of the large collection of classes in the CodeQL library for Go.

5.2.1 Overview

CodeQL ships with an extensive library for analyzing Go code. The classes in this library present the data from a CodeQL database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `go.qll` imports most other standard library modules, so you can include the complete library by beginning your query with:

```
import go
```

Broadly speaking, the CodeQL library for Go provides two views of a Go code base: at the *syntactic level*, source code is represented as an abstract syntax tree (AST), while at the *data-flow level* it is represented as a data-flow graph (DFG). In between, there is also an intermediate representation of the program as a control-flow graph

(CFG), though this representation is rarely useful on its own and mostly used to construct the higher-level DFG representation.

The AST representation captures the syntactic structure of the program. You can use it to reason about syntactic properties such as the nesting of statements within each other, but also about the types of expressions and which variable a name refers to.

The DFG, on the other hand, provides an approximation of how data flows through variables and operations at runtime. It is used, for example, by the security queries to model the way user-controlled input can propagate through the program. Additionally, the DFG contains information about which function may be invoked by a given call (taking virtual dispatch through interfaces into account), as well as control-flow information about the order in which different operations may be executed at runtime.

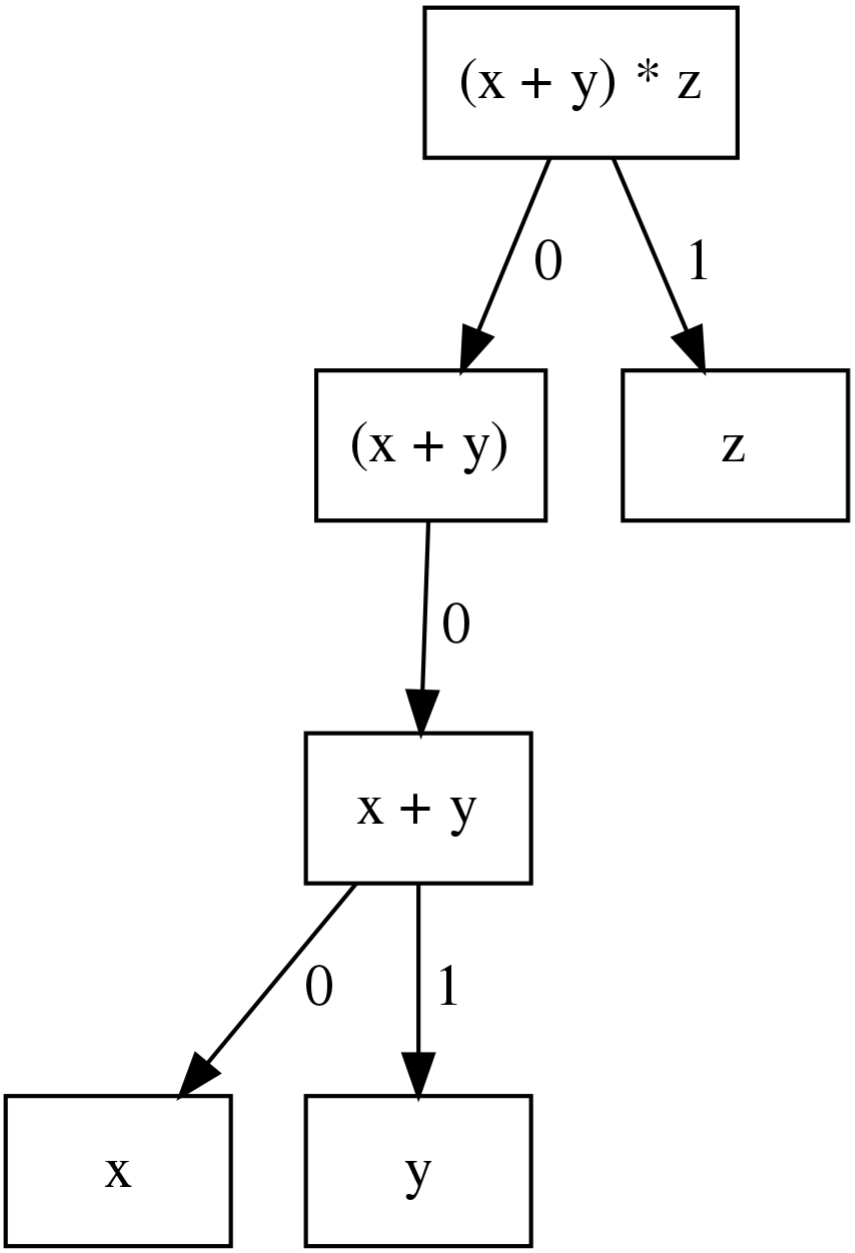
As a rule of thumb, you normally want to use the AST only for superficial syntactic queries. Any analysis involving deeper semantic properties of the program should be done on the DFG.

The rest of this tutorial briefly summarizes the most important classes and predicates provided by this library, including references to the detailed API documentation where applicable. We start by giving an overview of the AST representation, followed by an explanation of names and entities, which are used to represent name-binding information, and of types and type information. Then we move on to control flow and the data-flow graph, and finally the call graph and a few advanced topics.

5.2.2 Abstract syntax

The AST presents the program as a hierarchical structure of nodes, each of which corresponds to a syntactic element of the program source text. For example, there is an AST node for each expression and each statement in the program. These AST nodes are arranged into a parent-child relationship reflecting the nesting of syntactic elements and the order in which inner elements appear in enclosing ones.

For example, this is the AST for the expression `(x + y) * z`:



It is composed of six AST nodes, representing `x`, `y`, `x + y`, `(x + y)`, `z` and the entire expression `(x + y) * z`, respectively. The AST nodes representing `x` and `y` are children of the AST node representing `x + y`, `x` being the zeroth child and `y` being the first child, reflecting their order in the program text. Similarly, `x + y` is the only child of `(x + y)`, which is the zeroth child of `(x + y) * z`, whose first child is `z`.

All AST nodes belong to class AstNode, which defines generic tree traversal predicates:

- `getChild(i)`: returns the `i`th child of this AST node.
- `getAChild()`: returns any child of this AST node.
- `getParent()`: returns the parent node of this AST node, if any.

These predicates should only be used to perform generic AST traversal. To access children of specific AST node types, the specialized predicates introduced below should be used instead. In particular, queries should not rely

on the numeric indices of child nodes relative to their parent nodes: these are considered an implementation detail that may change between versions of the library.

The predicate `toString()` in class `AstNode` nodes gives a short description of the AST node, usually just indicating what kind of node it is. The `toString()` predicate does *not* provide access to the source text corresponding to an AST node. The source text is not stored in the dataset, and hence is not directly accessible to CodeQL queries.

The predicate `getLocation()` in class `AstNode` returns a Location entity describing the source location of the program element represented by the AST node. You can use its member predicates `getFile()`, `getStartLine()`, `getStartColumn`, `getEndLine()`, and `getEndColumn()` to obtain information about its file, start line and column, and end line and column.

The most important subclasses of AstNode are Stmt and Expr, which represent statements and expressions, respectively. This section briefly discusses some of their more important subclasses and predicates. For a full reference of all the subclasses of Stmt and Expr, see *Abstract syntax tree classes for Go*.

Statements

- `ExprStmt`: an expression statement; use `getExpr()` to access the expression itself
- `Assignment`: an assignment statement; use `getLhs(i)` to access the `i`th left-hand side and `getRhs(i)` to access the `i`th right-hand side; if there is only a single left-hand side you can use `getLhs()` instead, and similar for the right-hand side
  - `SimpleAssignStmt`: an assignment statement that does not involve a compound operator

    `AssignStmt`: a plain assignment statement of the form `lhs = rhs`

    `DefineStmt`: a short-hand variable declaration of the form `lhs := rhs`
  - `CompoundAssignStmt`: an assignment statement with a compound operator, such as `lhs += rhs`
- `IncStmt`, `DecStmt`: an increment statement or a decrement statement, respectively; use `getOperand()` to access the expression being incremented or decremented
- `BlockStmt`: a block of statements between curly braces; use `getStmt(i)` to access the `i`th statement in a block
- `IfStmt`: an `if` statement; use `getInit()`, `getCond()`, `getThen()`, and `getElse()` to access the (optional) init statement, the condition being checked, the then branch to evaluate if the condition is true, and the (optional) else branch to evaluate otherwise, respectively
- `LoopStmt`: a loop; use `getBody()` to access its body
  - `ForStmt`: a `for` statement; use `getInit()`, `getCond()`, and `getPost()` to access the init statement, loop condition, and post statement, respectively, all of which are optional
  - `RangeStmt`: a `range` statement; use `getDomain()` to access the iteration domain, and `getKey()` and `getValue()` to access the expressions to which successive keys and values are assigned, if any
- `GoStmt`: a `go` statement; use `getCall()` to access the call expression that is evaluated in the new goroutine
- `DeferStmt`: a `defer` statement; use `getCall()` to access the call expression being deferred
- `SendStmt`: a send statement; use `getChannel()` and `getValue()` to access the channel and the value being sent over the channel, respectively
- `ReturnStmt`: a `return` statement; use `getExpr(i)` to access the `i`th returned expression; if there is only a single returned expression you can use `getExpr()` instead

- `BranchStmt`: a statement that interrupts structured control flow; use `getLabel()` to get the optional target label
  - `BreakStmt`: a `break` statement
  - `ContinueStmt`: a `continue` statement
  - `FallthroughStmt`: a `fallthrough` statement at the end of a switch case
  - `GotoStmt`: a `goto` statement
- `DeclStmt`: a declaration statement, use `getDecl()` to access the declaration in this statement; note that one rarely needs to deal with declaration statements directly, since reasoning about the entities they declare is usually easier
- `SwitchStmt`: a `switch` statement; use `getInit()` to access the (optional) init statement, and `getCase(i)` to access the `i`th `case` or `default` clause
  - `ExpressionSwitchStmt`: a `switch` statement examining the value of an expression
  - `TypeSwitchStmt`: a `switch` statement examining the type of an expression
- `CaseClause`: a `case` or `default` clause in a `switch` statement; use `getExpr(i)` to access the `i`th expression, and `getStmt(i)` to access the `i`th statement in the body of this clause
- `SelectStmt`: a `select` statement; use `getCommClause(i)` to access the `i`th `case` or `default` clause
- `CommClause`: a `case` or `default` clause in a `select` statement; use `getComm()` to access the send/receive statement of this clause (not defined for `default` clauses), and `getStmt(i)` to access the `i`th statement in the body of this clause
- `RecvStmt`: a receive statement in a `case` clause of a `select` statement; use `getLhs(i)` to access the `i`th left-hand side of this statement, and `getExpr()` to access the underlying receive expression

Expressions

Class `Expression` has a predicate `isConst()` that holds if the expression is a compile-time constant. For such constant expressions, `getNumericValue()` and `getStringValue()` can be used to determine their numeric value and string value, respectively. Note that these predicates are not defined for expressions whose value cannot be determined at compile time. Also note that the result type of `getNumericValue()` is the QL type `float`. If an expression has a numeric value that cannot be represented as a QL `float`, this predicate is also not defined. In such cases, you can use `getExactValue()` to obtain a string representation of the value of the constant.

- `Ident`: an identifier; use `getName()` to access its name
- `SelectorExpr`: a selector of the form `base.sel`; use `getBase()` to access the part before the dot, and `getSelector()` for the identifier after the dot
- `BasicLit`: a literal of a basic type; subclasses `IntLit`, `FloatLit`, `ImagLit`, `RuneLit`, and `StringLit` represent various specific kinds of literals
- `FuncLit`: a function literal; use `getBody()` to access the body of the function
- `CompositeLit`: a composite literal; use `getKey(i)` and `getValue(i)` to access the `i`th key and the `i`th value, respectively
- `ParenExpr`: a parenthesized expression; use `getExpr()` to access the expression between the parentheses
- `IndexExpr`: an index expression `base[idx]`; use `getBase()` and `getIndex()` to access `base` and `idx`, respectively

- `SliceExpr`: a slice expression `base[lo:hi:max]`; use `getBase()`, `getLow()`, `getHigh()`, and `getMax()` to access `base`, `lo`, `hi`, and `max`, respectively; note that `lo`, `hi`, and `max` can be omitted, in which case the corresponding predicates are not defined
- `ConversionExpr`: a conversion expression `T(e)`; use `getTypeExpr()` and `getOperand()` to access `T` and `e`, respectively
- `TypeAssertExpr`: a type assertion `e.(T)`; use `getExpr()` and `getTypeExpr()` to access `e` and `T`, respectively
- `CallExpr`: a call expression `callee(arg0, ..., argn)`; use `getCalleeExpr()` to access `callee`, and `getArg(i)` to access the `i`th argument
- `StarExpr`: a star expression, which may be either a pointer-type expression or a pointer-dereference expression, depending on context; use `getBase()` to access the operand of the star
- `TypeExpr`: an expression that denotes a type
- `OperatorExpr`: an expression with a unary or binary operator; use `getOperator()` to access the operator
  - `UnaryExpr`: an expression with a unary operator; use `getAnOperand()` to access the operand of the operator
  - `BinaryExpr`: an expression with a binary operator; use `getLeftOperand()` and `getRightOperand()` to access the left and the right operand, respectively

    `ComparisonExpr`: a binary expression that performs a comparison, including both equality tests and relational comparisons

    - `EqualityTestExpr`: an equality test, that is, either `==` or `!=`; the predicate `getPolarity()` has result `true` for the former and `false` for the latter
    - `RelationalComparisonExpr`: a relational comparison; use `getLesserOperand()` and `getGreaterOperand()` to access the lesser and greater operand of the comparison, respectively; `isStrict()` holds if this is a strict comparison using `<` or `>`, as opposed to `<=` or `>=`

Names

While `Ident` and `SelectorExpr` are very useful classes, they are often too general: `Ident` covers all identifiers in a program, including both identifiers appearing in a declaration as well as references, and does not distinguish between names referring to packages, types, variables, constants, functions, or statement labels. Similarly, a `SelectorExpr` might refer to a package, a type, a function, or a method.

Class `Name` and its subclasses provide a more fine-grained mapping of this space, organized along the two axes of structure and namespace. In terms of structure, a name can be a `SimpleName`, meaning that it is a simple identifier (and hence an `Ident`), or it can be a `QualifiedName`, meaning that it is a qualified identifier (and hence a `SelectorExpr`). In terms of namespacing, a `Name` can be a `PackageName`, `TypeName`, `ValueName`, or `LabelName`. A `ValueName`, in turn, can be either a `ConstantName`, a `VariableName`, or a `FunctionName`, depending on what sort of entity the name refers to.

A related abstraction is provided by class `ReferenceExpr`: a reference expression is an expression that refers to a variable, a constant, a function, a field, or an element of an array or a slice. Use predicates `isLvalue()` and `isRvalue()` to determine whether a reference expression appears in a syntactic context where it is assigned to or read from, respectively.

Finally, `ValueExpr` generalizes `ReferenceExpr` to include all other kinds of expressions that can be evaluated to a value (as opposed to expressions that refer to a package, a type, or a statement label).

Functions

At the syntactic level, functions appear in two forms: in function declarations (represented by class `FuncDecl`) and as function literals (represented by class `FuncLit`). Since it is often convenient to reason about functions of either kind, these two classes share a common superclass `FuncDef`, which defines a few useful member predicates:

- `getBody()` provides access to the function body
- `getName()` gets the function name; it is undefined for function literals, which do not have a name
- `getParameter(i)` gets the `i`th parameter of the function
- `getResultVar(i)` gets the `i`th result variable of the function; if there is only one result, `getResultVar()` can be used to access it
- `getACall()` gets a data-flow node (see below) representing a call to this function

5.2.3 Entities and name binding

Not all elements of a code base can be represented as AST nodes. For example, functions defined in the standard library or in a dependency do not have a source-level definition within the source code of the program itself, and built-in functions like `len` do not have a definition at all. Hence functions cannot simplify be identified with their definition, and similarly for variables, types, and so on.

To smooth over this difference and provide a unified view of functions no matter where they are defined, the Go library introduces the concept of an *entity*. An entity is a named program element, that is, a package, a type, a constant, a variable, a field, a function, or a label. All entities belong to class `Entity`, which defines a few useful predicates:

- `getName()` gets the name of the entity
- `hasQualifiedName(pkg, n)` holds if this entity is declared in package `pkg` and has name `n`; this predicate is only defined for types, functions, and package-level variables and constants (but not for methods or local variables)
- `getDeclaration()` connects an entity to its declaring identifier, if any
- `getAReference()` gets a `Name` that refers to this entity

Conversely, class `Name` defines a predicate `getTarget()` that gets the entity to which the name refers.

Class `Entity` has several subclasses representing specific kinds of entities: `PackageEntity` for packages; `TypeEntity` for types; `ValueEntity` for constants (`Constant`), variables (`Variable`), and functions (`Function`); and `Label` for statement labels.

Class `Variable`, in turn, has a few subclasses representing specific kinds of variables: a `LocalVariable` is a variable declared in a local scope, that is, not at package level; `ReceiverVariable`, `Parameter` and `ResultVariable` describe receivers, parameters and results, respectively, and define a predicate `getFunction()` to access the corresponding function. Finally, class `Field` represents struct fields, and provides a member predicate `hasQualifiedName(pkg, tp, f)` that holds if this field has name `f` and belongs to type `tp` in package `pkg`. (Note that due to embedding the same field can belong to multiple types.)

Class `Function` has a subclass `Method` representing methods (including both interface methods and methods defined on a named type). Similar to `Field`, `Method` provides a member predicate `hasQualifiedName(pkg,`

`tp, m)` that holds if this method has name `m` and belongs to type `tp` in package `pkg`. Predicate `implements(m2)` holds if this method implements method `m2`, that is, it has the same name and signature as `m2` and it belongs to a type that implements the interface to which `m2` belongs. For any function, `getACall()` provides access to call sites that may call this function, possibly through virtual dispatch.

Finally, module `Builtin` provides a convenient way of looking up the entities corresponding to built-in functions and types. For example, `Builtin::len()` is the entity representing the built-in function `len`, `Builtin::bool()` is the `bool` type, and `Builtin::nil()` is the value `nil`.

5.2.4 Type information

Types are represented by class `Type` and its subclasses, such as `BoolType` for the built-in type `bool`; `NumericType` for the various numeric types including `IntType`, `Uint8Type`, `Float64Type` and others; `StringType` for the type `string`; `NamedType`, `ArrayType`, `SliceType`, `StructType`, `InterfaceType`, `PointerType`, `MapType`, `ChanType` for named types, arrays, slices, structs, interfaces, pointers, maps, and channels, respectively. Finally, `SignatureType` represents function types.

Note that the type `BoolType` is distinct from the entity `Builtin::bool()`: the latter views `bool` as a declared entity, the former as a type. You can, however, map from types to their corresponding entity (if any) using the predicate `getEntity()`.

Class `Expr` and class `Entity` both define a predicate `getType()` to determine the type of an expression or entity. If the type of an expression or entity cannot be determined (for example because some dependency could not be found during extraction), it will be associated with an invalid type of class `InvalidType`.

5.2.5 Control flow

Most CodeQL query writers will rarely use the control-flow representation of a program directly, but it is nevertheless useful to understand how it works.

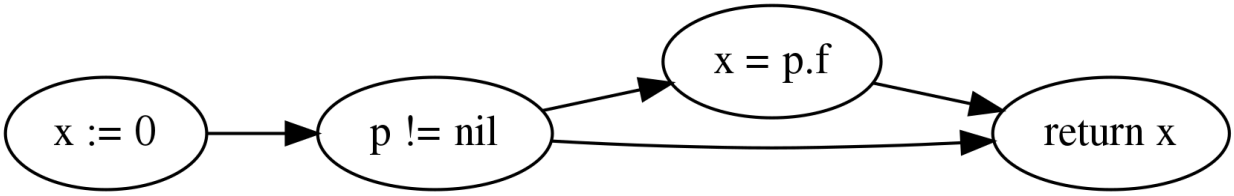
Unlike the abstract syntax tree, which views the program as a hierarchy of AST nodes, the control-flow graph views it as a collection of *control-flow nodes*, each representing a single operation performed at runtime. These nodes are connected to each other by (directed) edges representing the order in which operations are performed.

For example, consider the following code snippet:

```
x := 0
if p != nil {
  x = p.f
}
return x
```

In the AST, this is represented as an `IfStmt` and a `ReturnStmt`, with the former having an `NeqExpr` and a `BlockStmt` as its children, and so on. This provides a very detailed picture of the syntactic structure of the code, but it does not immediately help us reason about the order in which the various operations such as the comparison and the assignment are performed.

In the CFG, there are nodes corresponding to `x := 0`, `p != nil`, `x = p.f`, and `return x`, as well as a few others. The edges between these nodes model the possible execution orders of these statements and expressions, and look as follows (simplified somewhat for presentational purposes):



For example, the edge from `p != nil` to `x = p.f` models the case where the comparison evaluates to `true` and the then branch is evaluated, while the edge from `p != nil` to `return x` models the case where the comparison evaluates to `false` and the then branch is skipped.

Note, in particular, that a CFG node can have multiple outgoing edges (like from `p != nil`) as well as multiple incoming edges (like into `return x`) to represent control-flow branching at runtime.

Also note that only AST nodes that perform some kind of operation on values have a corresponding CFG node. This includes expressions (such as the comparison `p != nil`), assignment statements (such as `x = p.f`) and return statements (such as `return x`), but not statements that serve a purely syntactic purpose (such as block statements) and statements whose semantics is already reflected by the CFG edges (such as `if` statements).

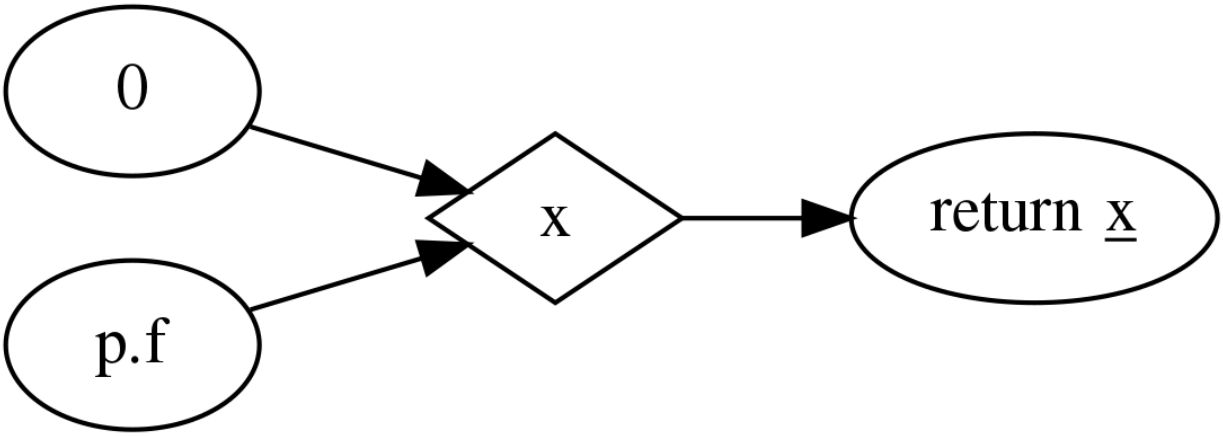
It is important to point out that the control-flow graph provided by the CodeQL libraries for Go only models *local* control flow, that is, flow within a single function. Flow from function calls to the function they invoke, for example, is not represented by control-flow edges.

In CodeQL, control-flow nodes are represented by class `ControlFlow::Node`, and the edges between nodes are captured by the member predicates `getASuccessor()` and `getAPredecessor()` of `ControlFlow::Node`. In addition to control-flow nodes representing runtime operations, each function also has a synthetic entry node and an exit node, representing the start and end of an execution of the function, respectively. These exist to ensure that the control-flow graph corresponding to a function has a unique entry node and a unique exit node, which is required for many standard control-flow analysis algorithms.

5.2.6 Data flow

At the data-flow level, the program is thought of as a collection of *data-flow nodes*. These nodes are connected to each other by (directed) edges representing the way data flows through the program at runtime.

For example, there are data-flow nodes corresponding to expressions and other data-flow nodes corresponding to variables (SSA variables, to be precise). Here is the data-flow graph corresponding to the code snippet shown above, ignoring SSA conversion for simplicity:



Note that unlike in the control-flow graph, the assignments `x := 0` and `x = p.f` are not represented as nodes. Instead, they are expressed as edges between the node representing the right-hand side of the assignment and the node representing the variable on the left-hand side. For any subsequent uses of that variable, there is a data-flow edge from the variable to that use, so by following the edges in the data-flow graph we can trace the flow of values through variables at runtime.

It is important to point out that the data-flow graph provided by the CodeQL libraries for Go only models *local* flow, that is, flow within a single function. Flow from arguments in a function call to the corresponding function parameters, for example, is not represented by data-flow edges.

In CodeQL, data-flow nodes are represented by class `DataFlow::Node`, and the edges between nodes are captured by the predicate `DataFlow::localFlowStep`. The predicate `DataFlow::localFlow` generalizes this from a single flow step to zero or more flow steps.

Most expressions have a corresponding data-flow node; exceptions include type expressions, statement labels and other expressions that do not have a value, as well as short-circuiting operators. To map from the AST node of an expression to the corresponding DFG node, use `DataFlow::exprNode`. Note that the AST node and the DFG node are different entities and cannot be used interchangeably.

There is also a predicate `asExpr()` on `DataFlow::Node` that allows you to recover the expression underlying a DFG node. However, this predicate should be used with caution, since many data-flow nodes do not correspond to an expression, and so this predicate will not be defined for them.

Similar to `Expr`, `DataFlow::Node` has a member predicate `getType()` to determine the type of a node, as well as predicates `getNumericValue()`, `getStringValue()`, and `getExactValue()` to retrieve the value of a node if it is constant.

Important subclasses of `DataFlow::Node` include:

- `DataFlow::CallNode`: a function call or method call; use `getArgument(i)` and `getResult(i)` to obtain the data-flow nodes corresponding to the `i`th argument and the `i`th result of this call, respectively; if there is only a single result, `getResult()` will return it
- `DataFlow::ParameterNode`: a parameter of a function; use `asParameter()` to access the corresponding AST node
- `DataFlow::BinaryOperationNode`: an operation involving a binary operator; each `BinaryExpr` has a corresponding `BinaryOperationNode`, but there are also binary operations that are not explicit at the AST level, such as those arising from compound assignments and increment/decrement statements; at the AST level, `x + 1`, `x += 1`, and `x++` are represented by different kinds of AST nodes, while at the DFG level they are all modeled as a binary operation node with operands `x` and `1`
- `DataFlow::UnaryOperationNode`: analogous, but for unary operators
  - `DataFlow::PointerDereferenceNode`: a pointer dereference, either explicit in an expression of the form `*p`, or implicit in a field or method reference through a pointer
  - `DataFlow::AddressOperationNode`: analogous, but for taking the address of an entity
  - `DataFlow::RelationalComparisonNode`, `DataFlow::EqualityTestNode`: data-flow nodes corresponding to `RelationalComparisonExpr` and `EqualityTestExpr` AST nodes

Finally, classes `Read` and `Write` represent, respectively, a read or a write of a variable, a field, or an element of an array, a slice or a map. Use their member predicates `readsVariable`, `writesVariable`, `readsField`, `writesField`, `readsElement`, and `writesElement` to determine what the read/write refers to.

5.2.7 Call graph

The call graph connects function (and method) calls to the functions they invoke. Call graph information is made available by two member predicates on `DataFlow::CallNode`: `getTarget()` returns the declared target of a call, while `getACallee()` returns all possible actual functions a call may invoke at runtime.

These two predicates differ in how they handle calls to interface methods: while `getTarget()` will return the interface method itself, `getACallee()` will return all concrete methods that implement the interface method.

5.2.8 Global data flow and taint tracking

The predicates `DataFlow::localFlowStep` and `DataFlow::localFlow` are useful for reasoning about the flow of values in a single function. However, more advanced use cases, particularly in security analysis, will invariably require reasoning about global data flow, including flow into, out of, and across function calls, and through fields.

In CodeQL, such reasoning is expressed in terms of *data-flow configurations*. A data-flow configuration has three ingredients: sources, sinks, and barriers (also called sanitizers), all of which are sets of data-flow nodes. Given these three sets, CodeQL provides a general mechanism for finding paths from a source to a sink, possibly going into and out of functions and fields, but never flowing through a barrier.

To define a data-flow configuration, you can define a subclass of `DataFlow::Configuration`, overriding the member predicates `isSource`, `isSink`, and `isBarrier` to define the sets of sources, sinks, and barriers.

Going beyond pure data flow, many security analyses need to perform more general *taint tracking*, which also considers flow through value-transforming operations such as string operations. To track taint, you can define a subclass of `TaintTracking::Configuration`, which works similar to data-flow configurations.

A detailed exposition of global data flow and taint tracking is out of scope for this brief introduction. For a general overview of data flow and taint tracking, see About data flow analysis.

5.2.9 Advanced libraries

Finally, we briefly describe a few concepts and libraries that are useful for advanced query writers.

Basic blocks and dominance

Many important control-flow analyses organize control-flow nodes into basic blocks, which are maximal straight-line sequences of control-flow nodes without any branching. In the CodeQL libraries, basic blocks are represented by class `BasicBlock`. Each control-flow node belongs to a basic block. You can use the predicate `getBasicBlock()` in class `ControlFlow::Node` and the predicate `getNode(i)` in `BasicBlock` to move from one to the other.

Dominance is a standard concept in control-flow analysis: a basic block `dom` is said to *dominate* a basic block `bb` if any path through the control-flow graph from the entry node to the first node of `bb` must pass through `dom`. In other words, whenever program execution reaches the beginning of `bb`, it must have come through `dom`. Each basic block is moreover considered to dominate itself.

Dually, a basic block `postdom` is said to *post-dominate* a basic block `bb` if any path through the control-flow graph from the last node of `bb` to the exit node must pass through `postdom`. In other words, after program execution leaves `bb`, it must eventually reach `postdom`.

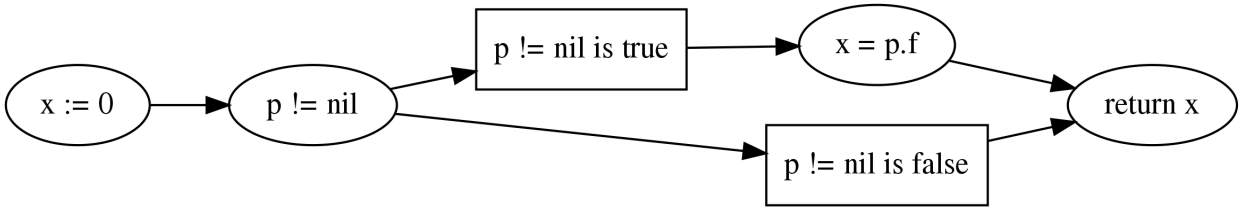
These two concepts are captured by two member predicates `dominates` and `postDominates` of class `BasicBlock`.

Condition guard nodes

A condition guard node is a synthetic control-flow node that records the fact that at some point in the control-flow graph the truth value of a condition is known. For example, consider again the code snippet we saw above:

```
x := 0
if p != nil {
  x = p.f
}
return x
```

At the beginning of the then branch `p` is known not be `nil`. This knowledge is encoded in the control-flow graph by a condition guard node preceding the assignment to `x`, recording the fact that `p != nil` is `true` at this point:



A typical use of this information would be in an analysis that looks for `nil` dereferences: such an analysis would be able to conclude that the field read `p.f` is safe because it is immediately preceded by a condition guard node guaranteeing that `p` is not `nil`.

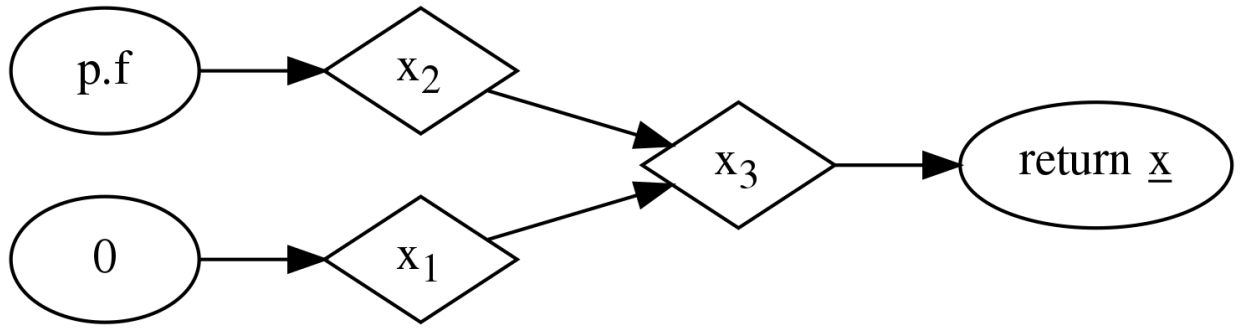
In CodeQL, condition guard nodes are represented by class `ControlFlow::ConditionGuardNode` which offers a variety of member predicates to reason about which conditions a guard node guarantees.

Static single-assignment form

Static single-assignment form (SSA form for short) is a program representation in which the original program variables are mapped onto more fine-grained *SSA variables*. Each SSA variable has exactly one definition, so program variables with multiple assignments correspond to multiple SSA variables.

Most of the time query authors do not have to deal with SSA form directly. The data-flow graph uses it under the hood, and so most of the benefits derived from SSA can be gained by simply using the data-flow graph.

For example, the data-flow graph for our running example actually looks more like this:



Note that the program variable `x` has been mapped onto three distinct SSA variables `x1`, `x2`, and `x3`. In this case there is not much benefit to such a representation, but in general SSA form has well-known advantages for data-flow analysis for which we refer to the literature.

If you do need to work with raw SSA variables, they are represented by the class `SsaVariable`. Class `SsaDefinition` represents definitions of SSA variables, which have a one-to-one correspondence with `SsaVariables`. Member predicates `getDefinition()` and `getVariable()` exist to map from one to the other. You can use member predicate `getAUse()` of `SsaVariable` to look for uses of an SSA variable. To access the program variable underlying an SSA variable, use member predicate `getSourceVariable()`.

Global value numbering

Global value numbering is a technique for determining when two computations in a program are guaranteed to yield the same result. This is done by associating with each data-flow node an abstract representation of its value (conventionally called a *value number*, even though in practice it is not usually a number) such that identical computations are represented by identical value numbers.

Since this is an undecidable problem, global value numbering is *conservative* in the sense that if two data-flow nodes have the same value number they are guaranteed to have the same value at runtime, but not conversely. (That is, there may be data-flow nodes that do, in fact, always evaluate to the same value, but their value numbers are different.)

In the CodeQL libraries for Go, you can use the `globalValueNumber(nd)` predicate to compute the global value number for a data-flow node `nd`. Value numbers are represented as an opaque QL type `GVN` that provides very little information. Usually, all you need to do with global value numbers is to compare them to each other to determine whether two data-flow nodes have the same value.

5.2.10 Further reading

- CodeQL queries for Go
- Example queries for Go
- CodeQL library reference for Go
- QL language reference
- CodeQL tools

5.3 Abstract syntax tree classes for working with Go programs

CodeQL has a large selection of classes for representing the abstract syntax tree of Go programs.

The abstract syntax tree (AST) represents the syntactic structure of a program. Nodes on the AST represent elements such as statements and expressions.

5.3.1 Statement classes

This table lists all subclasses of Stmt.

Statement syntax	CodeQL class	Superclasses	Remar
;	EmptyStmt		
Expr	ExprStmt		
{ Stmt ... }	BlockStmt		

Table 1 – continued from previous page

Statement syntax	CodeQL class	Superclasses	Remar
`if` Expr BlockStmt	IfStmt		
`if` Expr BlockStmt `else` Stmt			
`if` Stmt `;` Expr BlockStmt			
`for` Expr BlockStmt	ForStmt	LoopStmt	
`for` Stmt `;` Expr `;` Stmt BlockStmt			
`for` Expr `...` `=` `range` Expr BlockStmt	RangeStmt	LoopStmt	
`switch` Expr `{` CaseClause `...` `}`	ExpressionSwitchStmt	SwitchStmt	
`switch` Stmt `;` Expr `{` CaseClause `...` `}`			
`switch` Expr`.(type)` `{` CaseClause `...` `}`	TypeSwitchStmt	SwitchStmt	
`switch` SimpleAssignStmt`.(type)` `{` CaseClause `...` `}`			
`switch` Stmt `;` Expr`.(type)` `{` CaseClause `...` `}`			
`select` `{` CommClause `...` `}`	SelectStmt		
`return`	ReturnStmt		
`return` Expr `...`			
`break`	BreakStmt	BranchStmt	
`break` LabelName			
`continue`	ContinueStmt	BranchStmt	
`continue` LabelName			
`goto` LabelName	GotoStmt	BranchStmt	
`fallthrough`	FallthroughStmt	BranchStmt	can on
LabelName`:` Stmt	LabeledStmt		
`var` VariableName TypeName	DeclStmt		
`const` VariableName `=` Expr			
`type` TypeName TypeExpr			
`type` TypeName `=` TypeExpr			
Expr `...` `=` Expr `...`	AssignStmt	SimpleAssignStmt, Assignment	
VariableName `...` `:=` Expr `...`	DefineStmt	SimpleAssignStmt, Assignment	
Expr `+=` Expr	AddAssignStmt	CompoundAssignStmt, Assignment	
Expr `-=` Expr	SubAssignStmt	CompoundAssignStmt, Assignment	
Expr `*=` Expr	MulAssignStmt	CompoundAssignStmt, Assignment	
Expr `/=` Expr	QuoAssignStmt	CompoundAssignStmt, Assignment	
Expr `%=` Expr	RemAssignStmt	CompoundAssignStmt, Assignment	
Expr `*=` Expr	MulAssignStmt	CompoundAssignStmt, Assignment	
Expr `&=` Expr	AndAssignStmt	CompoundAssignStmt, Assignment	
Expr `|=` Expr	OrAssignStmt	CompoundAssignStmt, Assignment	
Expr `^=` Expr	XorAssignStmt	CompoundAssignStmt, Assignment	
Expr `<<=` Expr	ShlAssignStmt	CompoundAssignStmt, Assignment	
Expr `>>=` Expr	ShrAssignStmt	CompoundAssignStmt, Assignment	
Expr `&^=` Expr	AndNotAssignStmt	CompoundAssignStmt, Assignment	
Expr `++`	IncStmt	IncDecStmt	
Expr `--`	DecStmt	IncDecStmt	
`go` CallExpr	GoStmt		
`defer` CallExpr	DeferStmt		
Expr `<-` Expr	SendStmt		
`case` Expr `...:` Stmt `...`	CaseClause		can on

a Swit

Table 1 – continued from previous page

Statement syntax	CodeQL class	Superclasses	Remar
`case` TypeExpr `...:` Stmt `...`			
`default:` Stmt `...`			
`case` SendStmt`:` Stmt `...`	CommClause		can on
`case` RecvStmt`:` Stmt `...`			a Selec
`default:` Stmt `...`			
Expr `...` = RecvExpr	RecvStmt		can on
VariableName `...` := RecvExpr			a Com
(anything unparseable)	BadStmt		

5.3.2 Expression classes

There are many expression classes, so we present them by category. All classes in this section are subclasses of Expr.

Literals

Expression syntax example	CodeQL class	Superclass
`23`	IntLit	BasicLit
`4.2`	FloatLit	BasicLit
`4.2 + 2.7i`	ImagLit	BasicLit
`'a'`	CharLit	BasicLit
`"Hello"`	StringLit	BasicLit
`func(x, y int) int { return x + y }`	FuncLit	FuncDef
`map[string]int{"A": 1, "B": 2}`	MapLit	CompositeLit
`Point3D{0.5, -0.5, 0.5}`	StructLit	CompositeLit

Unary expressions

All classes in this subsection are subclasses of UnaryExpr.

Expression syntax	CodeQL class	Superclasses
`+`Expr	PlusExpr	ArithmeticUnaryExpr
`-`Expr	MinusExpr	ArithmeticUnaryExpr
`!`Expr	NotExpr	LogicalUnaryExpr
`^`Expr	ComplementExpr	BitwiseUnaryExpr
`&`Expr	AddressExpr	
`<-`Expr	RecvExpr	

Binary expressions

All classes in this subsection are subclasses of BinaryExpr.

Expression syntax	CodeQL class	Superclasses
Expr * Expr	MulExpr	ArithmeticBinaryExpr
Expr / Expr	QuoExpr	ArithmeticBinaryExpr
Expr % Expr	RemExpr	ArithmeticBinaryExpr
Expr + Expr	AddExpr	ArithmeticBinaryExpr
Expr - Expr	SubExpr	ArithmeticBinaryExpr
Expr << Expr	ShlExpr	ShiftExpr
Expr >> Expr	ShrExpr	ShiftExpr
Expr && Expr	LandExpr	LogicalBinaryExpr
Expr || Expr	LorExpr	LogicalBinaryExpr
Expr < Expr	LssExpr	RelationalComparisonExpr
Expr > Expr	GtrExpr	RelationalComparisonExpr
Expr <= Expr	LeqExpr	RelationalComparisonExpr
Expr >= Expr	GeqExpr	RelationalComparisonExpr
Expr == Expr	EqlExpr	EqualityTestExpr
Expr != Expr	NeqExpr	EqualityTestExpr
Expr & Expr	AndExpr	BitwiseBinaryExpr
Expr | Expr	OrExpr	BitwiseBinaryExpr
Expr ^ Expr	XorExpr	BitwiseBinaryExpr
Expr &^ Expr	AndNotExpr	BitwiseBinaryExpr

Type expressions

These classes represent different expressions for types. They do not have a common superclass.

Expression syntax	CodeQL class	Superclasses
`[`Expr`]` TypeExpr	ArrayTypeExpr	
`struct { ... }`	StructTypeExpr	
`func` FunctionName`(...) (...)`	FuncTypeExpr	
`interface { ... }`	InterfaceTypeExpr	
`map[`TypeExpr`]`TypeExpr	MapTypeExpr	
`chan<-` TypeExpr	SendChanTypeExpr	ChanTypeExpr
`<-chan` TypeExpr	RecvChanTypeExpr	ChanTypeExpr
`chan` TypeExpr	SendRecvChanTypeExpr	ChanTypeExpr

Name expressions

All classes in this subsection are subclasses of Name.

The following classes relate to the structure of the name.

Expression syntax	CodeQL class	Superclasses
Ident	SimpleName	Ident
Ident`.`Ident	QualifiedName	SelectorExpr

The following classes relate to what sort of entity the name refers to.

- PackageName

- TypeName
- LabelName
- ValueName
  - ConstantName
  - VariableName
  - FunctionName

Miscellaneous

Expression syntax	CodeQL class	Superclasses	Remarks
`foo`	Ident		
`_`	BlankIdent		
`...`	Ellipsis		
`(`Expr`)`	ParenExpr		
Ident`.`Ident	SelectorExpr		
Expr`[`Expr`]`	IndexExpr		
Expr`[`Expr`:`Expr`:`Expr`]`	SliceExpr		
Expr`.(`TypeExpr`)`	TypeAssertExpr		
`*`Expr	StarExpr		can be a ValueExpr or TypeExpr depending on context
Expr`:` Expr	KeyValueExpr		
TypeExpr`(`Expr`)`	ConversionExpr	CallOrConversionExpr	
Expr`(...)`	CallExpr	CallOrConversionExpr	
(anything unparseable)	BadExpr		

The following classes organize expressions by the kind of entity they refer to.

CodeQL class	Explanation
TypeExpr	an expression that denotes a type
ReferenceExpr	an expression that refers to a variable, a constant, a function, a field, or an element of an array or a slice
ValueExpr	an expression that can be evaluated to a value (as opposed to expressions that refer to a package, a type, or a statement label). This generalizes ReferenceExpr

5.3.3 Further reading

- CodeQL queries for Go
- Example queries for Go

- CodeQL library reference for Go
- QL language reference
- CodeQL tools

5.4 Modeling data flow in Go libraries

When analyzing a Go program, CodeQL does not examine the source code for external packages. To track the flow of untrusted data through a library, you can create a model of the library.

You can find existing models in the `ql/src/semmle/go/frameworks/` folder of the CodeQL for Go repository. To add a new model, you should make a new file in that folder, named after the library.

5.4.1 Sources

To mark a source of data that is controlled by an untrusted user, we create a class extending `UntrustedFlowSource::Range`. Inheritance and the characteristic predicate of the class should be used to specify exactly the dataflow node that introduces the data. Here is a short example from `Mux.qll`.

```
class RequestVars extends DataFlow::UntrustedFlowSource::Range, DataFlow::CallNode {
  RequestVars() { this.getTarget().hasQualifiedName("github.com/gorilla/mux", "Vars") }
}
```

This has the effect that all calls to the function Vars from the package mux are treated as sources of untrusted data.

5.4.2 Flow propagation

By default, we assume that all functions in libraries do not have any data flow. To indicate that a particular function does have data flow, create a class extending `TaintTracking::FunctionModel` (or `DataFlow::FunctionModel` if the untrusted user data is passed on without being modified).

Inheritance and the characteristic predicate of the class should specify the function. The class should also have a member predicate with the signature `override predicate hasTaintFlow(FunctionInput inp, FunctionOutput outp)` (or `override predicate hasDataFlow(FunctionInput inp, FunctionOutput outp)` if extending `DataFlow::FunctionModel`). The body should constrain `inp` and `outp`.

`FunctionInput` is an abstract representation of the inputs to a function. The options are:

- the receiver (`inp.isReceiver()`)
- one of the parameters (`inp.isParameter(i)`)
- one of the results (`inp.isResult(i)`, or `inp.isResult` if there is only one result)

Note that it may seem strange that the result of a function could be considered as a function input, but it is needed in some cases. For instance, the function `bufio.NewWriter` returns a writer `bw` that buffers write operations to an underlying writer `w`. If tainted data is written to `bw`, then it makes sense to propagate that taint back to the underlying writer `w`, which can be modeled by saying that `bufio.NewWriter` propagates taint from its result to its first argument.

Similarly, `FunctionOutput` is an abstract representation of the outputs to a function. The options are:

- the receiver (`outp.isReceiver()`)

- one of the parameters (`outp.isParameter(i)`)
- one of the results (`outp.isResult(i)`, or `outp.isResult` if there is only one result)

Here is an example from `Gin.qll`, which has been slightly simplified.

```
private class ParamsGet extends TaintTracking::FunctionModel, Method {
  ParamsGet() { this.hasQualifiedName("github.com/gin-gonic/gin", "Params", "Get") }

  override predicate hasTaintFlow(FunctionInput inp, FunctionOutput outp) {
    inp.isReceiver() and outp.isResult(0)
  }
}
```

This has the effect that calls to the `Get` method with receiver type `Params` from the `gin-gonic/gin` package allow taint to flow from the receiver to the first result. In other words, if `p` has type `Params` and taint can flow to it, then after the line `x := p.Get("foo")` taint can also flow to `x`.

5.4.3 Sanitizers

It is not necessary to indicate that library functions are sanitizers. Their bodies are not analyzed, so it is assumed that data does not flow through them.

5.4.4 Sinks

Data-flow sinks are specified by queries rather than by library models. However, you can use library models to indicate when functions belong to special categories. Queries can then use these categories when specifying sinks. Classes representing these special categories are contained in `ql/src/semmle/go/Concepts.qll` in the CodeQL for Go repository. `Concepts.qll` includes classes for logger mechanisms, HTTP response writers, HTTP redirects, and marshaling and unmarshaling functions.

Here is a short example from `Stdlib.qll`, which has been slightly simplified.

```
private class PrintfCall extends LoggerCall::Range, DataFlow::CallNode {
  PrintfCall() { this.getTarget().hasQualifiedName("fmt", ["Print", "Printf", "Println"]) }

  override DataFlow::Node getAMessageComponent() { result = this.getAnArgument() }
}
```

This has the effect that any call to `Print`, `Printf`, or `Println` in the package `fmt` is recognized as a logger call. Any query that uses logger calls as a sink will then identify when tainted data has been passed as an argument to `Print`, `Printf`, or `Println`.

- *Basic query for Go code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for Go*: When youre analyzing a Go program, you can make use of the large collection of classes in the CodeQL library for Go.
- *Abstract syntax tree classes for working with Go programs*: CodeQL has a large selection of classes for representing the abstract syntax tree of Go programs.
- *Modeling data flow in Go libraries*: When analyzing a Go program, CodeQL does not examine the source code for external packages. To track the flow of untrusted data through a library, you can create a model of the library.

CHAPTER

SIX

CODEQL FOR JAVA

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Java codebases.

6.1 Basic query for Java code

Learn to write and run a simple CodeQL query using LGTM.

6.1.1 About the query

The query were going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they have an empty then branch. For example, code such as:

```
if (error) { }
```

6.1.2 Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Java** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import java

from IfStmt ifstmt, Block block
where ifstmt.getThen() = block and
  block.getNumStmt() = 0
select ifstmt, "This 'if' statement is redundant."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis () at the bottom of the table indicates that the entire list is not displayedclick it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import java`	Imports the standard CodeQL libraries for Java.	Every query begins with one or more `import` statements.
`from IfStmt ifstmt, Block block`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `IfStmt` variable for `if` statements • a `Block` variable for the then block
`where ifstmt.getThen() = block and block.getNumStmt() = 0`	Defines a condition on the variables.	`ifstmt.getThen() = block` relates the two variables. The block must be the `then` branch of the `if` statement. `block.getNumStmt() = 0` states that the block must be empty (that is, it contains no statements).
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

6.1.3 Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where an empty `then` branch does serve a purpose. For example:

```
if (...) {
  ...
} else if ("-verbose".equals(option)) {
  // nothing to do - handled earlier
} else {
  error("unrecognized option");
}
```

In this case, identifying the `if` statement with the empty `then` branch as redundant is a false positive. One solution to this is to modify the query to ignore empty `then` branches if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Extend the where clause to include the following extra condition:

```
and not exists(ifstmt.getElse())
```

The `where` clause is now:

```
where ifstmt.getThen() = block and
  block.getNumStmt() = 0 and
  not exists(ifstmt.getElse())
```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer included.

See this in the query console

6.1.4 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.2 CodeQL library for Java

When youre analyzing a Java program, you can make use of the large collection of classes in the CodeQL library for Java.

6.2.1 About the CodeQL library for Java

There is an extensive library for analyzing CodeQL databases extracted from Java projects. The classes in this library present the data from a database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `java.qll` imports all the core Java library modules, so you can include the complete library by beginning your query with:

```
import java
```

The rest of this article briefly summarizes the most important classes and predicates provided by this library.

> Note
>
> The example queries in this article illustrate the types of results returned by different library classes. The results themselves are not interesting but can be used as the basis for developing a more complex query. The other articles in this section of the help show how you can take a simple query and fine-tune it to find precisely the results youre interested in.

6.2.2 Summary of the library classes

The most important classes in the standard Java library can be grouped into five main categories:

1. Classes for representing program elements (such as classes and methods)
2. Classes for representing AST nodes (such as statements and expressions)
3. Classes for representing metadata (such as annotations and comments)
4. Classes for computing metrics (such as cyclomatic complexity and coupling)
5. Classes for navigating the programs call graph

We will discuss each of these in turn, briefly describing the most important classes for each category.

6.2.3 Program elements

These classes represent named program elements: packages (`Package`), compilation units (`CompilationUnit`), types (`Type`), methods (`Method`), constructors (`Constructor`), and variables (`Variable`).

Their common superclass is `Element`, which provides general member predicates for determining the name of a program element and checking whether two elements are nested inside each other.

Its often convenient to refer to an element that might either be a method or a constructor; the class `Callable`, which is a common superclass of `Method` and `Constructor`, can be used for this purpose.

Types

Class `Type` has a number of subclasses for representing different kinds of types:

- `PrimitiveType` represents a primitive type, that is, one of `boolean`, `byte`, `char`, `double`, `float`, `int`, `long`, `short`; QL also classifies `void` and `<nulltype>` (the type of the `null` literal) as primitive types.
- `RefType` represents a reference (that is, non-primitive) type; it in turn has several subclasses:
  - `Class` represents a Java class.
  - `Interface` represents a Java interface.
  - `EnumType` represents a Java `enum` type.
  - `Array` represents a Java array type.

For example, the following query finds all variables of type `int` in the program:

```
import java

from Variable v, PrimitiveType pt
where pt = v.getType() and
    pt.hasName("int")
select v
```

See this in the query console on LGTM.com. Youre likely to get many results when you run this query because most projects contain many variables of type `int`.

Reference types are also categorized according to their declaration scope:

- `TopLevelType` represents a reference type declared at the top-level of a compilation unit.

- `NestedType` is a type declared inside another type.

For instance, this query finds all top-level types whose name is not the same as that of their compilation unit:

```
import java

from TopLevelType tl
where tl.getName() != tl.getCompilationUnit().getName()
select tl
```

See this in the query console on LGTM.com. This pattern is seen in many projects. When we ran it on the LGTM.com demo projects, most of the projects had at least one instance of this problem in the source code. There were many more instances in the files referenced by the source code.

Several more specialized classes are available as well:

- `TopLevelClass` represents a class declared at the top-level of a compilation unit.
- `NestedClass` represents a class declared inside another type, such as:
  - A `LocalClass`, which is a class declared inside a method or constructor.
  - An `AnonymousClass`, which is an anonymous class.

Finally, the library also has a number of singleton classes that wrap frequently used Java standard library classes: `TypeObject`, `TypeCloneable`, `TypeRuntime`, `TypeSerializable`, `TypeString`, `TypeSystem` and `TypeClass`. Each CodeQL class represents the standard Java class suggested by its name.

As an example, we can write a query that finds all nested classes that directly extend `Object`:

```
import java

from NestedClass nc
where nc.getASupertype() instanceof TypeObject
select nc
```

See this in the query console on LGTM.com. Youre likely to get many results when you run this query because many projects include nested classes that extend `Object` directly.

Generics

There are also several subclasses of `Type` for dealing with generic types.

A `GenericType` is either a `GenericInterface` or a `GenericClass`. It represents a generic type declaration such as interface `java.util.Map` from the Java standard library:

```
package java.util.;

public interface Map<K, V> {
    int size();

    // ...
}
```

Type parameters, such as `K` and `V` in this example, are represented by class `TypeVariable`.

A parameterized instance of a generic type provides a concrete type to instantiate the type parameter with, as in `Map<String, File>`. Such a type is represented by a `ParameterizedType`, which is distinct from the `GenericType` representing the generic type it was instantiated from. To go from a `ParameterizedType` to its corresponding `GenericType`, you can use predicate `getSourceDeclaration`.

For instance, we could use the following query to find all parameterized instances of `java.util.Map`:

```
import java

from GenericInterface map, ParameterizedType pt
where map.hasQualifiedName("java.util", "Map") and
    pt.getSourceDeclaration() = map
select pt
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects contain parameterized instances of `java.util.Map` in their source code, but they all have results in reference files.

In general, generic types may restrict which types a type parameter can be bound to. For instance, a type of maps from strings to numbers could be declared as follows:

```
class StringToNumMap<N extends Number> implements Map<String, N> {
    // ...
}
```

This means that a parameterized instance of `StringToNumberMap` can only instantiate type parameter `N` with type `Number` or one of its subtypes but not, for example, with `File`. We say that N is a bounded type parameter, with `Number` as its upper bound. In QL, a type variable can be queried for its type bound using predicate `getATypeBound`. The type bounds themselves are represented by class `TypeBound`, which has a member predicate `getType` to retrieve the type the variable is bounded by.

As an example, the following query finds all type variables with type bound `Number`:

```
import java

from TypeVariable tv, TypeBound tb
where tb = tv.getATypeBound() and
    tb.getType().hasQualifiedName("java.lang", "Number")
select tv
```

See this in the query console on LGTM.com. When we ran it on the LGTM.com demo projects, the *neo4j/neo4j*, *hibernate/hibernate-orm* and *apache/hadoop* projects all contained examples of this pattern.

For dealing with legacy code that is unaware of generics, every generic type has a raw version without any type parameters. In the CodeQL libraries, raw types are represented using class `RawType`, which has the expected subclasses `RawClass` and `RawInterface`. Again, there is a predicate `getSourceDeclaration` for obtaining the corresponding generic type. As an example, we can find variables of (raw) type `Map`:

```
import java

from Variable v, RawType rt
where rt = v.getType() and
    rt.getSourceDeclaration().hasQualifiedName("java.util", "Map")
select v
```

See this in the query console on LGTM.com. Many projects have variables of raw type `Map`.

For example, in the following code snippet this query would find `m1`, but not `m2`:

```
Map m1 = new HashMap();
Map<String, String> m2 = new HashMap<String, String>();
```

Finally, variables can be declared to be of a wildcard type:

```
Map<? extends Number, ? super Float> m;
```

The wildcards `? extends Number` and `? super Float` are represented by class `WildcardTypeAccess`. Like type parameters, wildcards may have type bounds. Unlike type parameters, wildcards can have upper bounds (as in `? extends Number`), and also lower bounds (as in `? super Float`). Class `WildcardTypeAccess` provides member predicates `getUpperBound` and `getLowerBound` to retrieve the upper and lower bounds, respectively.

For dealing with generic methods, there are classes `GenericMethod`, `ParameterizedMethod` and `RawMethod`, which are entirely analogous to the like-named classes for representing generic types.

For more information on working with types, see the *article on Java types*.

Variables

Class `Variable` represents a variable in the Java sense, which is either a member field of a class (whether static or not), or a local variable, or a parameter. Consequently, there are three subclasses catering to these special cases:

- `Field` represents a Java field.
- `LocalVariableDecl` represents a local variable.
- `Parameter` represents a parameter of a method or constructor.

6.2.4 Abstract syntax tree

Classes in this category represent abstract syntax tree (AST) nodes, that is, statements (class `Stmt`) and expressions (class `Expr`). For a full list of expression and statement types available in the standard QL library, see *Abstract syntax tree classes for working with Java programs*.

Both `Expr` and `Stmt` provide member predicates for exploring the abstract syntax tree of a program:

- `Expr.getAChildExpr` returns a sub-expression of a given expression.
- `Stmt.getAChild` returns a statement or expression that is nested directly inside a given statement.
- `Expr.getParent` and `Stmt.getParent` return the parent node of an AST node.

For example, the following query finds all expressions whose parents are `return` statements:

```
import java

from Expr e
where e.getParent() instanceof ReturnStmt
select e
```

See this in the query console on LGTM.com. Many projects have examples of `return` statements with child expressions.

Therefore, if the program contains a return statement `return x + y;`, this query will return `x + y`.

As another example, the following query finds statements whose parent is an `if` statement:

```
import java

from Stmt s
where s.getParent() instanceof IfStmt
select s
```

See this in the query console on LGTM.com. Many projects have examples of `if` statements with child statements.

This query will find both `then` branches and `else` branches of all `if` statements in the program.

Finally, here is a query that finds method bodies:

```
import java

from Stmt s
where s.getParent() instanceof Method
select s
```

See this in the query console on LGTM.com. Most projects have many method bodies.

As these examples show, the parent node of an expression is not always an expression: it may also be a statement, for example, an `IfStmt`. Similarly, the parent node of a statement is not always a statement: it may also be a method or a constructor. To capture this, the QL Java library provides two abstract class `ExprParent` and `StmtParent`, the former representing any node that may be the parent node of an expression, and the latter any node that may be the parent node of a statement.

For more information on working with AST classes, see the *article on overflow-prone comparisons in Java*.

6.2.5 Metadata

Java programs have several kinds of metadata, in addition to the program code proper. In particular, there are annotations and Javadoc comments. Since this metadata is interesting both for enhancing code analysis and as an analysis subject in its own right, the QL library defines classes for accessing it.

For annotations, class `Annotatable` is a superclass of all program elements that can be annotated. This includes packages, reference types, fields, methods, constructors, and local variable declarations. For every such element, its predicate `getAnAnnotation` allows you to retrieve any annotations the element may have. For example, the following query finds all annotations on constructors:

```
import java

from Constructor c
select c.getAnAnnotation()
```

See this in the query console on LGTM.com. The LGTM.com demo projects all use annotations, you can see examples where they are used to suppress warnings and mark code as deprecated.

These annotations are represented by class `Annotation`. An annotation is simply an expression whose type is an `AnnotationType`. For example, you can amend this query so that it only reports deprecated constructors:

```
import java

from Constructor c, Annotation ann, AnnotationType anntp
where ann = c.getAnAnnotation() and
    anntp = ann.getType() and
    anntp.hasQualifiedName("java.lang", "Deprecated")
select ann
```

See this in the query console on LGTM.com. Only constructors with the `@Deprecated` annotation are reported this time.

For more information on working with annotations, see the *article on annotations*.

For Javadoc, class `Element` has a member predicate `getDoc` that returns a delegate `Documentable` object, which can then be queried for its attached Javadoc comments. For example, the following query finds Javadoc comments on private fields:

```
import java

from Field f, Javadoc jdoc
where f.isPrivate() and
    jdoc = f.getDoc().getJavadoc()
select jdoc
```

See this in the query console on LGTM.com. You can see this pattern in many projects.

Class `Javadoc` represents an entire Javadoc comment as a tree of `JavadocElement` nodes, which can be traversed using member predicates `getAChild` and `getParent`. For instance, you could edit the query so that it finds all `@author` tags in Javadoc comments on private fields:

```
import java

from Field f, Javadoc jdoc, AuthorTag at
where f.isPrivate() and
    jdoc = f.getDoc().getJavadoc() and
    at.getParent+() = jdoc
select at
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects uses the `@author` tag on private fields.

> Note
>
> On line 5 we used `getParent+` to capture tags that are nested at any depth within the Javadoc comment.

For more information on working with Javadoc, see the *article on Javadoc*.

6.2.6 Metrics

The standard QL Java library provides extensive support for computing metrics on Java program elements. To avoid overburdening the classes representing those elements with too many member predicates related to metric computations, these predicates are made available on delegate classes instead.

Altogether, there are six such classes: `MetricElement`, `MetricPackage`, `MetricRefType`, `MetricField`, `MetricCallable`, and `MetricStmt`. The corresponding element classes each provide a member predicate `getMetrics` that can be used to obtain an instance of the delegate class, on which metric computations can then be performed.

For example, the following query finds methods with a cyclomatic complexity greater than 40:

```
import java

from Method m, MetricCallable mc
where mc = m.getMetrics() and
    mc.getCyclomaticComplexity() > 40
select m
```

See this in the query console on LGTM.com. Most large projects include some methods with a very high cyclomatic complexity. These methods are likely to be difficult to understand and test.

6.2.7 Call graph

CodeQL databases generated from Java code bases include precomputed information about the programs call graph, that is, which methods or constructors a given call may dispatch to at runtime.

The class `Callable`, introduced above, includes both methods and constructors. Call expressions are abstracted using class `Call`, which includes method calls, `new` expressions, and explicit constructor calls using `this` or `super`.

We can use predicate `Call.getCallee` to find out which method or constructor a specific call expression refers to. For example, the following query finds all calls to methods called `println`:

```
import java

from Call c, Method m
where m = c.getCallee() and
    m.hasName("println")
select c
```

See this in the query console on LGTM.com. The LGTM.com demo projects all include many calls to methods of this name.

Conversely, `Callable.getAReference` returns a `Call` that refers to it. So we can find methods and constructors that are never called using this query:

```
import java

from Callable c
where not exists(c.getAReference())
select c
```

See this in the query console on LGTM.com. The LGTM.com demo projects all appear to have many methods that are not called directly, but this is unlikely to be the whole story. To explore this area further, see *Navigating the call graph*.

For more information about callables and calls, see the *article on the call graph*.

6.2.8 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.3 Analyzing data flow in Java

You can use CodeQL to track the flow of data through a Java program to its use.

6.3.1 About this article

This article describes how data flow analysis is implemented in the CodeQL libraries for Java and includes examples to help you write your own data flow queries. The following sections describe how to use the libraries for local data flow, global data flow, and taint tracking.

For a more general introduction to modeling data flow, see *About data flow analysis*.

6.3.2 Local data flow

Local data flow is data flow within a single method or callable. Local data flow is usually easier, faster, and more precise than global data flow, and is sufficient for many queries.

Using local data flow

The local data flow library is in the module `DataFlow`, which defines the class `Node` denoting any element that data can flow through. `Node`s are divided into expression nodes (`ExprNode`) and parameter nodes (`ParameterNode`). You can map between data flow nodes and expressions/parameters using the member predicates `asExpr` and `asParameter`:

```
class Node {
  /** Gets the expression corresponding to this node, if any. */
  Expr asExpr() { ... }

  /** Gets the parameter corresponding to this node, if any. */
  Parameter asParameter() { ... }

  ...
}
```

or using the predicates `exprNode` and `parameterNode`:

```
/**
 * Gets the node corresponding to expression `e`.
 */
ExprNode exprNode(Expr e) { ... }
```

(continues on next page)

(continued from previous page)

```
/**
 * Gets the node corresponding to the value of parameter `p` at function entry.
 */
ParameterNode parameterNode(Parameter p) { ... }
```

The predicate `localFlowStep(Node nodeFrom, Node nodeTo)` holds if there is an immediate data flow edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively by using the `+` and `*` operators, or by using the predefined recursive predicate `localFlow`, which is equivalent to `localFlowStep*`.

For example, you can find flow from a parameter `source` to an expression `sink` in zero or more local steps:

```
DataFlow::localFlow(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Using local taint tracking

Local taint tracking extends local data flow by including non-value-preserving flow steps. For example:

```
String temp = x;
String y = temp + ", " + temp;
```

If `x` is a tainted string then `y` is also tainted.

The local taint tracking library is in the module `TaintTracking`. Like local data flow, a predicate `localTaintStep(DataFlow::Node nodeFrom, DataFlow::Node nodeTo)` holds if there is an immediate taint propagation edge from the node `nodeFrom` to the node `nodeTo`. You can apply the predicate recursively by using the `+` and `*` operators, or by using the predefined recursive predicate `localTaint`, which is equivalent to `localTaintStep*`.

For example, you can find taint propagation from a parameter `source` to an expression `sink` in zero or more local steps:

```
TaintTracking::localTaint(DataFlow::parameterNode(source), DataFlow::exprNode(sink))
```

Examples

This query finds the filename passed to `new FileReader(..)`.

```
import java

from Constructor fileReader, Call call
where
  fileReader.getDeclaringType().hasQualifiedName("java.io", "FileReader") and
  call.getCallee() = fileReader
select call.getArgument(0)
```

Unfortunately, this only gives the expression in the argument, not the values which could be passed to it. So we use local data flow to find all expressions that flow into the argument:

```
import java
import semmle.code.java.dataflow.DataFlow
```

(continues on next page)

(continued from previous page)

```
from Constructor fileReader, Call call, Expr src
where
  fileReader.getDeclaringType().hasQualifiedName("java.io", "FileReader") and
  call.getCallee() = fileReader and
  DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(call.getArgument(0)))
select src
```

Then we can make the source more specific, for example an access to a public parameter. This query finds where a public parameter is passed to `new FileReader(..)`:

```
import java
import semmle.code.java.dataflow.DataFlow

from Constructor fileReader, Call call, Parameter p
where
  fileReader.getDeclaringType().hasQualifiedName("java.io", "FileReader") and
  call.getCallee() = fileReader and
  DataFlow::localFlow(DataFlow::parameterNode(p), DataFlow::exprNode(call.getArgument(0)))
select p
```

This query finds calls to formatting functions where the format string is not hard-coded.

```
import java
import semmle.code.java.dataflow.DataFlow
import semmle.code.java.StringFormat

from StringFormatMethod format, MethodAccess call, Expr formatString
where
  call.getMethod() = format and
  call.getArgument(format.getFormatStringIndex()) = formatString and
  not exists(DataFlow::Node source, DataFlow::Node sink |
    DataFlow::localFlow(source, sink) and
    source.asExpr() instanceof StringLiteral and
    sink.asExpr() = formatString
  )
select call, "Argument to String format method isn't hard-coded."
```

Exercises

Exercise 1: Write a query that finds all hard-coded strings used to create a `java.net.URL`, using local data flow. (*Answer*)

6.3.3 Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow, and the analysis typically requires significantly more time and memory to perform.

Note

You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see Creating path queries.

Using global data flow

You use the global data flow library by extending the class `DataFlow::Configuration`:

```
import semmle.code.java.dataflow.DataFlow

class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource`defines where data may flow from
- `isSink`defines where data may flow to
- `isBarrier`optional, restricts the data flow
- `isAdditionalFlowStep`optional, adds additional flow steps

The characteristic predicate `MyDataFlowConfiguration()` defines the name of the configuration, so `"MyDataFlowConfiguration"` should be a unique name, for example, the name of your class.

The data flow analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`:

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Data flow to $@.", sink, sink.toString()
```

Using global taint tracking

Global taint tracking is to global data flow as local taint tracking is to local data flow. That is, global taint tracking extends global data flow with additional non-value-preserving steps. You use the global taint tracking library by extending the class `TaintTracking::Configuration`:

```
import semmle.code.java.dataflow.TaintTracking

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

  override predicate isSource(DataFlow::Node source) {
```

(continues on next page)

(continued from previous page)

```
    ...
  }

  override predicate isSink(DataFlow::Node sink) {
    ...
  }
}
```

These predicates are defined in the configuration:

- `isSource`defines where taint may flow from
- `isSink`defines where taint may flow to
- `isSanitizer`optional, restricts the taint flow
- `isAdditionalTaintStep`optional, adds additional taint steps

Similar to global data flow, the characteristic predicate `MyTaintTrackingConfiguration()` defines the unique name of the configuration.

The taint tracking analysis is performed using the predicate `hasFlow(DataFlow::Node source, DataFlow::Node sink)`.

Flow sources

The data flow library contains some predefined flow sources. The class `RemoteFlowSource` (defined in `semmle.code.java.dataflow.FlowSources`) represents data flow sources that may be controlled by a remote user, which is useful for finding security problems.

Examples

This query shows a taint-tracking configuration that uses remote user input as data sources.

```
import java
import semmle.code.java.dataflow.FlowSources

class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() {
    this = "..."
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof RemoteFlowSource
  }

  ...
}
```

Exercises

Exercise 2: Write a query that finds all hard-coded strings used to create a `java.net.URL`, using global data flow. (*Answer*)

Exercise 3: Write a class that represents flow sources from `java.lang.System.getenv(..)`. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query which finds all global data flows from `getenv` to `java.net.URL`. (*Answer*)

6.3.4 Answers

Exercise 1

```
import semmle.code.java.dataflow.DataFlow

from Constructor url, Call call, StringLiteral src
where
  url.getDeclaringType().hasQualifiedName("java.net", "URL") and
  call.getCallee() = url and
  DataFlow::localFlow(DataFlow::exprNode(src), DataFlow::exprNode(call.getArgument(0)))
select src
```

Exercise 2

```
import semmle.code.java.dataflow.DataFlow

class Configuration extends DataFlow::Configuration {
  Configuration() {
    this = "LiteralToURL Configuration"
  }

  override predicate isSource(DataFlow::Node source) {
    source.asExpr() instanceof StringLiteral
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
      call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("java.net", "URL")
    )
  }
}

from DataFlow::Node src, DataFlow::Node sink, Configuration config
where config.hasFlow(src, sink)
select src, "This string constructs a URL $@.", sink, "here"
```

Exercise 3

```
import java

class GetenvSource extends MethodAccess {
  GetenvSource() {
    exists(Method m | m = this.getMethod() |
      m.hasName("getenv") and
```

(continues on next page)

(continued from previous page)

```
      m.getDeclaringType() instanceof TypeSystem
    )
  }
}
```

Exercise 4

```
import semmle.code.java.dataflow.DataFlow

class GetenvSource extends DataFlow::ExprNode {
  GetenvSource() {
    exists(Method m | m = this.asExpr().(MethodAccess).getMethod() |
      m.hasName("getenv") and
      m.getDeclaringType() instanceof TypeSystem
    )
  }
}

class GetenvToURLConfiguration extends DataFlow::Configuration {
  GetenvToURLConfiguration() {
    this = "GetenvToURLConfiguration"
  }

  override predicate isSource(DataFlow::Node source) {
    source instanceof GetenvSource
  }

  override predicate isSink(DataFlow::Node sink) {
    exists(Call call |
      sink.asExpr() = call.getArgument(0) and
      call.getCallee().(Constructor).getDeclaringType().hasQualifiedName("java.net", "URL")
    )
  }
}

from DataFlow::Node src, DataFlow::Node sink, GetenvToURLConfiguration config
where config.hasFlow(src, sink)
select src, "This environment variable constructs a URL $@.", sink, "here"
```

6.3.5 Further reading

- Exploring data flow with path queries
- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.4 Java types

You can use CodeQL to find out information about data types used in Java code. This allows you to write queries to identify specific type-related issues.

6.4.1 About working with Java types

The standard CodeQL library represents Java types by means of the `Type` class and its various subclasses.

In particular, class `PrimitiveType` represents primitive types that are built into the Java language (such as `boolean` and `int`), whereas `RefType` and its subclasses represent reference types, that is classes, interfaces, array types, and so on. This includes both types from the Java standard library (like `java.lang.Object`) and types defined by non-library code.

Class `RefType` also models the class hierarchy: member predicates `getASupertype` and `getASubtype` allow you to find a reference types immediate super types and sub types. For example, consider the following Java program:

```
class A {}

interface I {}

class B extends A implements I {}
```

Here, class `A` has exactly one immediate super type (`java.lang.Object`) and exactly one immediate sub type (`B`); the same is true of interface `I`. Class `B`, on the other hand, has two immediate super types (`A` and `I`), and no immediate sub types.

To determine ancestor types (including immediate super types, and also *their* super types, etc.), we can use transitive closure. For example, to find all ancestors of `B` in the example above, we could use the following query:

```
import java

from Class B
where B.hasName("B")
select B.getASupertype+()
```

See this in the query console on LGTM.com. If this query were run on the example snippet above, the query would return `A`, `I`, and `java.lang.Object`.

> Tip
>
> If you want to see the location of `B` as well as `A`, you can replace `B.getASupertype+()` with `B.getASupertype*()` and re-run the query.

Besides class hierarchy modeling, `RefType` also provides member predicate `getAMember` for accessing members (that is, fields, constructors, and methods) declared in the type, and predicate `inherits(Method m)` for checking whether the type either declares or inherits a method `m`.

6.4.2 Example: Finding problematic array casts

As an example of how to use the class hierarchy API, we can write a query that finds downcasts on arrays, that is, cases where an expression `e` of some type `A[]` is converted to type `B[]`, such that `B` is a (not necessarily immediate) subtype of `A`.

This kind of cast is problematic, since downcasting an array results in a runtime exception, even if every individual array element could be downcast. For example, the following code throws a `ClassCastException`:

```
Object[] o = new Object[] { "Hello", "world" };
String[] s = (String[])o;
```

If the expression `e` happens to actually evaluate to a `B[]` array, on the other hand, the cast will succeed:

```
Object[] o = new String[] { "Hello", "world" };
String[] s = (String[])o;
```

In this tutorial, we dont try to distinguish these two cases. Our query should simply look for cast expressions `ce` that cast from some type `source` to another type `target`, such that:

- Both `source` and `target` are array types.
- The element type of `source` is a transitive super type of the element type of `target`.

This recipe is not too difficult to translate into a query:

```
import java

from CastExpr ce, Array source, Array target
where source = ce.getExpr().getType() and
    target = ce.getType() and
    target.getElementType().(RefType).getASupertype+() = source.getElementType()
select ce, "Potentially problematic array downcast."
```

See this in the query console on LGTM.com. Many projects return results for this query.

Note that by casting `target.getElementType()` to a `RefType`, we eliminate all cases where the element type is a primitive type, that is, `target` is an array of primitive type: the problem we are looking for cannot arise in that case. Unlike in Java, a cast in QL never fails: if an expression cannot be cast to the desired type, it is simply excluded from the query results, which is exactly what we want.

Improvements

Running this query on old Java code, before version 5, often returns many false positive results arising from uses of the method `Collection.toArray(T[])`, which converts a collection into an array of type `T[]`.

In code that does not use generics, this method is often used in the following way:

```
List l = new ArrayList();
// add some elements of type A to l
A[] as = (A[])l.toArray(new A[0]);
```

Here, `l` has the raw type `List`, so `l.toArray` has return type `Object[]`, independent of the type of its argument array. Hence the cast goes from `Object[]` to `A[]` and will be flagged as problematic by our query, although at runtime this cast can never go wrong.

To identify these cases, we can create two CodeQL classes that represent, respectively, the `Collection.toArray` method, and calls to this method or any method that overrides it:

```
/** class representing java.util.Collection.toArray(T[]) */
class CollectionToArray extends Method {
    CollectionToArray() {
        this.getDeclaringType().hasQualifiedName("java.util", "Collection") and
        this.hasName("toArray") and
        this.getNumberOfParameters() = 1
    }
}

/** class representing calls to java.util.Collection.toArray(T[]) */
class CollectionToArrayCall extends MethodAccess {
    CollectionToArrayCall() {
        exists(CollectionToArray m |
            this.getMethod().getSourceDeclaration().overridesOrInstantiates*(m)
        )
    }

    /** the call's actual return type, as determined from its argument */
    Array getActualReturnType() {
        result = this.getArgument(0).getType()
    }
}
```

Notice the use of `getSourceDeclaration` and `overridesOrInstantiates` in the constructor of `CollectionToArrayCall`: we want to find calls to `Collection.toArray` and to any method that overrides it, as well as any parameterized instances of these methods. In our example above, for instance, the call `l.toArray` resolves to method `toArray` in the raw class `ArrayList`. Its source declaration is `toArray` in the generic class `ArrayList<T>`, which overrides `AbstractCollection<T>.toArray`, which in turn overrides `Collection<T>.toArray`, which is an instantiation of `Collection.toArray` (since the type parameter `T` in the overridden method belongs to `ArrayList` and is an instantiation of the type parameter belonging to `Collection`).

Using these new classes we can extend our query to exclude calls to `toArray` on an argument of type `A[]` which are then cast to `A[]`:

```
import java

// Insert the class definitions from above

from CastExpr ce, Array source, Array target
where source = ce.getExpr().getType() and
    target = ce.getType() and
    target.getElementType().(RefType).getASupertype+() = source.getElementType() and
    not ce.getExpr().(CollectionToArrayCall).getActualReturnType() = target
select ce, "Potentially problematic array downcast."
```

See this in the query console on LGTM.com. Notice that fewer results are found by this improved query.

6.4.3 Example: Finding mismatched contains checks

Well now develop a query that finds uses of `Collection.contains` where the type of the queried element is unrelated to the element type of the collection, which guarantees that the test will always return `false`.

For example, Apache Zookeeper used to have a snippet of code similar to the following in class `QuorumPeerConfig`:

```
Map<Object, Object> zkProp;

// ...

if (zkProp.entrySet().contains("dynamicConfigFile")){
    // ...
}
```

Since `zkProp` is a map from `Object` to `Object`, `zkProp.entrySet` returns a collection of type `Set<Entry<Object, Object>>`. Such a set cannot possibly contain an element of type `String`. (The code has since been fixed to use `zkProp.containsKey`.)

In general, we want to find calls to `Collection.contains` (or any of its overriding methods in any parameterized instance of `Collection`), such that the type `E` of collection elements and the type `A` of the argument to `contains` are unrelated, that is, they have no common subtype.

We start by creating a class that describes `java.util.Collection`:

```
class JavaUtilCollection extends GenericInterface {
    JavaUtilCollection() {
        this.hasQualifiedName("java.util", "Collection")
    }
}
```

To make sure we have not mistyped anything, we can run a simple test query:

```
from JavaUtilCollection juc
select juc
```

This query should return precisely one result.

Next, we can create a class that describes `java.util.Collection.contains`:

```
class JavaUtilCollectionContains extends Method {
    JavaUtilCollectionContains() {
        this.getDeclaringType() instanceof JavaUtilCollection and
        this.hasStringSignature("contains(Object)")
    }
}
```

Notice that we use `hasStringSignature` to check that:

- The method in question has name `contains`.
- It has exactly one argument.
- The type of the argument is `Object`.

Alternatively, we could have implemented these three checks more verbosely using `hasName`, `getNumberOfParameters`, and `getParameter(0).getType() instanceof TypeObject`.

As before, it is a good idea to test the new class by running a simple query to select all instances of `JavaUtilCollectionContains`; again there should only be a single result.

Now we want to identify all calls to `Collection.contains`, including any methods that override it, and considering all parameterized instances of `Collection` and its subclasses. That is, we are looking for method accesses where the source declaration of the invoked method (reflexively or transitively) overrides `Collection.contains`. We encode this in a CodeQL class `JavaUtilCollectionContainsCall`:

```
class JavaUtilCollectionContainsCall extends MethodAccess {
    JavaUtilCollectionContainsCall() {
        exists(JavaUtilCollectionContains jucc |
            this.getMethod().getSourceDeclaration().overrides*(jucc)
        )
    }
}
```

This definition is slightly subtle, so you should run a short query to test that `JavaUtilCollectionContainsCall` correctly identifies calls to `Collection.contains`.

For every call to `contains`, we are interested in two things: the type of the argument, and the element type of the collection on which it is invoked. So we need to add two member predicates `getArgumentType` and `getCollectionElementType` to class `JavaUtilCollectionContainsCall` to compute this information.

The former is easy:

```
Type getArgumentType() {
    result = this.getArgument(0).getType()
}
```

For the latter, we proceed as follows:

- Find the declaring type `D` of the `contains` method being invoked.
- Find a (reflexive or transitive) super type `S` of `D` that is a parameterized instance of `java.util.Collection`.
- Return the (only) type argument of `S`.

We encode this as follows:

```
Type getCollectionElementType() {
    exists(RefType D, ParameterizedInterface S |
        D = this.getMethod().getDeclaringType() and
        D.hasSupertype*(S) and S.getSourceDeclaration() instanceof JavaUtilCollection and
        result = S.getTypeArgument(0)
    )
}
```

Having added these two member predicates to `JavaUtilCollectionContainsCall`, we need to write a predicate that checks whether two given reference types have a common subtype:

```
predicate haveCommonDescendant(RefType tp1, RefType tp2) {
    exists(RefType commondesc | commondesc.hasSupertype*(tp1) and commondesc.hasSupertype*(tp2))
}
```

Now we are ready to write a first version of our query:

```
import java

// Insert the class definitions from above

from JavaUtilCollectionContainsCall juccc, Type collEltType, Type argType
where collEltType = juccc.getCollectionElementType() and argType = juccc.getArgumentType() and
    not haveCommonDescendant(collEltType, argType)
select juccc, "Element type " + collEltType + " is incompatible with argument type " + argType
```

See this in the query console on LGTM.com.

Improvements

For many programs, this query yields a large number of false positive results due to type variables and wild cards: if the collection element type is some type variable `E` and the argument type is `String`, for example, CodeQL will consider that the two have no common subtype, and our query will flag the call. An easy way to exclude such false positive results is to simply require that neither `collEltType` nor `argType` are instances of `TypeVariable`.

Another source of false positives is autoboxing of primitive types: if, for example, the collections element type is `Integer` and the argument is of type `int`, predicate `haveCommonDescendant` will fail, since `int` is not a `RefType`. To account for this, our query should check that `collEltType` is not the boxed type of `argType`.

Finally, `null` is special because its type (known as `<nulltype>` in the CodeQL library) is compatible with every reference type, so we should exclude it from consideration.

Adding these three improvements, our final query becomes:

```
import java

// Insert the class definitions from above

from JavaUtilCollectionContainsCall juccc, Type collEltType, Type argType
where collEltType = juccc.getCollectionElementType() and argType = juccc.getArgumentType() and
    not haveCommonDescendant(collEltType, argType) and
    not collEltType instanceof TypeVariable and not argType instanceof TypeVariable and
    not collEltType = argType.(PrimitiveType).getBoxedType() and
    not argType.hasName("<nulltype>")
select juccc, "Element type " + collEltType + " is incompatible with argument type " + argType
```

See the full query in the query console on LGTM.com.

6.4.4 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.5 Overflow-prone comparisons in Java

You can use CodeQL to check for comparisons in Java code where one side of the comparison is prone to overflow.

6.5.1 About this article

In this tutorial article youll write a query for finding comparisons between integers and long integers in loops that may lead to non-termination due to overflow.

To begin, consider this code snippet:

```
void foo(long l) {
    for(int i=0; i<l; i++) {
        // do something
    }
}
```

If `l` is bigger than 2^{31}- 1 (the largest positive value of type `int`), then this loop will never terminate: `i` will start at zero, being incremented all the way up to 2^{31}- 1, which is still smaller than `l`. When it is incremented once more, an arithmetic overflow occurs, and `i` becomes -2^{31}, which also is smaller than `l`! Eventually, `i` will reach zero again, and the cycle repeats.

> More about overflow
>
> All primitive numeric types have a maximum value, beyond which they will wrap around to their lowest possible value (called an overflow). For `int`, this maximum value is 2^{31}- 1. Type `long` can accommodate larger values up to a maximum of 2^{63}- 1. In this example, this means that `l` can take on a value that is higher than the maximum for type `int`; `i` will never be able to reach this value, instead overflowing and returning to a low value.

Were going to develop a query that finds code that looks like it might exhibit this kind of behavior. Well be using several of the standard library classes for representing statements and functions. For a full list, see *Abstract syntax tree classes for working with Java programs*.

6.5.2 Initial query

Well start by writing a query that finds less-than expressions (CodeQL class `LTExpr`) where the left operand is of type `int` and the right operand is of type `long`:

```
import java

from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long")
select expr
```

See this in the query console on LGTM.com. This query usually finds results on most projects.

Notice that we use the predicate `getType` (available on all subclasses of `Expr`) to determine the type of the operands. Types, in turn, define the `hasName` predicate, which allows us to identify the primitive types `int` and `long`. As it stands, this query finds *all* less-than expressions comparing `int` and `long`, but in fact we are only interested in comparisons that are part of a loop condition. Also, we want to filter out comparisons where either operand is constant, since these are less likely to be real bugs. The revised query looks like this:

```
import java

from LTExpr expr
where expr.getLeftOperand().getType().hasName("int") and
    expr.getRightOperand().getType().hasName("long") and
    exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr) and
    not expr.getAnOperand().isCompileTimeConstant()
select expr
```

See this in the query console on LGTM.com. Notice that fewer results are found.

The class `LoopStmt` is a common superclass of all loops, including, in particular, `for` loops as in our example above. While different kinds of loops have different syntax, they all have a loop condition, which can be accessed through predicate `getCondition`. We use the reflexive transitive closure operator `*` applied to the `getAChildExpr` predicate to express the requirement that `expr` should be nested inside the loop condition. In particular, it can be the loop condition itself.

The final conjunct in the `where` clause takes advantage of the fact that predicates can return more than one value (they are really relations). In particular, `getAnOperand` may return *either* operand of `expr`, so `expr.getAnOperand().isCompileTimeConstant()` holds if at least one of the operands is constant. Negating this condition means that the query will only find expressions where *neither* of the operands is constant.

6.5.3 Generalizing the query

Of course, comparisons between `int` and `long` are not the only problematic case: any less-than comparison between a narrower and a wider type is potentially suspect, and less-than-or-equals, greater-than, and greater-than-or-equals comparisons are just as problematic as less-than comparisons.

In order to compare the ranges of types, we define a predicate that returns the width (in bits) of a given integral type:

```
int width(PrimitiveType pt) {
    (pt.hasName("byte") and result=8) or
    (pt.hasName("short") and result=16) or
    (pt.hasName("char") and result=16) or
    (pt.hasName("int") and result=32) or
    (pt.hasName("long") and result=64)
}
```

We now want to generalize our query to apply to any comparison where the width of the type on the smaller end of the comparison is less than the width of the type on the greater end. Lets call such a comparison *overflow prone*, and introduce an abstract class to model it:

```
abstract class OverflowProneComparison extends ComparisonExpr {
    Expr getLesserOperand() { none() }
    Expr getGreaterOperand() { none() }
}
```

There are two concrete child classes of this class: one for <= or < comparisons, and one for >= or > comparisons. In both cases, we implement the constructor in such a way that it only matches the expressions we want:

```
class LTOverflowProneComparison extends OverflowProneComparison {
    LTOverflowProneComparison() {
        (this instanceof LEExpr or this instanceof LTExpr) and
        width(this.getLeftOperand().getType()) < width(this.getRightOperand().getType())
    }
}

class GTOverflowProneComparison extends OverflowProneComparison {
    GTOverflowProneComparison() {
        (this instanceof GEExpr or this instanceof GTExpr) and
        width(this.getRightOperand().getType()) < width(this.getLeftOperand().getType())
    }
}
```

Now we rewrite our query to make use of these new classes:

```
import Java

// Insert the class definitions from above

from OverflowProneComparison expr
where exists(LoopStmt l | l.getCondition().getAChildExpr*() = expr) and
not expr.getAnOperand().isCompileTimeConstant()
select expr
```

See the full query in the query console on LGTM.com.

6.5.4 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.6 Navigating the call graph

CodeQL has classes for identifying code that calls other code, and code that can be called from elsewhere. This allows you to find, for example, methods that are never used.

6.6.1 Call graph classes

The CodeQL library for Java provides two abstract classes for representing a programs call graph: `Callable` and `Call`. The former is simply the common superclass of `Method` and `Constructor`, the latter is a common superclass of `MethodAccess`, `ClassInstanceExpression`, `ThisConstructorInvocationStmt` and `SuperConstructorInvocationStmt`. Simply put, a `Callable` is something that can be invoked, and a `Call` is something that invokes a `Callable`.

For example, in the following program all callables and calls have been annotated with comments:

```
class Super {
    int x;

    // callable
    public Super() {
        this(23);       // call
    }

    // callable
    public Super(int x) {
        this.x = x;
    }

    // callable
    public int getX() {
        return x;
    }
}

    class Sub extends Super {
    // callable
    public Sub(int x) {
        super(x+19);    // call
    }

    // callable
    public int getX() {
        return x-19;
    }
}

class Client {
    // callable
    public static void main(String[] args) {
        Super s = new Sub(42);  // call
        s.getX();               // call
    }
}
```

Class `Call` provides two call graph navigation predicates:

- `getCallee` returns the `Callable` that this call (statically) resolves to; note that for a call to an instance (that is, non-static) method, the actual method invoked at runtime may be some other method that overrides this method.
- `getCaller` returns the `Callable` of which this call is syntactically part.

For instance, in our example `getCallee` of the second call in `Client.main` would return `Super.getX`. At runtime, though, this call would actually invoke `Sub.getX`.

Class `Callable` defines a large number of member predicates; for our purposes, the two most important ones are:

- `calls(Callable target)` succeeds if this callable contains a call whose callee is `target`.

- `polyCalls(Callable target)` succeeds if this callable may call `target` at runtime; this is the case if it contains a call whose callee is either `target` or a method that `target` overrides.

In our example, `Client.main` calls the constructor `Sub(int)` and the method `Super.getX`; additionally, it `polyCalls` method `Sub.getX`.

6.6.2 Example: Finding unused methods

We can use the `Callable` class to write a query that finds methods that are not called by any other method:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee))
select callee
```

See this in the query console on LGTM.com. This simple query typically returns a large number of results.

> Note
>
> We have to use `polyCalls` instead of `calls` here: we want to be reasonably sure that `callee` is not called, either directly or via overriding.

Running this query on a typical Java project results in lots of hits in the Java standard library. This makes sense, since no single client program uses every method of the standard library. More generally, we may want to exclude methods and constructors from compiled libraries. We can use the predicate `fromSource` to check whether a compilation unit is a source file, and refine our query:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource()
select callee, "Not called."
```

See this in the query console on LGTM.com. This change reduces the number of results returned for most projects.

We might also notice several unused methods with the somewhat strange name `<clinit>`: these are class initializers; while they are not explicitly called anywhere in the code, they are called implicitly whenever the surrounding class is loaded. Hence it makes sense to exclude them from our query. While we are at it, we can also exclude finalizers, which are similarly invoked implicitly:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize")
select callee, "Not called."
```

See this in the query console on LGTM.com. This also reduces the number of results returned by most projects.

We may also want to exclude public methods from our query, since they may be external API entry points:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize") and
    not callee.isPublic()
select callee, "Not called."
```

See this in the query console on LGTM.com. This should have a more noticeable effect on the number of results returned.

A further special case is non-public default constructors: in the singleton pattern, for example, a class is provided with private empty default constructor to prevent it from being instantiated. Since the very purpose of such constructors is their not being called, they should not be flagged up:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize") and
    not callee.isPublic() and
    not callee.(Constructor).getNumberOfParameters() = 0
select callee, "Not called."
```

See this in the query console on LGTM.com. This change has a large effect on the results for some projects but little effect on the results for others. Use of this pattern varies widely between different projects.

Finally, on many Java projects there are methods that are invoked indirectly by reflection. So, while there are no calls invoking these methods, they are, in fact, used. It is in general very hard to identify such methods. A very common special case, however, is JUnit test methods, which are reflectively invoked by a test runner. The CodeQL library for Java has support for recognizing test classes of JUnit and other testing frameworks, which we can employ to filter out methods defined in such classes:

```
import java

from Callable callee
where not exists(Callable caller | caller.polyCalls(callee)) and
    callee.getCompilationUnit().fromSource() and
    not callee.hasName("<clinit>") and not callee.hasName("finalize") and
    not callee.isPublic() and
    not callee.(Constructor).getNumberOfParameters() = 0 and
    not callee.getDeclaringType() instanceof TestClass
select callee, "Not called."
```

See this in the query console on LGTM.com. This should give a further reduction in the number of results returned.

6.6.3 Further reading

- CodeQL queries for Java

- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.7 Annotations in Java

CodeQL databases of Java projects contain information about all annotations attached to program elements.

6.7.1 About working with annotations

Annotations are represented by these CodeQL classes:

- The class `Annotatable` represents all entities that may have an annotation attached to them (that is, packages, reference types, fields, methods, and local variables).
- The class `AnnotationType` represents a Java annotation type, such as `java.lang.Override`; annotation types are interfaces.
- The class `AnnotationElement` represents an annotation element, that is, a member of an annotation type.
- The class `Annotation` represents an annotation such as `@Override`; annotation values can be accessed through member predicate `getValue`.

For example, the Java standard library defines an annotation `SuppressWarnings` that instructs the compiler not to emit certain kinds of warnings:

```
package java.lang;

public @interface SuppressWarnings {
    String[] value;
}
```

`SuppressWarnings` is represented as an `AnnotationType`, with `value` as its only `AnnotationElement`.

A typical usage of `SuppressWarnings` would be this annotation for preventing a warning about using raw types:

```
class A {
    @SuppressWarnings("rawtypes")
    public A(java.util.List rawlist) {
    }
}
```

The expression `@SuppressWarnings("rawtypes")` is represented as an `Annotation`. The string literal `"rawtypes"` is used to initialize the annotation element `value`, and its value can be extracted from the annotation by means of the `getValue` predicate.

We could then write this query to find all `@SuppressWarnings` annotations attached to constructors, and return both the annotation itself and the value of its `value` element:

```
import java

from Constructor c, Annotation ann, AnnotationType anntp
where ann = c.getAnAnnotation() and
    anntp = ann.getType() and
    anntp.hasQualifiedName("java.lang", "SuppressWarnings")
select ann, ann.getValue("value")
```

See the full query in the query console on LGTM.com. Several of the LGTM.com demo projects use the `@SuppressWarnings` annotation. Looking at the `values` of the annotation element returned by the query, we can see that the *apache/activemq* project uses the `"rawtypes"` value described above.

As another example, this query finds all annotation types that only have a single annotation element, which has name `value`:

```
import java

from AnnotationType anntp
where forex(AnnotationElement elt |
    elt = anntp.getAnAnnotationElement() |
    elt.getName() = "value"
)
select anntp
```

See the full query in the query console on LGTM.com.

6.7.2 Example: Finding missing `@Override` annotations

In newer versions of Java, its recommended (though not required) that you annotate methods that override another method with an `@Override` annotation. These annotations, which are checked by the compiler, serve as documentation, and also help you avoid accidental overloading where overriding was intended.

For example, consider this example program:

```
class Super {
    public void m() {}
}

class Sub1 extends Super {
    @Override public void m() {}
}

class Sub2 extends Super {
    public void m() {}
}
```

Here, both `Sub1.m` and `Sub2.m` override `Super.m`, but only `Sub1.m` is annotated with `@Override`.

Well now develop a query for finding methods like `Sub2.m` that should be annotated with `@Override`, but are not.

As a first step, lets write a query that finds all `@Override` annotations. Annotations are expressions, so their type can be accessed using `getType`. Annotation types, on the other hand, are interfaces, so their qualified name can

be queried using `hasQualifiedName`. Therefore we can implement the query like this:

```
import java

from Annotation ann
where ann.getType().hasQualifiedName("java.lang", "Override")
select ann
```

As always, it is a good idea to try this query on a CodeQL database for a Java project to make sure it actually produces some results. On the earlier example, it should find the annotation on `Sub1.m`. Next, we encapsulate the concept of an `@Override` annotation as a CodeQL class:

```
class OverrideAnnotation extends Annotation {
    OverrideAnnotation() {
        this.getType().hasQualifiedName("java.lang", "Override")
    }
}
```

This makes it very easy to write our query for finding methods that override another method, but dont have an `@Override` annotation: we use predicate `overrides` to find out whether one method overrides another, and predicate `getAnAnnotation` (available on any `Annotatable`) to retrieve some annotation.

```
import java

from Method overriding, Method overridden
where overriding.overrides(overridden) and
    not overriding.getAnAnnotation() instanceof OverrideAnnotation
select overriding, "Method overrides another method, but does not have an @Override annotation."
```

See this in the query console on LGTM.com. In practice, this query may yield many results from compiled library code, which arent very interesting. Its therefore a good idea to add another conjunct `overriding.fromSource()` to restrict the result to only report methods for which source code is available.

6.7.3 Example: Finding calls to deprecated methods

As another example, we can write a query that finds calls to methods marked with a `@Deprecated` annotation.

For example, consider this example program:

```
class A {
    @Deprecated void m() {}

    @Deprecated void n() {
        m();
    }

    void r() {
        m();
    }
}
```

Here, both `A.m` and `A.n` are marked as deprecated. Methods `n` and `r` both call `m`, but note that `n` itself is deprecated,

so we probably should not warn about this call.

As in the previous example, well start by defining a class for representing `@Deprecated` annotations:

```
class DeprecatedAnnotation extends Annotation {
    DeprecatedAnnotation() {
        this.getType().hasQualifiedName("java.lang", "Deprecated")
    }
}
```

Now we can define a class for representing deprecated methods:

```
class DeprecatedMethod extends Method {
    DeprecatedMethod() {
        this.getAnAnnotation() instanceof DeprecatedAnnotation
    }
}
```

Finally, we use these classes to find calls to deprecated methods, excluding calls that themselves appear in deprecated methods:

```
import java

from Call call
where call.getCallee() instanceof DeprecatedMethod
    and not call.getCaller() instanceof DeprecatedMethod
select call, "This call invokes a deprecated method."
```

In our example, this query flags the call to `A.m` in `A.r`, but not the one in `A.n`.

For more information about the class `Call`, see *Navigating the call graph*.

Improvements

The Java standard library provides another annotation type `java.lang.SupressWarnings` that can be used to suppress certain categories of warnings. In particular, it can be used to turn off warnings about calls to deprecated methods. Therefore, it makes sense to improve our query to ignore calls to deprecated methods from inside methods that are marked with `@SuppressWarnings("deprecated")`.

For instance, consider this slightly updated example:

```
class A {
    @Deprecated void m() {}

    @Deprecated void n() {
        m();
    }

    @SuppressWarnings("deprecated")
    void r() {
        m();
    }
}
```

Here, the programmer has explicitly suppressed warnings about deprecated calls in `A.r`, so our query should not flag the call to `A.m` any more.

To do so, we first introduce a class for representing all `@SuppressWarnings` annotations where the string `deprecated` occurs among the list of warnings to suppress:

```
class SuppressDeprecationWarningAnnotation extends Annotation {
    SuppressDeprecationWarningAnnotation() {
        this.getType().hasQualifiedName("java.lang", "SuppressWarnings") and
        this.getAValue().(Literal).getLiteral().regexpMatch(".*deprecation.*")
    }
}
```

Here, we use `getAValue()` to retrieve any annotation value: in fact, annotation type `SuppressWarnings` only has a single annotation element, so every `@SuppressWarnings` annotation only has a single annotation value. Then, we ensure that it is a literal, obtain its string value using `getLiteral`, and check whether it contains the string `deprecation` using a regular expression match.

For real-world use, this check would have to be generalized a bit: for example, the OpenJDK Java compiler allows `@SuppressWarnings("all")` annotations to suppress all warnings. We may also want to make sure that `deprecation` is matched as an entire word, and not as part of another word, by changing the regular expression to `".*\\bdeprecation\\b.*"`.

Now we can extend our query to filter out calls in methods carrying a `SuppressDeprecationWarningAnnotation`:

```
import java

// Insert the class definitions from above

from Call call
where call.getCallee() instanceof DeprecatedMethod
    and not call.getCaller() instanceof DeprecatedMethod
    and not call.getCaller().getAnAnnotation() instanceof SuppressDeprecationWarningAnnotation
select call, "This call invokes a deprecated method."
```

See this in the query console on LGTM.com. Its fairly common for projects to contain calls to methods that appear to be deprecated.

6.7.4 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.8 Javadoc

You can use CodeQL to find errors in Javadoc comments in Java code.

6.8.1 About analyzing Javadoc

To access Javadoc associated with a program element, we use member predicate `getDoc` of class `Element`, which returns a `Documentable`. Class `Documentable`, in turn, offers a member predicate `getJavadoc` to retrieve the Javadoc attached to the element in question, if any.

Javadoc comments are represented by class `Javadoc`, which provides a view of the comment as a tree of `JavadocElement` nodes. Each `JavadocElement` is either a `JavadocTag`, representing a tag, or a `JavadocText`, representing a piece of free-form text.

The most important member predicates of class `Javadoc` are:

- `getAChild` - retrieves a top-level `JavadocElement` node in the tree representation.
- `getVersion` - returns the value of the `@version` tag, if any.
- `getAuthor` - returns the value of the `@author` tag, if any.

For example, the following query finds all classes that have both an `@author` tag and a `@version` tag, and returns this information:

```
import java

from Class c, Javadoc jdoc, string author, string version
where jdoc = c.getDoc().getJavadoc() and
    author = jdoc.getAuthor() and
    version = jdoc.getVersion()
select c, author, version
```

`JavadocElement` defines member predicates `getAChild` and `getParent` to navigate up and down the tree of elements. It also provides a predicate `getTagName` to return the tags name, and a predicate `getText` to access the text associated with the tag.

We could rewrite the above query to use this API instead of `getAuthor` and `getVersion`:

```
import java

from Class c, Javadoc jdoc, JavadocTag authorTag, JavadocTag versionTag
where jdoc = c.getDoc().getJavadoc() and
    authorTag.getTagName() = "@author" and authorTag.getParent() = jdoc and
    versionTag.getTagName() = "@version" and versionTag.getParent() = jdoc
select c, authorTag.getText(), versionTag.getText()
```

The `JavadocTag` has several subclasses representing specific kinds of Javadoc tags:

- `ParamTag` represents `@param` tags; member predicate `getParamName` returns the name of the parameter being documented.
- `ThrowsTag` represents `@throws` tags; member predicate `getExceptionName` returns the name of the exception being documented.
- `AuthorTag` represents `@author` tags; member predicate `getAuthorName` returns the name of the author.

6.8.2 Example: Finding spurious @param tags

As an example of using the CodeQL Javadoc API, lets write a query that finds `@param` tags that refer to a non-existent parameter.

For example, consider this program:

```
class A {
    /**
    * @param lst a list of strings
    */
    public String get(List<String> list) {
        return list.get(0);
    }
}
```

Here, the `@param` tag on `A.get` misspells the name of parameter `list` as `lst`. Our query should be able to find such cases.

To begin with, we write a query that finds all callables (that is, methods or constructors) and their `@param` tags:

```
import java

from Callable c, ParamTag pt
where c.getDoc().getJavadoc() = pt.getParent()
select c, pt
```

Its now easy to add another conjunct to the `where` clause, restricting the query to `@param` tags that refer to a non-existent parameter: we simply need to require that no parameter of `c` has the name `pt.getParamName()`.

```
import java

from Callable c, ParamTag pt
where c.getDoc().getJavadoc() = pt.getParent() and
    not c.getAParameter().hasName(pt.getParamName())
select pt, "Spurious @param tag."
```

6.8.3 Example: Finding spurious @throws tags

A related, but somewhat more involved, problem is finding `@throws` tags that refer to an exception that the method in question cannot actually throw.

For example, consider this Java program:

```
import java.io.IOException;

class A {
    /**
    * @throws IOException thrown if some IO operation fails
    * @throws RuntimeException thrown if something else goes wrong
    */
    public void foo() {
        // ...
    }
}
```

Notice that the Javadoc comment of `A.foo` documents two thrown exceptions: `IOException` and `RuntimeException`. The former is clearly spurious: `A.foo` doesnt have a `throws IOException` clause, and

therefore cant throw this kind of exception. On the other hand, `RuntimeException` is an unchecked exception, so it can be thrown even if there is no explicit `throws` clause listing it. So our query should flag the `@throws` tag for `IOException`, but not the one for `RuntimeException`.

Remember that the CodeQL library represents `@throws` tags using class `ThrowsTag`. This class doesnt provide a member predicate for determining the exception type that is being documented, so we first need to implement our own version. A simple version might look like this:

```
RefType getDocumentedException(ThrowsTag tt) {
    result.hasName(tt.getExceptionName())
}
```

Similarly, `Callable` doesnt come with a member predicate for querying all exceptions that the method or constructor may possibly throw. We can, however, implement this ourselves by using `getAnException` to find all `throws` clauses of the callable, and then use `getType` to resolve the corresponding exception types:

```
predicate mayThrow(Callable c, RefType exn) {
    exn.getASupertype*() = c.getAnException().getType()
}
```

Note the use of `getASupertype*` to find both exceptions declared in a `throws` clause and their subtypes. For instance, if a method has a `throws IOException` clause, it may throw `MalformedURLException`, which is a subtype of `IOException`.

Now we can write a query for finding all callables `c` and `@throws` tags `tt` such that:

- `tt` belongs to a Javadoc comment attached to `c`.
- `c` cant throw the exception documented by `tt`.

```
import java

// Insert the definitions from above

from Callable c, ThrowsTag tt, RefType exn
where c.getDoc().getJavadoc() = tt.getParent+() and
    exn = getDocumentedException(tt) and
    not mayThrow(c, exn)
select tt, "Spurious @throws tag."
```

See this in the query console on LGTM.com. This finds several results in the LGTM.com demo projects.

Improvements

Currently, there are two problems with this query:

1. `getDocumentedException` is too liberal: it will return *any* reference type with the right name, even if its in a different package and not actually visible in the current compilation unit.
2. `mayThrow` is too restrictive: it doesnt account for unchecked exceptions, which do not need to be declared.

To see why the former is a problem, consider this program:

```
class IOException extends Exception {}

class B {
    /** @throws IOException an IO exception */
    void bar() throws IOException {}
}
```

This program defines its own class `IOException`, which is unrelated to the class `java.io.IOException` in the standard library: they are in different packages. Our `getDocumentedException` predicate doesnt check packages, however, so it will consider the `@throws` clause to refer to both `IOException` classes, and thus flag the `@param` tag as spurious, since `B.bar` cant actually throw `java.io.IOException`.

As an example of the second problem, method `A.foo` from our previous example was annotated with a `@throws RuntimeException` tag. Our current version of `mayThrow`, however, would think that `A.foo` cant throw a `RuntimeException`, and thus flag the tag as spurious.

We can make `mayThrow` less restrictive by introducing a new class to represent unchecked exceptions, which are just the subtypes of `java.lang.RuntimeException` and `java.lang.Error`:

```
class UncheckedException extends RefType {
    UncheckedException() {
        this.getASupertype*().hasQualifiedName("java.lang", "RuntimeException") or
        this.getASupertype*().hasQualifiedName("java.lang", "Error")
    }
}
```

Now we incorporate this new class into our `mayThrow` predicate:

```
predicate mayThrow(Callable c, RefType exn) {
    exn instanceof UncheckedException or
    exn.getASupertype*() = c.getAnException().getType()
}
```

Fixing `getDocumentedException` is more complicated, but we can easily cover three common cases:

1. The `@throws` tag specifies the fully qualified name of the exception.
2. The `@throws` tag refers to a type in the same package.
3. The `@throws` tag refers to a type that is imported by the current compilation unit.

The first case can be covered by changing `getDocumentedException` to use the qualified name of the `@throws` tag. To handle the second and the third case, we can introduce a new predicate `visibleIn` that checks whether a reference type is visible in a compilation unit, either by virtue of belonging to the same package or by being explicitly imported. We then rewrite `getDocumentedException` as:

```
predicate visibleIn(CompilationUnit cu, RefType tp) {
    cu.getPackage() = tp.getPackage()
    or
    exists(ImportType it | it.getCompilationUnit() = cu | it.getImportedType() = tp)
}

RefType getDocumentedException(ThrowsTag tt) {
```

(continues on next page)

(continued from previous page)

```
    result.getQualifiedName() = tt.getExceptionName()
    or
    (result.hasName(tt.getExceptionName()) and visibleIn(tt.getFile(), result))
}
```

See this in the query console on LGTM.com. This finds many fewer, more interesting results in the LGTM.com demo projects.

Currently, `visibleIn` only considers single-type imports, but you could extend it with support for other kinds of imports.

6.8.4 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.9 Working with source locations

You can use the location of entities within Java code to look for potential errors. Locations allow you to deduce the presence, or absence, of white space which, in some cases, may indicate a problem.

6.9.1 About source locations

Java offers a rich set of operators with complex precedence rules, which are sometimes confusing to developers. For instance, the class `ByteBufferCache` in the OpenJDK Java compiler (which is a member class of `com.sun.tools.javac.util.BaseFileManager`) contains this code for allocating a buffer:

```
ByteBuffer.allocate(capacity + capacity>>1)
```

Presumably, the author meant to allocate a buffer that is 1.5 times the size indicated by the variable `capacity`. In fact, however, operator `+` binds tighter than operator `>>`, so the expression `capacity + capacity>>1` is parsed as `(capacity + capacity)>>1`, which equals `capacity` (unless there is an arithmetic overflow).

Note that the source layout gives a fairly clear indication of the intended meaning: there is more white space around `+` than around `>>`, suggesting that the latter is meant to bind more tightly.

Were going to develop a query that finds this kind of suspicious nesting, where the operator of the inner expression has more white space around it than the operator of the outer expression. This pattern may not necessarily indicate a bug, but at the very least it makes the code hard to read and prone to misinterpretation.

White space is not directly represented in the CodeQL database, but we can deduce its presence from the location information associated with program elements and AST nodes. So, before we write our query, we need an understanding of source location management in the standard library for Java.

6.9.2 Location API

For every entity that has a representation in Java source code (including, in particular, program elements and AST nodes), the standard CodeQL library provides these predicates for accessing source location information:

- `getLocation` returns a `Location` object describing the start and end position of the entity.
- `getFile` returns a `File` object representing the file containing the entity.
- `getTotalNumberOfLines` returns the number of lines the source code of the entity spans.
- `getNumberOfCommentLines` returns the number of comment lines.
- `getNumberOfLinesOfCode` returns the number of non-comment lines.

For example, lets assume this Java class is defined in the compilation unit `SayHello.java`:

```
package pkg;

class SayHello {
    public static void main(String[] args) {
        System.out.println(
            // Display personalized message
            "Hello, " + args[0];
        );
    }
}
```

Invoking `getFile` on the expression statement in the body of `main` returns a `File` object representing the file `SayHello.java`. The statement spans four lines in total (`getTotalNumberOfLines`), of which one is a comment line (`getNumberOfCommentLines`), while three lines contain code (`getNumberOfLinesOfCode`).

Class `Location` defines member predicates `getStartLine`, `getEndLine`, `getStartColumn` and `getEndColumn` to retrieve the line and column number an entity starts and ends at, respectively. Both lines and columns are counted starting from 1 (not 0), and the end position is inclusive, that is, it is the position of the last character belonging to the source code of the entity.

In our example, the expression statement starts at line 5, column 3 (the first two characters on the line are tabs, which each count as one character), and it ends at line 8, column 4.

Class `File` defines these member predicates:

- `getAbsolutePath` returns the fully qualified name of the file.
- `getRelativePath` returns the path of the file relative to the base directory of the source code.
- `getExtension` returns the extension of the file.
- `getStem` returns the base name of the file, without its extension.

In our example, assume file `A.java` is located in directory `/home/testuser/code/pkg`, where `/home/testuser/code` is the base directory of the program being analyzed. Then, a `File` object for `A.java` returns:

- `getAbsolutePath` is `/home/testuser/code/pkg/A.java`.
- `getRelativePath` is `pkg/A.java`.
- `getExtension` is `java`.
- `getStem` is `A`.

6.9.3 Determining white space around an operator

Lets start by considering how to write a predicate that computes the total amount of white space surrounding the operator of a given binary expression. If `rcol` is the start column of the expressions right operand and `lcol` is the end column of its left operand, then `rcol - (lcol+1)` gives us the total number of characters in between the two operands (note that we have to use `lcol+1` instead of `lcol` because end positions are inclusive).

This number includes the length of the operator itself, which we need to subtract out. For this, we can use predicate `getOp`, which returns the operator string, surrounded by one white space on either side. Overall, the expression for computing the amount of white space around the operator of a binary expression `expr` is:

```
rcol - (lcol+1) - (expr.getOp().length()-2)
```

Clearly, however, this only works if the entire expression is on a single line, which we can check using predicate `getTotalNumberOfLines` introduced above. We are now in a position to define our predicate for computing white space around operators:

```
int operatorWS(BinaryExpr expr) {
    exists(int lcol, int rcol |
        expr.getNumberOfLinesOfCode() = 1 and
        lcol = expr.getLeftOperand().getLocation().getEndColumn() and
        rcol = expr.getRightOperand().getLocation().getStartColumn() and
        result = rcol - (lcol+1) - (expr.getOp().length()-2)
    )
}
```

Notice that we use an `exists` to introduce our temporary variables `lcol` and `rcol`. You could write the predicate without them by just inlining `lcol` and `rcol` into their use, at some cost in readability.

6.9.4 Find suspicious nesting

Heres a first version of our query:

```
import java

// Insert predicate defined above

from BinaryExpr outer, BinaryExpr inner,
    int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

See this in the query console on LGTM.com. This query is likely to find results on most projects.

The first conjunct of the `where` clause restricts `inner` to be an operand of `outer`, the second conjunct binds `wsinner` and `wsouter`, while the last conjunct selects the suspicious cases.

At first, we might be tempted to write `inner = outer.getAnOperand()` in the first conjunct. This, however, wouldnt be quite correct: `getAnOperand` strips off any surrounding parentheses from its result, which is often useful, but not what we want here: if there are parentheses around the inner expression, then the programmer probably knew what they were doing, and the query should not flag this expression.

Improving the query

If we run this initial query, we might notice some false positives arising from asymmetric white space. For instance, the following expression is flagged as suspicious, although it is unlikely to cause confusion in practice:

```
i< start + 100
```

Note that our predicate `operatorWS` computes the **total** amount of white space around the operator, which, in this case, is one for the `<` and two for the `+`. Ideally, we would like to exclude cases where the amount of white space before and after the operator are not the same. Currently, CodeQL databases dont record enough information to figure this out, but as an approximation we could require that the total number of white space characters is even:

```
import java

// Insert predicate definition from above

from BinaryExpr outer, BinaryExpr inner,
    int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

See this in the query console on LGTM.com. Any results will be refined by our changes to the query.

Another source of false positives are associative operators: in an expression of the form `x + y+z`, the first plus is syntactically nested inside the second, since `+` in Java associates to the left; hence the expression is flagged as suspicious. But since `+` is associative to begin with, it does not matter which way around the operators are nested, so this is a false positive. To exclude these cases, let us define a new class identifying binary expressions with an associative operator:

```
class AssociativeOperator extends BinaryExpr {
    AssociativeOperator() {
        this instanceof AddExpr or
        this instanceof MulExpr or
        this instanceof BitwiseExpr or
        this instanceof AndLogicalExpr or
        this instanceof OrLogicalExpr
    }
}
```

Now we can extend our query to discard results where the outer and the inner expression both have the same, associative operator:

```
import java

// Insert predicate and class definitions from above

from BinaryExpr inner, BinaryExpr outer, int wsouter, int wsinner
where inner = outer.getAChildExpr() and
    not (inner.getOp() = outer.getOp() and outer instanceof AssociativeOperator) and
```

(continues on next page)

(continued from previous page)

```
    wsinner = operatorWS(inner) and wsouter = operatorWS(outer) and
    wsinner % 2 = 0 and wsouter % 2 = 0 and
    wsinner > wsouter
select outer, "Whitespace around nested operators contradicts precedence."
```

See this in the query console on LGTM.com.

Notice that we again use `getOp`, this time to determine whether two binary expressions have the same operator. Running our improved query now finds the Java standard library bug described in the Overview. It also flags up the following suspicious code in Hadoop HBase:

```
KEY_SLAVE = tmp[ i+1 % 2 ];
```

Whitespace suggests that the programmer meant to toggle `i` between zero and one, but in fact the expression is parsed as `i + (1%2)`, which is the same as `i + 1`, so `i` is simply incremented.

6.9.5 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

6.10 Abstract syntax tree classes for working with Java programs

CodeQL has a large selection of classes for representing the abstract syntax tree of Java programs.

The abstract syntax tree (AST) represents the syntactic structure of a program. Nodes on the AST represent elements such as statements and expressions.

6.10.1 Statement classes

This table lists all subclasses of Stmt.

Statement syntax	CodeQL class	Superclasses	Remarks
`;`	EmptyStmt		
Expr `;`	ExprStmt		
`{` Stmt `...` `}`	Block		
`if (` Expr `)` Stmt `else` Stmt	IfStmt	ConditionalStmt	
`if (` Expr `)` Stmt			
`while (` Expr `)` Stmt	WhileStmt	ConditionalStmt, LoopStmt	
`do` Stmt `while (` Expr `)`	DoStmt	ConditionalStmt, LoopStmt	
`for (` Expr `;` Expr `;` Expr `)` Stmt	ForStmt	ConditionalStmt, LoopStmt	
`for (` VarAccess `:` Expr `)` Stmt	EnhancedForStmt	LoopStmt	
`switch (` Expr `) {` SwitchCase `...` `}`	SwitchStmt		
`try {` Stmt `...` `} finally {` Stmt `...` `}`	TryStmt		
`return` Expr `;`	ReturnStmt		
`return ;`			
`throw` Expr `;`	ThrowStmt		
`break ;`	BreakStmt	JumpStmt	
`break label ;`			
`continue ;`	ContinueStmt	JumpStmt	
`continue label ;`			
`label :` Stmt	LabeledStmt		
`synchronized (` Expr `)` Stmt	SynchronizedStmt		
`assert` Expr `:` Expr `;`	AssertStmt		
`assert` Expr `;`			
TypeAccess `name ;`	LocalVariableDeclStmt		
`class name {` Member `...` `} ;`	LocalClassDeclStmt		
`this (` Expr `, ...` `) ;`	ThisConstructorInvocationStmt		
`super (` Expr `, ...` `) ;`	SuperConstructorInvocationStmt		
`catch (` TypeAccess `name )` `{` Stmt `...` `}`	CatchClause		can only occur as child of a TryStmt
`case` Literal `:` Stmt `...`	ConstCase		can only occur as child of a SwitchStmt
`default :` Stmt `...`	DefaultCase		can only occur as child of a SwitchStmt

6.10.2 Expression classes

There are many expression classes, so we present them by category. All classes in this section are subclasses of Expr.

Literals

All classes in this subsection are subclasses of Literal.

Expression syntax example	CodeQL class
`true`	BooleanLiteral
`23`	IntegerLiteral
`23l`	LongLiteral
`4.2f`	FloatingPointLiteral
`4.2`	DoubleLiteral
`'a'`	CharacterLiteral
`"Hello"`	StringLiteral
`null`	NullLiteral

Unary expressions

All classes in this subsection are subclasses of UnaryExpr.

Expression syntax	CodeQL class	Superclasses	Remarks
Expr`++`	PostIncExpr	UnaryAssignExpr	
Expr`--`	PostDecExpr	UnaryAssignExpr	
`++`Expr	PreIncExpr	UnaryAssignExpr	
`--`Expr	PreDecExpr	UnaryAssignExpr	
`~`Expr	BitNotExpr	BitwiseExpr	see below for other subclasses of BitwiseExpr
`-`Expr	MinusExpr		
`+`Expr	PlusExpr		
`!`Expr	LogNotExpr	LogicExpr	see below for other subclasses of LogicExpr

Binary expressions

All classes in this subsection are subclasses of BinaryExpr.

Expression syntax	CodeQL class	Superclasses
Expr * Expr	MulExpr	
Expr / Expr	DivExpr	
Expr % Expr	RemExpr	
Expr + Expr	AddExpr	
Expr - Expr	SubExpr	
Expr << Expr	LShiftExpr	
Expr >> Expr	RShiftExpr	
Expr >>> Expr	URShiftExpr	
Expr && Expr	AndLogicalExpr	LogicExpr
Expr || Expr	OrLogicalExpr	LogicExpr
Expr < Expr	LTExpr	ComparisonExpr
Expr > Expr	GTExpr	ComparisonExpr
Expr <= Expr	LEExpr	ComparisonExpr
Expr >= Expr	GEExpr	ComparisonExpr
Expr == Expr	EQExpr	EqualityTest
Expr != Expr	NEExpr	EqualityTest
Expr & Expr	AndBitwiseExpr	BitwiseExpr
Expr | Expr	OrBitwiseExpr	BitwiseExpr
Expr ^ Expr	XorBitwiseExpr	BitwiseExpr

Assignment expressions

All classes in this table are subclasses of Assignment.

Expression syntax	CodeQL class	Superclasses
Expr = Expr	AssignExpr	
Expr += Expr	AssignAddExpr	AssignOp
Expr -= Expr	AssignSubExpr	AssignOp
Expr *= Expr	AssignMulExpr	AssignOp
Expr /= Expr	AssignDivExpr	AssignOp
Expr %= Expr	AssignRemExpr	AssignOp
Expr &= Expr	AssignAndExpr	AssignOp
Expr |= Expr	AssignOrExpr	AssignOp
Expr ^= Expr	AssignXorExpr	AssignOp
Expr <<= Expr	AssignLShiftExpr	AssignOp
Expr >>= Expr	AssignRShiftExpr	AssignOp
Expr >>>= Expr	AssignURShiftExpr	AssignOp

Accesses

Expression syntax examples	CodeQL class
`this`	ThisAccess
`Outer.this`	
`super`	SuperAccess
`Outer.super`	
`x`	VarAccess
`e.f`	
`a[i]`	ArrayAccess
`f(...)`	MethodAccess
`e.m(...)`	
`String`	TypeAccess
`java.lang.String`	
`? extends Number`	WildcardTypeAccess
`? super Double`	

A VarAccess that refers to a field is a FieldAccess.

Miscellaneous

Expression syntax examples	CodeQL class	Remarks
`(int) f`	CastExpr	
`(23 + 42)`	ParExpr	
`o instanceof String`	InstanceOfExpr	
Expr `?` Expr `:` Expr	ConditionalExpr	
`String. class`	TypeLiteral	
`new A()`	ClassInstanceExpr	
`new String[3][2]`	ArrayCreationExpr	
`new int[] { 23, 42 }`		
`{ 23, 42 }`	ArrayInit	can only appear as an initializer or as a child of an ArrayCreationExpr
`@Annot(key=val)`	Annotation	

6.10.3 Further reading

- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools
- *Basic query for Java code*: Learn to write and run a simple CodeQL query using LGTM.

- *CodeQL library for Java*: When analyzing Java code, you can use the large collection of classes in the CodeQL library for Java.
- *Analyzing data flow in Java*: You can use CodeQL to track the flow of data through a Java program to its use.
- *Java types*: You can use CodeQL to find out information about data types used in Java code. This allows you to write queries to identify specific type-related issues.
- *Overflow-prone comparisons in Java*: You can use CodeQL to check for comparisons in Java code where one side of the comparison is prone to overflow.
- *Navigating the call graph*: CodeQL has classes for identifying code that calls other code, and code that can be called from elsewhere. This allows you to find, for example, methods that are never used.
- *Annotations in Java*: CodeQL databases of Java projects contain information about all annotations attached to program elements.
- *Javadoc*: You can use CodeQL to find errors in Javadoc comments in Java code.
- *Working with source locations*: You can use the location of entities within Java code to look for potential errors. Locations allow you to deduce the presence, or absence, of white space which, in some cases, may indicate a problem.
- *Abstract syntax tree classes for working with Java programs*: CodeQL has a large selection of classes for representing the abstract syntax tree of Java programs.

CHAPTER

SEVEN

CODEQL FOR JAVASCRIPT

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from JavaScript codebases.

7.1 Basic query for JavaScript code

Learn to write and run a simple CodeQL query using LGTM.

7.1.1 About the query

In JavaScript, any expression can be turned into an expression statement. While this is sometimes convenient, it can be dangerous. For example, imagine a programmer wants to assign a new value to a variable `x` by means of an assignment `x = 42`. However, they accidentally type two equals signs, producing the comparison statement `x == 42`. This is valid JavaScript, so no error is generated. The statement simply compares `x` to `42`, and then discards the result of the comparison.

The query you will run finds instances of this problem. The query searches for expressions `e` that are purethat is, their evaluation does not lead to any side effectsbut appear as an expression statement.

7.1.2 Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **JavaScript** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import javascript

from Expr e
where e.isPure() and
```

(continues on next page)

(continued from previous page)

```
  e.getParent() instanceof ExprStmt
select e, "This expression has no effect."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `e` and is linked to the location in the source code of the project where `e` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis () at the bottom of the table indicates that the entire list is not displayedclick it to show more results.

6. If any matching code is found, click one of the links in the `e` column to view the expression in the code viewer.

   The matching statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import javascript`	Imports the standard CodeQL libraries for JavaScript.	Every query begins with one or more `import` statements.
`from Expr e`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	`e` is declared as a variable that ranges over expressions.
`where e.isPure() and e.getParent() instanceof ExprStmt`	Defines a condition on the variables.	`e.isPure()`: The expression is side-effect-free. `e.getParent() instanceof ExprStmt`: The parent of the expression is an expression statement.
`select e, "This expression has no effect."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Report the expression with a string that explains the problem.

7.1.3 Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find `use strict` directives. These are interpreted specially by modern browsers with strict mode support and so these expressions *do* have an effect.

To remove directives from the results:

1. Extend the `where` clause to include the following extra condition:

   ```
   and not e.getParent() instanceof Directive
   ```

   The `where` clause is now:

   ```
   where e.isPure() and
     e.getParent() instanceof ExprStmt and
     not e.getParent() instanceof Directive
   ```

2. Click **Run**.

   There are now fewer results as `use strict` directives are no longer reported.

The improved query finds several results on the example project including this result:

```
point.bias == -1;
```

As written, this statement compares `point.bias` against `-1` and then discards the result. Most likely, it was instead meant to be an assignment `point.bias = -1`.

7.1.4 Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- QL language reference
- CodeQL tools

7.2 CodeQL library for JavaScript

When youre analyzing a JavaScript program, you can make use of the large collection of classes in the CodeQL library for JavaScript.

7.2.1 Overview

There is an extensive CodeQL library for analyzing JavaScript code. The classes in this library present the data from a CodeQL database in an object-oriented form and provide abstractions and predicates to help you with common analysis tasks.

The library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `javascript.qll` imports most other standard library modules, so you can include the complete library by beginning your query with:

```
import javascript
```

The rest of this tutorial briefly summarizes the most important classes and predicates provided by this library, including references to the detailed API documentation where applicable.

7.2.2 Introducing the library

The CodeQL library for JavaScript presents information about JavaScript source code at different levels:

- **Textual** classes that represent source code as unstructured text files
- **Lexical** classes that represent source code as a series of tokens and comments
- **Syntactic** classes that represent source code as an abstract syntax tree
- **Name binding** classes that represent scopes and variables
- **Control flow** classes that represent the flow of control during execution
- **Data flow** classes that you can use to reason about data flow in JavaScript source code
- **Type inference** classes that you can use to approximate types for JavaScript expressions and variables
- **Call graph** classes that represent the caller-callee relationship between functions
- **Inter-procedural data flow** classes that you can use to define inter-procedural data flow and taint tracking analyses
- **Frameworks** classes that represent source code entities that have a special meaning to JavaScript tools and frameworks

Note that representations above the textual level (for example the lexical representation or the flow graphs) are only available for JavaScript code that does not contain fatal syntax errors. For code with such errors, the only information available is at the textual level, as well as information about the errors themselves.

Additionally, there is library support for working with HTML documents, JSON, and YAML data, JSDoc comments, and regular expressions.

Textual level

At its most basic level, a JavaScript code base can simply be viewed as a collection of files organized into folders, where each file is composed of zero or more lines of text.

Note that the textual content of a program is not included in the CodeQL database unless you specifically request it during extraction. In particular, databases on LGTM (also known as snapshots) do not normally include textual information.

Files and folders

In the CodeQL libraries, files are represented as entities of class File, and folders as entities of class Folder, both of which are subclasses of class Container.

Class Container provides the following member predicates:

- `Container.getParentContainer()` returns the parent folder of the file or folder.
- `Container.getAFile()` returns a file within the folder.
- `Container.getAFolder()` returns a folder nested within the folder.

Note that while `getAFile` and `getAFolder` are declared on class Container, they currently only have results for Folders.

Both files and folders have paths, which can be accessed by the predicate `Container.getAbsolutePath()`. For example, if `f` represents a file with the path `/home/user/project/src/index.js`, then `f.getAbsolutePath()` evaluates to the string `"/home/user/project/src/index.js"`, while `f.getParentContainer().getAbsolutePath()` returns `"/home/user/project/src"`.

These paths are absolute file system paths. If you want to obtain the path of a file relative to the source location in the CodeQL database, use `Container.getRelativePath()` instead. Note, however, that a database may contain files that are not located underneath the source location; for such files, `getRelativePath()` will not return anything.

The following member predicates of class Container provide more information about the name of a file or folder:

- `Container.getBaseName()` returns the base name of a file or folder, not including its parent folder, but including its extension. In the above example, `f.getBaseName()` would return the string `"index.js"`.
- `Container.getStem()` is similar to `Container.getBaseName()`, but it does *not* include the file extension; so `f.getStem()` returns `"index"`.
- `Container.getExtension()` returns the file extension, not including the dot; so `f.getExtension()` returns `"js"`.

For example, the following query computes, for each folder, the number of JavaScript files (that is, files with extension `js`) contained in the folder:

```
import javascript

from Folder d
select d.getRelativePath(), count(File f | f = d.getAFile() and f.getExtension() = "js")
```

See this in the query console on LGTM.com. When you run the query on most projects, the results include folders that contain files with a `js` extension and folders that dont.

Locations

Most entities in a CodeQL database have an associated source location. Locations are identified by four pieces of information: a file, a start line, a start column, an end line, and an end column. Line and column counts are 1-based (so the first character of a file is at line 1, column 1), and the end position is inclusive.

All entities associated with a source location belong to the class Locatable. The location itself is modeled by the class Location and can be accessed through the member predicate `Locatable.getLocation()`. The Location class provides the following member predicates:

- `Location.getFile()`, `Location.getStartLine()`, `Location.getStartColumn()`, `Location.getEndLine()`, `Location.getEndColumn()` return detailed information about the location.
- `Location.getNumLines()` returns the number of (whole or partial) lines covered by the location.
- `Location.startsBefore(Location)` and `Location.endsAfter(Location)` determine whether one location starts before or ends after another location.
- `Location.contains(Location)` indicates whether one location completely contains another location; `l1.contains(l2)` holds if, and only if, `l1.startsBefore(l2)` and `l1.endsAfter(l2)`.

Lines

Lines of text in files are represented by the class Line. This class offers the following member predicates:

- `Line.getText()` returns the text of the line, excluding any terminating newline characters.
- `Line.getTerminator()` returns the terminator character(s) of the line. The last line in a file may not have any terminator characters, in which case this predicate does not return anything; otherwise it returns either the two-character string `"\r\n"` (carriage-return followed by newline), or one of the one-character strings `"\n"` (newline), `"\r"` (carriage-return), `"\u2028"` (Unicode character LINE SEPARATOR), `"\u2029"` (Unicode character PARAGRAPH SEPARATOR).

Note that, as mentioned above, the textual representation of the program is not included in the CodeQL database by default.

Lexical level

A slightly more structured view of a JavaScript program is provided by the classes Token and Comment, which represent tokens and comments, respectively.

Tokens

The most important member predicates of class Token are as follows:

- `Token.getValue()` returns the source text of the token.

- `Token.getIndex()` returns the index of the token within its enclosing script.
- `Token.getNextToken()` and `Token.getPreviousToken()` navigate between tokens.

The Token class has nine subclasses, each representing a particular kind of token:

- EOFToken: a marker token representing the end of a script
- NullLiteralToken, BooleanLiteralToken, NumericLiteralToken, StringLiteralToken and RegularExpressionToken: different kinds of literals
- IdentifierToken and KeywordToken: identifiers and keywords (including reserved words) respectively
- PunctuatorToken: operators and other punctuation symbols

As an example of a query operating entirely on the lexical level, consider the following query, which finds consecutive comma tokens arising from an omitted element in an array expression:

```
import javascript

class CommaToken extends PunctuatorToken {
    CommaToken() {
        getValue() = ","
    }
}

from CommaToken comma
where comma.getNextToken() instanceof CommaToken
select comma, "Omitted array elements are bad style."
```

See this in the query console on LGTM.com. If the query returns no results, this pattern isnt used in the projects that you analyzed.

You can use predicate `Locatable.getFirstToken()` and `Locatable.getLastToken()` to access the first and last token (if any) belonging to an element with a source location.

Comments

The class Comment and its subclasses represent the different kinds of comments that can occur in JavaScript programs:

- Comment: any comment
  - LineComment: a single-line comment terminated by an end-of-line character
    - SlashSlashComment: a plain JavaScript single-line comment starting with `//`
    - HtmlLineComment: a (non-standard) HTML comment
      - HtmlCommentStart: an HTML comment starting with `<!--`
      - HtmlCommentEnd: an HTML comment ending with `-->`
- BlockComment: a block comment potentially spanning multiple lines
  - SlashStarComment: a plain JavaScript block comment surrounded with `/*...*/`
  - DocComment: a documentation block comment surrounded with `/**...*/`

The most important member predicates are as follows:

- `Comment.getText()` returns the source text of the comment, not including delimiters.
- `Comment.getLine(i)` returns the `i`th line of text within the comment (0-based).
- `Comment.getNumLines()` returns the number of lines in the comment.
- `Comment.getNextToken()` returns the token immediately following a comment. Note that such a token always exists: if a comment appears at the end of a file, its following token is an EOFToken.

As an example of a query using only lexical information, consider the following query for finding HTML comments, which are not a standard ECMAScript feature and should be avoided:

```
import javascript

from HtmlLineComment c
select c, "Do not use HTML comments."
```

See this in the query console on LGTM.com. When we ran this query on the *mozilla/pdf.js* project in LGTM.com, we found three HTML comments.

Syntactic level

The majority of classes in the JavaScript library is concerned with representing a JavaScript program as a collection of abstract syntax trees (ASTs).

The class ASTNode contains all entities representing nodes in the abstract syntax trees and defines generic tree traversal predicates:

- `ASTNode.getChild(i)`: returns the `i`th child of this AST node.
- `ASTNode.getAChild()`: returns any child of this AST node.
- `ASTNode.getParent()`: returns the parent node of this AST node, if any.

  Note

  These predicates should only be used to perform generic AST traversal. To access children of specific AST node types, the specialized predicates introduced below should be used instead. In particular, queries should not rely on the numeric indices of child nodes relative to their parent nodes: these are considered an implementation detail that may change between versions of the library.

Top-levels

From a syntactic point of view, each JavaScript program is composed of one or more top-level code blocks (or *top-levels* for short), which are blocks of JavaScript code that do not belong to a larger code block. Top-levels are represented by the class TopLevel and its subclasses:

- TopLevel
  - Script: a stand-alone file or HTML `<script>` element
    - ExternalScript: a stand-alone JavaScript file
    - InlineScript: code embedded inline in an HTML `<script>` tag
  - CodeInAttribute: a code block originating from an HTML attribute value
    - EventHandlerCode: code from an event handler attribute such as `onload`

JavaScriptURL: code from a URL with the `javascript:` scheme

- Externs: a JavaScript file containing externs definitions

Every TopLevel class is contained in a File class, but a single File may contain more than one TopLevel. To go from a `TopLevel tl` to its File, use `tl.getFile()`; conversely, for a `File f`, predicate `f.getATopLevel()` returns a top-level contained in `f`. For every AST node, predicate `ASTNode.getTopLevel()` can be used to find the top-level it belongs to.

The TopLevel class additionally provides the following member predicates:

- `TopLevel.getNumberOfLines()` returns the total number of lines (including code, comments and whitespace) in the top-level.
- `TopLevel.getNumberOfLinesOfCode()` returns the number of lines of code, that is, lines that contain at least one token.
- `TopLevel.getNumberOfLinesOfComments()` returns the number of lines containing or belonging to a comment.
- `TopLevel.isMinified()` determines whether the top-level contains minified code, using a heuristic based on the average number of statements per line.

  Note

  By default, LGTM filters out alerts in minified top-levels, since they are often hard to interpret. When writing your own queries in the LGTM query console, this filtering is *not* done automatically, so you may want to explicitly add a condition of the form `and not e.getTopLevel().isMinified()` or similar to your query to exclude results in minified code.

Statements and expressions

The most important subclasses of ASTNode besides TopLevel are Stmt and Expr, which, together with their subclasses, represent statements and expressions, respectively. This section briefly discusses some of the more important classes and predicates. For a full reference of all the subclasses of Stmt and Expr and their API, see Stmt.qll and Expr.qll.

- Stmt: use `Stmt.getContainer()` to access the innermost function or top-level in which the statement is contained.
  - ControlStmt: a statement that controls the execution of other statements, that is, a conditional, loop, `try` or `with` statement; use `ControlStmt.getAControlledStmt()` to access the statements that it controls.

    IfStmt: an `if` statement; use `IfStmt.getCondition()`, `IfStmt.getThen()` and `IfStmt.getElse()` to access its condition expression, then branch and else branch, respectively.

    LoopStmt: a loop; use `Loop.getBody()` and `Loop.getTest()` to access its body and its test expression, respectively.

    · WhileStmt, DoWhileStmt: a while or do-while loop, respectively.

    · ForStmt: a for statement; use `ForStmt.getInit()` and `ForStmt.getUpdate()` to access the init and update expressions, respectively.

    · EnhancedForLoop: a for-in or for-of loop; use `EnhancedForLoop.getIterator()` to access the loop iterator (which may be a expression or variable declaration), and `EnhancedForLoop.getIterationDomain()` to access the expression being iterated over.

· ForInStmt, ForOfStmt: a for-in or for-of loop, respectively.

WithStmt: a with statement; use `WithStmt.getExpr()` and `WithStmt.getBody()` to access the controlling expression and the body of the with statement, respectively.

SwitchStmt: a switch statement; use `SwitchStmt.getExpr()` to access the expression on which the statement switches; use `SwitchStmt.getCase(int)` and `SwitchStmt.getACase()` to access individual switch cases; each case is modeled by an entity of class Case, whose member predicates `Case.getExpr()` and `Case.getBodyStmt(int)` provide access to the expression checked by the switch case (which is undefined for `default`), and its body.

TryStmt: a try statement; use `TryStmt.getBody()`, `TryStmt.getCatchClause()` and `TryStmt.getFinally` to access its body, catch clause and finally block, respectively.

– BlockStmt: a block of statements; use `BlockStmt.getStmt(int)` to access the individual statements in the block.

– ExprStmt: an expression statement; use `ExprStmt.getExpr()` to access the expression itself.

– JumpStmt: a statement that disrupts structured control flow, that is, one of `break`, `continue`, `return` and `throw`; use predicate `JumpStmt.getTarget()` to determine the target of the jump, which is either a statement or (for `return` and uncaught `throw` statements) the enclosing function.

BreakStmt: a break statement; use `BreakStmt.getLabel()` to access its (optional) target label.

ContinueStmt: a continue statement; use `ContinueStmt.getLabel()` to access its (optional) target label.

ReturnStmt: a return statement; use `ReturnStmt.getExpr()` to access its (optional) result expression.

ThrowStmt: a throw statement; use `ThrowStmt.getExpr()` to access its thrown expression.

– FunctionDeclStmt: a function declaration statement; see below for available member predicates.

– ClassDeclStmt: a class declaration statement; see below for available member predicates.

– DeclStmt: a declaration statement containing one or more declarators which can be accessed by predicate `DeclStmt.getDeclarator(int)`.

VarDeclStmt, ConstDeclStmt, LetStmt: a `var`, `const` or `let` declaration statement.

• Expr: use `Expr.getEnclosingStmt()` to obtain the innermost statement to which this expression belongs; `Expr.isPure()` determines whether the expression is side-effect-free.

– Identifier: an identifier; use `Identifier.getName()` to obtain its name.

– Literal: a literal value; use `Literal.getValue()` to obtain a string representation of its value, and `Literal.getRawValue()` to obtain its raw source text (including surrounding quotes for string literals).

NullLiteral, BooleanLiteral, NumberLiteral, StringLiteral, RegExpLiteral: different kinds of literals.

– ThisExpr: a this expression.

– SuperExpr: a super expression.

– ArrayExpr: an array expression; use `ArrayExpr.getElement(i)` to obtain the `i`th element expression, and `ArrayExpr.elementIsOmitted(i)` to check whether the `i`th element is omitted.

- ObjectExpr: an object expression; use `ObjectExpr.getProperty(i)` to obtain the `i`th property in the object expression; properties are modeled by class Property, which is described in more detail below.
- FunctionExpr: a function expression; see below for available member predicates.
- ArrowFunctionExpr: an ECMAScript 2015-style arrow function expression; see below for available member predicates.
- ClassExpr: a class expression; see below for available member predicates.
- ParExpr: a parenthesized expression; use `ParExpr.getExpression()` to obtain the operand expression; for any expression, `Expr.stripParens()` can be used to recursively strip off any parentheses
- SeqExpr: a sequence of two or more expressions connected by the comma operator; use `SeqExpr.getOperand(i)` to obtain the `i`th sub-expression.
- ConditionalExpr: a ternary conditional expression; member predicates `ConditionalExpr.getCondition()`, `ConditionalExpr.getConsequent()` and `ConditionalExpr.getAlternate()` provide access to the condition expression, the then expression and the else expression, respectively.
- InvokeExpr: a function call or a new expression; use `InvokeExpr.getCallee()` to obtain the expression specifying the function to be called, and `InvokeExpr.getArgument(i)` to obtain the `i`th argument expression.

  CallExpr: a function call.

  NewExpr: a new expression.

  MethodCallExpr: a function call whose callee expression is a property access; use `MethodCallExpr.getReceiver` to access the receiver expression of the method call, and `MethodCallExpr.getMethodName()` to get the method name (if it can be determined statically).
- PropAccess: a property access, that is, either a dot expression of the form `e.f` or an index expression of the form `e[p]`; use `PropAccess.getBase()` to obtain the base expression on which the property is accessed (`e` in the example), and `PropAccess.getPropertyName()` to determine the name of the accessed property; if the name cannot be statically determined, `getPropertyName()` does not return any value.

  DotExpr: a dot expression.

  IndexExpr: an index expression (also known as computed property access).
- UnaryExpr: a unary expression; use `UnaryExpr.getOperand()` to obtain the operand expression.

  NegExpr (-), PlusExpr (+), LogNotExpr (!), BitNotExpr (~), TypeofExpr, VoidExpr, DeleteExpr, SpreadElement (): various types of unary expressions.
- BinaryExpr: a binary expression; use `BinaryExpr.getLeftOperand()` and `BinaryExpr.getRightOperand()` to access the operand expressions.

  Comparison: any comparison expression.

  - EqualityTest: any equality or inequality test.
  - EqExpr (==), NEqExpr (!=): non-strict equality and inequality tests.
  - StrictEqExpr (===), StrictNEqExpr (!==): strict equality and inequality tests.

· LTExpr (<), LEExpr (<=), GTExpr (>), GEExpr (>=): numeric comparisons.

LShiftExpr (<<), RShiftExpr (>>), URShiftExpr (>>>): shift operators.

AddExpr (+), SubExpr (-), MulExpr (*), DivExpr (/), ModExpr (%), ExpExpr (**): arithmetic operators.

BitOrExpr (|), XOrExpr (^), BitAndExpr (&): bitwise operators.

InExpr: an `in` test.

InstanceofExpr: an `instanceof` test.

LogAndExpr (&&), LogOrExpr (||): short-circuiting logical operators.

– Assignment: assignment expressions, either simple or compound; use `Assignment.getLhs()` and `Assignment.getRhs()` to access the left- and right-hand side, respectively.

AssignExpr: a simple assignment expression.

CompoundAssignExpr: a compound assignment expression.

· AssignAddExpr, AssignSubExpr, AssignMulExpr, AssignDivExpr, AssignModExpr, AssignLShiftExpr, AssignRShiftExpr, AssignURShiftExpr, AssignOrExpr, AssignXOrExpr, AssignAndExpr, AssignExpExpr: different kinds of compound assignment expressions.

– UpdateExpr: an increment or decrement expression; use `UpdateExpr.getOperand()` to obtain the operand expression.

PreIncExpr, PostIncExpr: an increment expression.

PreDecExpr, PostDecExpr: a decrement expression.

– YieldExpr: a yield expression; use `YieldExpr.getOperand()` to access the (optional) operand expression; use `YieldExpr.isDelegating()` to check whether this is a delegating `yield*`.

– TemplateLiteral: an ECMAScript 2015 template literal; `TemplateLiteral.getElement(i)` returns the `i`th element of the template, which may either be an interpolated expression or a constant template element.

– TaggedTemplateExpr: an ECMAScript 2015 tagged template literal; use `TaggedTemplateExpr.getTag()` to access the tagging expression, and `TaggedTemplateExpr.getTemplate()` to access the template literal being tagged.

– TemplateElement: a constant template element; as for literals, use `TemplateElement.getValue()` to obtain the value of the element, and `TemplateElement.getRawValue()` for its raw value

– AwaitExpr: an await expression; use `AwaitExpr.getOperand()` to access the operand expression.

Stmt and Expr share a common superclass ExprOrStmt which is useful for queries that should operate either on statements or on expressions, but not on any other AST nodes.

As an example of how to use expression AST nodes, here is a query that finds expressions of the form `e + f >> g`; such expressions should be rewritten as `(e + f) >> g` to clarify operator precedence:

```
import javascript

from ShiftExpr shift, AddExpr add
where add = shift.getAnOperand()
select add, "This expression should be bracketed to clarify precedence rules."
```

See this in the query console on LGTM.com. When we ran this query on the *meteor/meteor* project in LGTM.com, we found many results where precedence could be clarified using brackets.

Functions

JavaScript provides several ways of defining functions: in ECMAScript 5, there are function declaration statements and function expressions, and ECMAScript 2015 adds arrow function expressions. These different syntactic forms are represented by the classes FunctionDeclStmt (a subclass of Stmt), FunctionExpr (a subclass of Expr) and ArrowFunctionExpr (also a subclass of Expr), respectively. All three are subclasses of Function, which provides common member predicates for accessing function parameters or the function body:

- `Function.getId()` returns the Identifier naming the function, which may not be defined for function expressions.
- `Function.getParameter(i)` and `Function.getAParameter()` access the `i`th parameter or any parameter, respectively; parameters are modeled by the class Parameter, which is a subclass of BindingPattern (see below).
- `Function.getBody()` returns the body of the function, which is usually a Stmt, but may be an Expr for arrow function expressions and legacy expression closures.

As an example, here is a query that finds all expression closures:

```
import javascript

from FunctionExpr fe
where fe.getBody() instanceof Expr
select fe, "Use arrow expressions instead of expression closures."
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects uses expression closures, but you may find this query gets results on other projects.

As another example, this query finds functions that have two parameters that bind the same variable:

```
import javascript

from Function fun, Parameter p, Parameter q, int i, int j
where p = fun.getParameter(i) and
    q = fun.getParameter(j) and
    i < j and
    p.getAVariable() = q.getAVariable()
select fun, "This function has two parameters that bind the same variable."
```

See this in the query console on LGTM.com. None of the LGTM.com demo projects has functions where two parameters bind the same variable.

Classes

Classes can be defined either by class declaration statements, represented by the CodeQL class ClassDeclStmt (which is a subclass of Stmt), or by class expressions, represented by the CodeQL class ClassExpr (which is a subclass of Expr). Both of these classes are also subclasses of ClassDefinition, which provides common member predicates for accessing the name of a class, its superclass, and its body:

- `ClassDefinition.getIdentifier()` returns the Identifier naming the function, which may not be defined for class expressions.
- `ClassDefinition.getSuperClass()` returns the Expr specifying the superclass, which may not be defined.
- `ClassDefinition.getMember(n)` returns the definition of member `n` of this class.
- `ClassDefinition.getMethod(n)` restricts `ClassDefinition.getMember(n)` to methods (as opposed to fields).
- `ClassDefinition.getField(n)` restricts `ClassDefinition.getMember(n)` to fields (as opposed to methods).
- `ClassDefinition.getConstructor()` gets the constructor of this class, possibly a synthetic default constructor.

Note that class fields are not a standard language feature yet, so details of their representation may change.

Method definitions are represented by the class MethodDefinition, which (like its counterpart FieldDefinition for fields) is a subclass of MemberDefinition. That class provides the following important member predicates:

- `MemberDefinition.isStatic()`: holds if this is a static member.
- `MemberDefinition.isComputed()`: holds if the name of this member is computed at runtime.
- `MemberDefinition.getName()`: gets the name of this member if it can be determined statically.
- `MemberDefinition.getInit()`: gets the initializer of this field; for methods, the initializer is a function expressions, for fields it may be an arbitrary expression, and may be undefined.

There are three classes for modeling special methods: ConstructorDefinition models constructors, while GetterMethodDefinition and SetterMethodDefinition model getter and setter methods, respectively.

Declarations and binding patterns

Variables are declared by declaration statements (class DeclStmt), which come in three flavors: `var` statements (represented by class VarDeclStmt), `const` statements (represented by class ConstDeclStmt), and `let` statements (represented by class LetStmt). Every declaration statement has one or more declarators, represented by class VariableDeclarator.

Each declarator consists of a binding pattern, returned by predicate `VariableDeclarator.getBindingPattern()`, and an optional initializing expression, returned by `VariableDeclarator.getInit()`.

Often, the binding pattern is a simple identifier, as in `var x = 42`. In ECMAScript 2015 and later, however, it can also be a more complex destructuring pattern, as in `var [x, y] = arr`.

The various kinds of binding patterns are represented by class BindingPattern and its subclasses:

- VarRef: a simple identifier in an l-value position, for example the `x` in `var x` or in `x = 42`
- Parameter: a function or catch clause parameter
- ArrayPattern: an array pattern, for example, the left-hand side of `[x, y] = arr`
- ObjectPattern: an object pattern, for example, the left-hand side of `{x, y: z} = o`

Here is an example of a query to find declaration statements that declare the same variable more than once, excluding results in minified code:

```
import javascript

from DeclStmt ds, VariableDeclarator d1, VariableDeclarator d2, Variable v, int i, int j
where d1 = ds.getDecl(i) and
    d2 = ds.getDecl(j) and
    i < j and
    v = d1.getBindingPattern().getAVariable() and
    v = d2.getBindingPattern().getAVariable() and
    not ds.getTopLevel().isMinified()
select ds, "Variable " + v.getName() + " is declared both $@ and $@.", d1, "here", d2, "here"
```

See this in the query console on LGTM.com. This is not a common problem, so you may not find any results in your own projects. The *angular/angular.js* project on LGTM.com has one instance of this problem at the time of writing.

> Notice the use of `not ... isMinified()` here and in the next few queries. This excludes any results found in minified code. If you delete `and not ds.getTopLevel().isMinified()` and re-run the query, two results in minified code in the *meteor/meteor* project are reported.

Properties

Properties in object literals are represented by class Property, which is also a subclass of ASTNode, but neither of Expr nor of Stmt.

Class Property has two subclasses ValueProperty and PropertyAccessor, which represent, respectively, normal value properties and getter/setter properties. Class PropertyAccessor, in turn, has two subclasses PropertyGetter and PropertySetter representing getters and setters, respectively.

The predicates `Property.getName()` and `Property.getInit()` provide access to the defined propertys name and its initial value. For PropertyAccessor and its subclasses, `getInit()` is overloaded to return the getter/setter function.

As an example of a query involving properties, consider the following query that flags object expressions containing two identically named properties, excluding results in minified code:

```
import javascript

from ObjectExpr oe, Property p1, Property p2, int i, int j
where p1 = oe.getProperty(i) and
    p2 = oe.getProperty(j) and
    i < j and
    p1.getName() = p2.getName() and
    not oe.getTopLevel().isMinified()
select oe, "Property " + p1.getName() + " is defined both $@ and $@.", p1, "here", p2, "here"
```

See this in the query console on LGTM.com. Many projects have a few instances of object expressions with two identically named properties.

Modules

The JavaScript library has support for working with ECMAScript 2015 modules, as well as legacy CommonJS modules (still commonly employed by Node.js code bases) and AMD-style modules. The classes ES2015Module,

NodeModule, and AMDModule represent these three types of modules, and all three extend the common superclass Module.

The most important member predicates defined by Module are:

- `Module.getName()`: gets the name of the module, which is just the stem (that is, the basename without extension) of the enclosing file.
- `Module.getAnImportedModule()`: gets another module that is imported (through `import` or `require`) by this module.
- `Module.getAnExportedSymbol()`: gets the name of a symbol that this module exports.

Moreover, there is a class Import that models both ECMAScript 2015-style `import` declarations and CommonJS/AMD-style `require` calls; its member predicate `Import.getImportedModule` provides access to the module the import refers to, if it can be determined statically.

Name binding

Name binding is modeled in the JavaScript libraries using four concepts: *scopes*, *variables*, *variable declarations*, and *variable accesses*, represented by the classes Scope, Variable, VarDecl and VarAccess, respectively.

Scopes

In ECMAScript 5, there are three kinds of scopes: the global scope (one per program), function scopes (one per function), and catch clause scopes (one per `catch` clause). These three kinds of scopes are represented by the classes GlobalScope, FunctionScope and CatchScope. ECMAScript 2015 adds block scopes for `let`-bound variables, which are also represented by class Scope, class expression scopes (ClassExprScope), and module scopes (ModuleScope).

Class Scope provides the following API:

- `Scope.getScopeElement()` returns the AST node inducing this scope; undefined for GlobalScope.
- `Scope.getOuterScope()` returns the lexically enclosing scope of this scope.
- `Scope.getAnInnerScope()` returns a scope lexically nested inside this scope.
- `Scope.getVariable(name)`, `Scope.getAVariable()` return a variable declared (implicitly or explicitly) in this scope.

Variables

The Variable class models all variables in a JavaScript program, including global variables, local variables, and parameters (both of functions and `catch` clauses), whether explicitly declared or not.

It is important not to confuse variables and their declarations: local variables may have more than one declaration, while global variables and the implicitly declared local `arguments` variable need not have a declaration at all.

Variable declarations and accesses

Variables may be declared by variable declarators, by function declaration statements and expressions, by class declaration statements or expressions, or by parameters of functions and `catch` clauses. While these declarations differ in their syntactic form, in each case there is an identifier naming the declared variable. We consider that identifier to be the declaration proper, and assign it the class VarDecl. Identifiers that reference a variable, on the other hand, are given the class VarAccess.

The most important predicates involving variables, their declarations, and their accesses are as follows:

- `Variable.getName()`, `VarDecl.getName()`, `VarAccess.getName()` return the name of the variable.
- `Variable.getScope()` returns the scope to which the variable belongs.
- `Variable.isGlobal()`, `Variable.isLocal()`, `Variable.isParameter()` determine whether the variable is a global variable, a local variable, or a parameter variable, respectively.
- `Variable.getAnAccess()` maps a Variable to all VarAccesses that refer to it.
- `Variable.getADeclaration()` maps a Variable to all VarDecls that declare it (of which there may be none, one, or more than one).
- `Variable.isCaptured()` determines whether the variable is ever accessed in a scope that is lexically nested within the scope where it is declared.

As an example, consider the following query which finds distinct function declarations that declare the same variable, that is, two conflicting function declarations within the same scope (again excluding minified code):

```
import javascript

from FunctionDeclStmt f, FunctionDeclStmt g
where f != g and f.getVariable() = g.getVariable() and
    not f.getTopLevel().isMinified() and
    not g.getTopLevel().isMinified()
select f, g
```

See this in the query console on LGTM.com. Some projects declare conflicting functions of the same name and rely on platform-specific behavior to disambiguate the two declarations.

Control flow

A different program representation in terms of intraprocedural control flow graphs (CFGs) is provided by the classes in library CFG.qll.

Class ControlFlowNode represents a single node in the control flow graph, which is either an expression, a statement, or a synthetic control flow node. Note that Expr and Stmt do not inherit from ControlFlowNode at the CodeQL level, although their entity types are compatible, so you can explicitly cast from one to the other if you need to map between the AST-based and the CFG-based program representations.

There are two kinds of synthetic control flow nodes: entry nodes (class ControlFlowEntryNode), which represent the beginning of a top-level or function, and exit nodes (class ControlFlowExitNode), which represent their end. They do not correspond to any AST nodes, but simply serve as the unique entry point and exit point of a control flow graph. Entry and exit nodes can be accessed through the predicates `StmtContainer.getEntry()` and `StmtContainer.getExit()`.

Most, but not all, top-levels and functions have another distinguished CFG node, the *start node*. This is the CFG node at which execution begins. Unlike the entry node, which is a synthetic construct, the start node corresponds to an actual program element: for top-levels, it is the first CFG node of the first statement; for functions, it is the CFG node corresponding to their first parameter or, if there are no parameters, the first CFG node of the body. Empty top-levels do not have a start node.

For most purposes, using start nodes is preferable to using entry nodes.

The structure of the control flow graph is reflected in the member predicates of ControlFlowNode:

- `ControlFlowNode.getASuccessor()` returns a ControlFlowNode that is a successor of this ControlFlowNode in the control flow graph.
- `ControlFlowNode.getAPredecessor()` is the inverse of `getASuccessor()`.
- `ControlFlowNode.isBranch()` determines whether this node has more than one successor.
- `ControlFlowNode.isJoin()` determines whether this node has more than one predecessor.
- `ControlFlowNode.isStart()` determines whether this node is a start node.

Many control-flow-based analyses are phrased in terms of basic blocks rather than single control flow nodes, where a basic block is a maximal sequence of control flow nodes without branches or joins. The class BasicBlock from BasicBlocks.qll represents all such basic blocks. Similar to ControlFlowNode, it provides member predicates `getASuccessor()` and `getAPredecessor()` to navigate the control flow graph at the level of basic blocks, and member predicates `getANode()`, `getNode(int)`, `getFirstNode()` and `getLastNode()` to access individual control flow nodes within a basic block. The predicate `Function.getEntryBB()` returns the entry basic block in a function, that is, the basic block containing the functions entry node. Similarly, `Function.getStartBB()` provides access to the start basic block, which contains the functions start node. As for CFG nodes, `getStartBB()` should normally be preferred over `getEntryBB()`.

As an example of an analysis using basic blocks, `BasicBlock.isLiveAtEntry(v, u)` determines whether variable `v` is live at the entry of the given basic block, and if so binds `u` to a use of `v` that refers to its value at the entry. We can use it to find global variables that are used in a function where they are not live (that is, every read of the variable is preceded by a write), suggesting that the variable was meant to be declared as a local variable instead:

```
import javascript

from Function f, GlobalVariable gv
where gv.getAnAccess().getEnclosingFunction() = f and
    not f.getStartBB().isLiveAtEntry(gv, _)
select f, "This function uses " + gv + " like a local variable."
```

See this in the query console on LGTM.com. Many projects have some variables which look as if they were intended to be local.

Data flow

Definitions and uses

Library DefUse.qll provides classes and predicates to determine def-use relationships between definitions and uses of variables.

Classes VarDef and VarUse contain all expressions that define and use a variable, respectively. For the former, you can use predicate `VarDef.getAVariable()` to find out which variables are defined by a given variable definition (recall that destructuring assignments in ECMAScript 2015 define several variables at the same time). Similarly, predicate `VarUse.getVariable()` returns the (single) variable being accessed by a variable use.

The def-use information itself is provided by predicate `VarUse.getADef()`, that connects a use of a variable to a definition of the same variable, where the definition may reach the use.

As an example, the following query finds definitions of local variables that are not used anywhere; that is, the variable is either not referenced at all after the definition, or its value is overwritten:

```
import javascript

from VarDef def, LocalVariable v
where v = def.getAVariable() and
    not exists (VarUse use | def = use.getADef())
select def, "Dead store of local variable."
```

See this in the query console on LGTM.com. Many projects have some examples of useless assignments to local variables.

SSA

A more fine-grained representation of a programs data flow based on Static Simple Assignment Form (SSA) is provided by the library `semmle.javascript.SSA`.

In SSA form, each use of a local variable has exactly one (SSA) definition that reaches it. SSA definitions are represented by class SsaDefinition. They are not AST nodes, since not every SSA definition corresponds to an explicit element in the source code.

Altogether, there are five kinds of SSA definitions:

1. Explicit definitions (SsaExplicitDefinition): these simply wrap a VarDef, that is, a definition like `x = 1` appearing explicitly in the source code.
2. Implicit initializations (SsaImplicitInit): these represent the implicit initialization of local variables with `undefined` at the beginning of their scope.
3. Phi nodes (SsaPhiNode): these are pseudo-definitions that merge two or more SSA definitions where necessary; see the Wikipedia page linked to above for an explanation.
4. Variable captures (SsaVariableCapture): these are pseudo-definitions appearing at places in the code where the value of a captured variable may change without there being an explicit assignment, for example due to a function call.
5. Refinement nodes (SsaRefinementNode): these are pseudo-definitions appearing at places in the code where something becomes known about a variable; for example, a conditional `if (x === null)` induces a refinement node at the beginning of its then branch recording the fact that `x` is known to be `null` there. (In the literature, these are sometimes known as pi nodes.)

Data flow nodes

Moving beyond just variable definitions and uses, library `semmle.javascript.dataflow.DataFlow` provides a representation of the program as a data flow graph. Its nodes are values of class DataFlow::Node, which has two subclasses `ValueNode` and `SsaDefinitionNode`. Nodes of the former kind wrap an expression or a statement that is considered to produce a value (specifically, a function or class declaration statement, or a TypeScript namespace or enum declaration). Nodes of the latter kind wrap SSA definitions.

You can use the predicate `DataFlow::valueNode` to convert an expression, function or class into its corresponding `ValueNode`, and similarly `DataFlow::ssaDefinitionNode` to map an SSA definition to its corresponding `SsaDefinitionNode`.

There is also an auxiliary predicate `DataFlow::parameterNode` that maps a parameter to its corresponding data flow node. (This is really just a convenience wrapper around `DataFlow::ssaDefinitionNode`, since parameters are also considered to be SSA definitions.)

Going in the other direction, there is a predicate `ValueNode.getAstNode()` for mapping from `ValueNodes` to `ASTNodes`, and `SsaDefinitionNode.getSsaVariable()` for mapping from `SsaDefinitionNodes` to `SsaVariables`. There is also a utility predicate `Node.asExpr()` that gets the underlying expression for a `ValueNode`, and is undefined for all nodes that do not correspond to an expression. (Note in particular that this predicate is not defined for `ValueNodes` wrapping function or class declaration statements!)

You can use the predicate `DataFlow::Node.getAPredecessor()` to find other data flow nodes from which values may flow into this node, and `getASuccessor` for the other direction.

For example, here is a query that finds all invocations of a method called `send` on a value that comes from a parameter named `res`, indicating that it is perhaps sending an HTTP response:

```
import javascript

from SimpleParameter res, DataFlow::Node resNode, MethodCallExpr send
where res.getName() = "res" and
      resNode = DataFlow::parameterNode(res) and
      resNode.getASuccessor+() = DataFlow::valueNode(send.getReceiver()) and
      send.getMethodName() = "send"
select send
```

See this in the query console on LGTM.com. The query finds HTTP response sends in the AMP HTML project.

Note that the data flow modeling in this library is intraprocedural, that is, flow across function calls and returns is *not* modeled. Likewise, flow through object properties and global variables is not modeled.

Type inference

The library `semmle.javascript.dataflow.TypeInference` implements a simple type inference for JavaScript based on intraprocedural, heap-insensitive flow analysis. Basically, the inference algorithm approximates the possible concrete runtime values of variables and expressions as sets of abstract values (represented by the class AbstractValue), each of which stands for a set of concrete values.

For example, there is an abstract value representing all non-zero numbers, and another representing all non-empty strings except for those that can be converted to a number. Both of these abstract values are fairly coarse approximations that represent very large sets of concrete values.

Other abstract values are more precise, to the point where they represent single concrete values: for example, there is an abstract value representing the concrete `null` value, and another representing the number zero.

There is a special group of abstract values called *indefinite* abstract values that represent all concrete values. The analysis uses these to handle expressions for which it cannot infer a more precise value, such as function parameters (as mentioned above, the analysis is intraprocedural and hence does not model argument passing) or property reads (the analysis does not model property values either).

Each indefinite abstract value is associated with a string value describing the cause of imprecision. In the above examples, the indefinite value for the parameter would have cause `"call"`, while the indefinite value for the property would have cause `"heap"`.

To check whether an abstract value is indefinite, you can use the `isIndefinite` member predicate. Its single argument describes the cause of imprecision.

Each abstract value has one or more associated types (CodeQL class InferredType corresponding roughly to the type tags computed by the `typeof` operator. The types are `null`, `undefined`, `boolean`, `number`, `string`, `function`, `class`, `date` and `object`.

To access the results of the type inference, use class DataFlow::AnalyzedNode: any DataFlow::Node can be cast to this class, and additionally there is a convenience predicate `Expr::analyze` that maps expressions directly to their corresponding `AnalyzedNodes`.

Once you have an `AnalyzedNode`, you can use predicate `AnalyzedNode.getAValue()` to access the abstract values inferred for it, and `getAType()` to get the inferred types.

For example, here is a query that looks for `null` checks on expressions that cannot, in fact, be null:

```
import javascript

from StrictEqualityTest eq, DataFlow::AnalyzedNode nd, NullLiteral null
where eq.hasOperands(nd.asExpr(), null) and
      not nd.getAValue().isIndefinite(_) and
      not nd.getAValue() instanceof AbstractNull
select eq, "Spurious null check."
```

To paraphrase, the query looks for equality tests `eq` where one operand is a `null` literal and the other some expression that we convert to an `AnalyzedNode`. If the type inference results for that node are precise (that is, none of the inferred values is indefinite) and (the abstract representation of) `null` is not among them, we flag `eq`.

You can add custom type inference rules by defining new subclasses of `DataFlow::AnalyzedNode` and overriding `getAValue`. You can also introduce new abstract values by extending the abstract class `CustomAbstractValueTag`, which is a subclass of `string`: each string belonging to that class induces a corresponding abstract value of type `CustomAbstractValue`. You can use the predicate `CustomAbstractValue.getTag()` to map from the abstract value to its tag. By implementing the abstract predicates of class `CustomAbstractValueTag` you can define the semantics of your custom abstract values, such as what primitive value they coerce to and what type they have.

Call graph

The JavaScript library implements a simple call graph construction algorithm to statically approximate the possible call targets of function calls and `new` expressions. Due to the dynamically typed nature of JavaScript and its support for higher-order functions and reflective language features, building static call graphs is quite difficult. Simple call graph algorithms tend to be incomplete, that is, they often fail to resolve all possible call targets. More sophisticated algorithms can suffer from the opposite problem of imprecision, that is, they may infer many spurious call targets.

The call graph is represented by the member predicate `getACallee()` of class DataFlow::InvokeNode, which computes possible callees of the given invocation, that is, functions that may at runtime be invoked by this expression.

Furthermore, there are three member predicates that indicate the quality of the callee information for this invocation:

- `DataFlow::InvokeNode.isImprecise()`: holds for invocations where the call graph builder might infer spurious call targets.
- `DataFlow::InvokeNode.isIncomplete()`: holds for invocations where the call graph builder might fail to infer possible call targets.
- `DataFlow::InvokeNode.isUncertain()`: holds if either `isImprecise()` or `isUncertain()` holds.

As an example of a call-graph-based query, here is a query to find invocations for which the call graph builder could not find any callees, despite the analysis being complete for this invocation:

```
import javascript

from DataFlow::InvokeNode invk
where not invk.isIncomplete() and
      not exists(invk.getACallee())
select invk, "Unable to find a callee for this invocation."
```

See this in the query console on LGTM.com

Inter-procedural data flow

The data flow graph-based analyses described so far are all intraprocedural: they do not take flow from function arguments to parameters or from a `return` to the functions caller into account. The data flow library also provides a framework for constructing custom inter-procedural analyses.

We distinguish here between data flow proper, and *taint tracking*: the latter not only considers value-preserving flow (such as from variable definitions to uses), but also cases where one value influences (taints) another without determining it entirely. For example, in the assignment `s2 = s1.substring(i)`, the value of `s1` influences the value of `s2`, because `s2` is assigned a substring of `s1`. In general, `s2` will not be assigned `s1` itself, so there is no data flow from `s1` to `s2`, but `s1` still taints `s2`.

The simplest way of implementing an interprocedural data flow analysis is to extend either class `DataFlow::TrackedNode` or `DataFlow::TrackedExpr`. The former is a subclass of `DataFlow::Node`, the latter of `Expr`, and extending them ensures that the newly added values are tracked interprocedurally. You can use the predicate `flowsTo` to find out which nodes/expressions the tracked value flows to.

For example, suppose that we are developing an analysis to find hard-coded passwords. We might start by writing a simple query that looks for string constants flowing into variables named `"password"`. To do this, we can extend `TrackedExpr` to track all constant strings, `flowsTo` to find cases where such a string flows into a (SSA) definition of a password variable:

```
import javascript

class TrackedStringLiteral extends DataFlow::TrackedNode {
    TrackedStringLiteral() {
        this.asExpr() instanceof ConstantString
    }
}

from TrackedStringLiteral source, DataFlow::Node sink, SsaExplicitDefinition def
where source.flowsTo(sink) and sink = DataFlow::ssaDefinitionNode(def) and
      def.getSourceVariable().getName().toLowerCase() = "password"
select sink
```

Note that `TrackedNode` and `TrackedExpr` do not restrict the set of sinks for the inter-procedural flow analysis, tracking flow into any expression that they might flow to. This can be expensive for large code bases, and is often unnecessary, since usually you are only interested in flow to a particular set of sinks. For example, the above query only looks for flow into assignments to password variables.

This is a particular instance of a general pattern, whereby we want to specify a data flow or taint analysis in

terms of its *sources* (where flow starts), *sinks* (where it should be tracked), and *barriers* or *sanitizers* (where flow is interrupted). The example does not include any sanitizers, but they are very common in security analyses: for example, an analysis that tracks the flow of untrusted user input into, say, a SQL query has to keep track of code that validates the input, thereby making it safe to use. Such a validation step is an example of a sanitizer.

The classes `DataFlow::Configuration` and `TaintTracking::Configuration` allow specifying a data flow or taint analysis, respectively, by overriding the following predicates:

- `isSource(DataFlow::Node nd)` selects all nodes `nd` from where flow tracking starts.
- `isSink(DataFlow::Node nd)` selects all nodes `nd` to which the flow is tracked.
- `isBarrier(DataFlow::Node nd)` selects all nodes `nd` that act as a barrier for data flow; `isSanitizer` is the corresponding predicate for taint tracking configurations.
- `isBarrierEdge(DataFlow::Node src, DataFlow::Node trg)` is a variant of `isBarrier(nd)` that allows specifying barrier *edges* in addition to barrier nodes; again, `isSanitizerEdge` is the corresponding predicate for taint tracking;
- `isAdditionalFlowStep(DataFlow::Node src, DataFlow::Node trg)` allows specifying custom additional flow steps for this analysis; `isAdditionalTaintStep` is the corresponding predicate for taint tracking configurations.

Since for technical reasons both `Configuration` classes are subtypes of `string`, you have to choose a unique name for each flow configuration and equate `this` with it in the characteristic predicate (as in the example below).

The predicate `Configuration.hasFlow` performs the actual flow tracking, starting at a source and looking for flow to a sink that does not pass through a barrier node or edge.

To continue with our above example, we can phrase it as a data flow configuration as follows:

```
class PasswordTracker extends DataFlow::Configuration {
    PasswordTracker() {
        // unique identifier for this configuration
        this = "PasswordTracker"
    }

    override predicate isSource(DataFlow::Node nd) {
        nd.asExpr() instanceof StringLiteral
    }

    override predicate isSink(DataFlow::Node nd) {
        passwordVarAssign(_, nd)
    }

    predicate passwordVarAssign(Variable v, DataFlow::Node nd) {
        exists (SsaExplicitDefinition def |
            nd = DataFlow::ssaDefinitionNode(def) and
            def.getSourceVariable() = v and
            v.getName().toLowerCase() = "password"
        )
    }
}
```

Now we can rephrase our query to use `Configuration.hasFlow`:

```
from PasswordTracker pt, DataFlow::Node source, DataFlow::Node sink, Variable v
where pt.hasFlow(source, sink) and pt.passwordVarAssign(v, sink)
select sink, "Password variable " + v + " is assigned a constant string."
```

Note that while analyses implemented in this way are inter-procedural in that they track flow and taint across function calls and returns, flow through global variables is not tracked. Flow through object properties is only tracked in limited cases, for example through properties of object literals or CommonJS `module` and `exports` objects.

Syntax errors

JavaScript code that contains syntax errors cannot usually be analyzed. For such code, the lexical and syntactic representations are not available, and hence no name binding information, call graph or control and data flow. All that is available in this case is a value of class JSParseError representing the syntax error. It provides information about the syntax error location (JSParseError is a subclass of Locatable) and the error message through predicate `JSParseError.getMessage`.

Note that for some very simple syntax errors the parser can recover and continue parsing. If this happens, lexical and syntactic information is available in addition to the JSParseError values representing the (recoverable) syntax errors encountered during parsing.

Frameworks

AngularJS

The `semmle.javascript.frameworks.AngularJS` library provides support for working with AngularJS (Angular 1.x) code. Its most important classes are:

- AngularJS::AngularModule: an Angular module
- AngularJS::DirectiveDefinition, AngularJS::FactoryRecipeDefinition, AngularJS::FilterDefinition, AngularJS::ControllerDefinition: a definition of a directive, service, filter or controller, respectively
- AngularJS::InjectableFunction: a function that is subject to dependency injection

HTTP framework libraries

The library `semmle.javacript.frameworks.HTTP` provides classes modeling common concepts from various HTTP frameworks.

Currently supported frameworks are Express, the standard Node.js `http` and `https` modules, Connect, Koa, Hapi and Restify.

The most important classes include (all in module `HTTP`):

- `ServerDefinition`: an expression that creates a new HTTP server.
- `RouteHandler`: a callback for handling an HTTP request.
- `RequestExpr`: an expression that may contain an HTTP request object.
- `ResponseExpr`: an expression that may contain an HTTP response object.
- `HeaderDefinition`: an expression that sets one or more HTTP response headers.
- `CookieDefinition`: an expression that sets a cookie in an HTTP response.

- `RequestInputAccess`: an expression that accesses user-controlled request data.

For each framework library, there is a corresponding CodeQL library (for example `semmle.javascript.frameworks.Express`) that instantiates the above classes for that framework and adds framework-specific classes.

Node.js

The `semmle.javascript.NodeJS` library provides support for working with Node.js modules through the following classes:

- NodeModule: a top-level that defines a Node.js module; see the section on *Modules* for more information.
- Require: a call to the special `require` function that imports a module.

As an example of the use of these classes, here is a query that counts for every module how many other modules it imports:

```
import javascript

from NodeModule m
select m, count(m.getAnImportedModule())
```

See this in the query console on LGTM.com. When you analyze a project, for each module you can see how many other modules it imports.

NPM

The `semmle.javascript.NPM` library provides support for working with NPM packages through the following classes:

- PackageJSON: a `package.json` file describing an NPM package; various getter predicates are available for accessing detailed information about the package, which are described in the online API documentation.
- BugTrackerInfo, ContributorInfo, RepositoryInfo: these classes model parts of the `package.json` file providing information on bug tracking systems, contributors and repositories.
- PackageDependencies: models the dependencies of an NPM package; the predicate `PackageDependencies.getADependency(pkg, v)` binds `pkg` to the name and `v` to the version of a package required by a `package.json` file.
- NPMPackage: a subclass of Folder that models an NPM package; important member predicates include:
  - `NPMPackage.getPackageName()` returns the name of this package.
  - `NPMPackage.getPackageJSON()` returns the `package.json` file for this package.
  - `NPMPackage.getNodeModulesFolder()` returns the `node_modules` folder for this package.
  - `NPMPackage.getAModule()` returns a Node.js module belonging to this package (not including modules in the `node_modules` folder).

As an example of the use of these classes, here is a query that identifies unused dependencies, that is, module dependencies that are listed in the `package.json` file, but which are not imported by any `require` call:

```
import javascript

from NPMPackage pkg, PackageDependencies deps, string name
where deps = pkg.getPackageJSON().getDependencies() and
deps.getADependency(name, _) and
not exists (Require req | req.getTopLevel() = pkg.getAModule() | name = req.getImportedPath().
↪getValue())
select deps, "Unused dependency '" + name + "'."
```

See this in the query console on LGTM.com. It is not uncommon for projects to have some unused dependencies.

React

The `semmle.javascript.frameworks.React` library provides support for working with React code through the ReactComponent class, which models a React component defined either in the functional style or the class-based style (both ECMAScript 2015 classes and old-style `React.createClass` classes are supported).

Databases

The class `SQL::SqlString` represents an expression that is interpreted as a SQL command. Currently, we model SQL commands issued through the following npm packages: mysql, pg, pg-pool, sqlite3, mssql and sequelize.

Similarly, the class `NoSQL::Query` represents an expression that is interpreted as a NoSQL query by the `mongodb` or `mongoose` package.

Finally, the class `DatabaseAccess` contains all data flow nodes that perform a database access using any of the packages above.

For example, here is a query to find SQL queries that use string concatenation (instead of a templating-based solution, which is usually safer):

```
import javascript

from SQL::SqlString ss
where ss instanceof AddExpr
select ss, "Use templating instead of string concatenation."
```

See this in the query console on LGTM.com, showing two (benign) results on strong-arc.

Miscellaneous

Externs

The `semmle.javascript.Externs` library provides support for working with externs through the following classes:

- ExternalDecl: common superclass modeling all different kinds of externs declarations; it defines two member predicates:
  - `ExternalDecl.getQualifiedName()` returns the fully qualified name of the declared entity.
  - `ExternalDecl.getName()` returns the unqualified name of the declared entity.

- ExternalTypedef: a subclass of ExternalDecl representing type declarations; unlike other externs declarations, such declarations do not declare a function or object that is present at runtime, but simply introduce an alias for a type.
- ExternalVarDecl: a subclass of ExternalDecl representing a variable or function declaration; it defines two member predicates:
  - `ExternalVarDecl.getInit()` returns the initializer associated with this declaration, if any; this can either be a Function or an Expr.
  - `ExternalVarDecl.getDocumentation()` returns the JSDoc comment associated with this declaration.

Variables and functions declared in an externs file are either globals (represented by class ExternalGlobalDecl), or members (represented by class ExternalMemberDecl).

Members are further subdivided into static members (class ExternalStaticMemberDecl) and instance members (class ExternalInstanceMemberDecl).

For more details on these and other classes representing externs, see the API documentation.

HTML

The `semmle.javascript.HTML` library provides support for working with HTML documents. They are represented as a tree of `HTML::Element` nodes, each of which may have zero or more attributes represented by class `HTML::Attribute`.

Similar to the abstract syntax tree representation, `HTML::Element` has member predicates `getChild(i)` and `getParent()` to navigate from an element to its `i`th child element and its parent element, respectively. Use predicate `HTML::Element.getAttribute(i)` to get the `i`th attribute of the element, and `HTML::Element.getAttributeByName(n)` to get the attribute with name `n`.

For `HTML::Attribute`, predicates `getName()` and `getValue()` provide access to the attributes name and value, respectively.

Both `HTML::Element` and `HTML::Attribute` have a predicate `getRoot()` that gets the root `HTML::Element` of the document to which they belong.

JSDoc

The `semmle.javascript.JSDoc` library provides support for working with JSDoc comments. Documentation comments are parsed into an abstract syntax tree representation closely following the format employed by the Doctrine JSDoc parser.

A JSDoc comment as a whole is represented by an entity of class JSDoc, while individual tags are represented by class JSDocTag. Important member predicates of these two classes include:

- `JSDoc.getDescription()` returns the descriptive header of the JSDoc comment, if any.
- `JSDoc.getComment()` maps the JSDoc entity to its underlying Comment entity.
- `JSDocTag.getATag()` returns a tag in this JSDoc comment.
- `JSDocTag.getTitle()` returns the title of his tag; for instance, an `@param` tag has title `"param"`.
- `JSDocTag.getName()` returns the name of the parameter or variable documented by this tag.
- `JSDocTag.getType()` returns the type of the parameter or variable documented by this tag.

- `JSDocTag.getDescription()` returns the description associated with this tag.

Types in JSDoc comments are represented by the class JSDocTypeExpr and its subclasses, which again represent type expressions as abstract syntax trees. Examples of type expressions are JSDocAnyTypeExpr, representing the any type `*`, or JSDocNullTypeExpr, representing the null type.

As an example, here is a query that finds `@param` tags that do not specify the name of the documented parameter:

```
import javascript

from JSDocTag t
where t.getTitle() = "param" and
not exists(t.getName())
select t, "@param tag is missing name."
```

See this in the query console on LGTM.com. Of the LGTM.com demo projects analyzed, only *Semantic-Org/Semantic-UI* has an example where the `@param` tag omits the name.

For full details on these and other classes representing JSDoc comments and type expressions, see the API documentation.

JSX

The `semmle.javascript.JSX` library provides support for working with JSX code.

Similar to the representation of HTML documents, JSX fragments are modeled as a tree of JSXElements, each of which may have zero or more JSXAttributes.

However, unlike HTML, JSX is interleaved with JavaScript, hence JSXElement is a subclass of Expr. Like `HTML::Element`, it has predicates `getAttribute(i)` and `getAttributeByName(n)` to look up attributes of a JSX element. Its body elements can be accessed by predicate `getABodyElement()`; note that the results of this predicate are arbitrary expressions, which may either be further JSXElements, or other expressions that are interpolated into the body of the outer element.

JSXAttribute, again not unlike `HTML::Attribute`, has predicates `getName()` and `getValue()` to access the attribute name and value.

JSON

The `semmle.javascript.JSON` library provides support for working with JSON files that were processed by the JavaScript extractor when building the CodeQL database.

JSON files are modeled as trees of JSON values. Each JSON value is represented by an entity of class JSONValue, which provides the following member predicates:

- `JSONValue.getParent()` returns the JSON object or array in which this value occurs.
- `JSONValue.getChild(i)` returns the `i`th child of this JSON object or array.

Note that JSONValue is a subclass of Locatable, so the usual member predicates of Locatable can be used to determine the file in which a JSON value appears, and its location within that file.

Class JSONValue has the following subclasses:

- JSONPrimitiveValue: a JSON-encoded primitive value; use `JSONPrimitiveValue.getValue()` to obtain a string representation of the value.

  - JSONNull, JSONBoolean, JSONNumber, JSONString: subclasses of JSONPrimitiveValue representing the various kinds of primitive values.

- JSONArray: a JSON-encoded array; use `JSONArray.getElementValue(i)` to access the `i`th element of the array.
- JSONObject: a JSON-encoded object; use `JSONObject.getValue(n)` to access the value of property `n` of the object.

Regular expressions

The `semmle.javascript.Regexp` library provides support for working with regular expression literals. The syntactic structure of regular expression literals is represented as an abstract syntax tree of regular expression terms, modeled by the class RegExpTerm. Similar to ASTNode, class RegExpTerm provides member predicates `getParent()` and `getChild(i)` to navigate the structure of the syntax tree.

Various subclasses of RegExpTerm model different kinds of regular expression constructs and operators; see the API documentation for details.

YAML

The `semmle.javascript.YAML` library provides support for working with YAML files that were processed by the JavaScript extractor when building the CodeQL database.

YAML files are modeled as trees of YAML nodes. Each YAML node is represented by an entity of class YAMLNode, which provides, among others, the following member predicates:

- `YAMLNode.getParentNode()` returns the YAML collection in which this node is syntactically nested.
- `YAMLNode.getChildNode(i)` returns the `i`th child node of this node, `YAMLNode.getAChildNode()` returns any child node of this node.
- `YAMLNode.getTag()` returns the tag of this YAML node.
- `YAMLNode.getAnchor()` returns the anchor associated with this YAML node, if any.
- `YAMLNode.eval()` returns the YAMLValue this YAML node evaluates to after resolving aliases and includes.

The various kinds of scalar values available in YAML are represented by classes YAMLInteger, YAMLFloat, YAMLTimestamp, YAMLBool, YAMLNull and YAMLString. Their common superclass is YAMLScalar, which has a member predicate `getValue()` to obtain the value of a scalar as a string.

YAMLMapping and YAMLSequence represent mappings and sequences, respectively, and are subclasses of YAMLCollection.

Alias nodes are represented by class YAMLAliasNode, while YAMLMergeKey and YAMLInclude represent merge keys and `!include` directives, respectively.

Predicate `YAMLMapping.maps(key, value)` models the key-value relation represented by a mapping, taking merge keys into account.

7.2.3 Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript

- QL language reference
- CodeQL tools

7.3 CodeQL library for TypeScript

When youre analyzing a TypeScript program, you can make use of the large collection of classes in the CodeQL library for TypeScript.

7.3.1 Overview

Support for analyzing TypeScript code is bundled with the CodeQL libraries for JavaScript, so you can include the full TypeScript library by importing the `javascript.qll` module:

```
import javascript
```

CodeQL libraries for JavaScript covers most of this library, and is also relevant for TypeScript analysis. This document supplements the JavaScript documentation with the TypeScript-specific classes and predicates.

7.3.2 Syntax

Most syntax in TypeScript is represented in the same way as its JavaScript counterpart. For example, `a+b` is represented by an AddExpr; the same as it would be in JavaScript. On the other hand, `x as number` is represented by TypeAssertion, a class that is specific to TypeScript.

Type annotations

The TypeExpr class represents anything that is part of a type annotation.

Only type annotations that are explicit in the source code occur as a `TypeExpr`. Types inferred by the TypeScript compiler are `Type` entities; for details about this, see the section on *static type information*.

There are several ways to access type annotations, for example:

- `VariableDeclaration.getTypeAnnotation()`
- `Function.getReturnTypeAnnotation()`
- `BindingPattern.getTypeAnnotation()`
- `Parameter.getTypeAnnotation()` (special case of `BindingPattern.getTypeAnnotation()`)
- `VarDecl.getTypeAnnotation()` (special case of `BindingPattern.getTypeAnnotation()`)
- `FieldDeclaration.getTypeAnnotation()`

The TypeExpr class provides some convenient member predicates such as `isString()` and `isVoid()` to recognize commonly used types.

The subclasses that represent type annotations are:

- TypeAccess: a name referring to a type, such as `Date` or `http.ServerRequest`.
  - LocalTypeAccess: an unqualified name, such as `Date`.
  - QualifiedTypeAccess: a name prefixed by a namespace, such as `http.ServerRequest`.
  - ImportTypeAccess: an `import` used as a type, such as `import("./foo")`.

- PredefinedTypeExpr: a predefined type, such as `number`, `string`, `void`, or `any`.
- ThisTypeExpr: the `this` type.
- InterfaceTypeExpr, also known as a literal type, such as `{x: number}`.
- FunctionTypeExpr: a type such as `(x: number) => string`.
- GenericTypeExpr: a named type with type arguments, such as `Array<string>`.
- LiteralTypeExpr: a string, number, or boolean constant used as a type, such as `'foo'`.
- ArrayTypeExpr: a type such as `string[]`.
- UnionTypeExpr: a type such as `string | number`.
- IntersectionTypeExpr: a type such as `S & T`.
- IndexedAccessTypeExpr: a type such as `T[K]`.
- ParenthesizedTypeExpr: a type such as `(string)`.
- TupleTypeExpr: a type such as `[string, number]`.
- KeyofTypeExpr: a type such as `keyof T`.
- TypeofTypeExpr: a type such as `typeof x`.
- IsTypeExpr: a type such as `x is string`.
- MappedTypeExpr: a type such as `{ [K in C]: T }`.

There are some subclasses that may be part of a type annotation, but are not themselves types:

- TypeParameter: a type parameter declared on a type or function, such as `T` in `class C<T> {}`.
- NamespaceAccess: a name referring to a namespace from inside a type, such as `http` in `http.ServerRequest`.
  - LocalNamespaceAccess: the initial identifier in a prefix, such as `http` in `http.ServerRequest`.
  - QualifiedNamespaceAccess: a qualified name in a prefix, such as `net.client` in `net.client.Connection`.
  - ImportNamespaceAccess: an `import` used as a namespace in a type, such as in `import("http").ServerRequest`.
- VarTypeAccess: a reference to a value from inside a type, such as `x` in `typeof x` or `x is string`.

Function signatures

The Function class is a broad class that includes both concrete functions and function signatures.

Function signatures can take several forms:

- Function types, such as `(x: number) => string`.
- Abstract methods, such as `abstract foo(): void`.
- Overload signatures, such as `foo(x: number): number` followed by an implementation of `foo`.
- Call signatures, such as in `{ (x: string): number }`.
- Index signatures, such as in `{ [x: string]: number }`.

- Functions in an ambient context, such as `declare function foo(x: number): string`.

We recommend that you use the predicate `Function.hasBody()` to distinguish concrete functions from signatures.

Type parameters

The TypeParameter class represents type parameters, and the TypeParameterized class represents entities that can declare type parameters. Classes, interfaces, type aliases, functions, and mapped type expressions are all `TypeParameterized`.

You can access type parameters using the following predicates:

- `TypeParameterized.getTypeParameter(n)` gets the nth declared type parameter.
- `TypeParameter.getHost()` gets the entity declaring a given type parameter.

You can access type arguments using the following predicates:

- `GenericTypeExpr.getTypeArgument(n)` gets the nth type argument of a type.
- `TypeAccess.getTypeArgument(n)` is a convenient alternative for the above (a TypeAccess with type arguments is wrapped in a GenericTypeExpr).
- `InvokeExpr.getTypeArgument(n)` gets the nth type argument of a call.
- `ExpressionWithTypeArguments.getTypeArgument(n)` gets the nth type argument of a generic superclass expression.

To select references to a given type parameter, use `getLocalTypeName()` (see *Name binding* below).

Examples

Select expressions that cast a value to a type parameter:

```
import javascript

from TypeParameter param, TypeAssertion assertion
where assertion.getTypeAnnotation() = param.getLocalTypeName().getAnAccess()
select assertion, "Cast to type parameter."
```

See this in the query console on LGTM.com.

Classes and interfaces

The CodeQL class ClassOrInterface is a common supertype of classes and interfaces, and provides some TypeScript-specific member predicates:

- `ClassOrInterface.isAbstract()` holds if this is an interface or a class with the `abstract` modifier.
- `ClassOrInterface.getASuperInterface()` gets a type from the `implements` clause of a class or from the `extends` clause of an interface.
- `ClassOrInterface.getACallSignature()` gets a call signature of an interface, such as in `{ (arg: string): number }`.
- `ClassOrInterface.getAnIndexSignature()` gets an index signature, such as in `{ [key: string]: number }`.

- `ClassOrInterface.getATypeParameter()` gets a declared type parameter (special case of `TypeParameterized.getATypeParameter()`).

Note that the superclass of a class is an expression, not a type annotation. If the superclass has type arguments, it will be an expression of kind ExpressionWithTypeArguments.

Also see the documentation for classes in the CodeQL libraries for JavaScript.

To select the type references to a class or an interface, use `getTypeName()`.

Statements

The following are TypeScript-specific statements:

- NamespaceDeclaration: a statement such as `namespace M {}`.
- EnumDeclaration: a statement such as `enum Color { red, green, blue }`.
- TypeAliasDeclaration: a statement such as `type A = number`.
- InterfaceDeclaration: a statement such as `interface Point { x: number; y: number; }`.
- ImportEqualsDeclaration: a statement such as `import fs = require("fs")`.
- ExportAssignDeclaration: a statement such as `export = M`.
- ExportAsNamespaceDeclaration: a statement such as `export as namespace M`.
- ExternalModuleDeclaration: a statement such as `module "foo" {}`.
- GlobalAugmentationDeclaration: a statement such as `global {}`

Expressions

The following are TypeScript-specific expressions:

- ExpressionWithTypeArguments: occurs when the `extends` clause of a class has type arguments, such as in `class C extends D<string>`.
- TypeAssertion: asserts that a value has a given type, such as `x as number` or `<number> x`.
- NonNullAssertion: asserts that a value is not null or undefined, such as `x!`.
- ExternalModuleReference: a `require` call on the right-hand side of an import-assign, such as `import fs = require("fs")`.

Ambient declarations

Type annotations, interfaces, and type aliases are considered ambient AST nodes, as is anything with a `declare` modifier.

The predicate `ASTNode.isAmbient()` can be used to determine if an AST node is ambient.

Ambient nodes are mostly ignored by control flow and data flow analysis. The outermost part of an ambient declaration has a single no-op node in the control flow graph, and it has no internal control flow.

7.3.3 Static type information

Static type information and global name binding is available for projects with full TypeScript extraction enabled. This option is enabled by default for projects on LGTM.com and when you create databases with the CodeQL CLI.

> Note
>
> If you are using the legacy QL command-line tools, you must enable full TypeScript extraction by passing `--typescript-full` to the JavaScript extractor. For further information on customizing calls to the extractor, see Customizing JavaScript extraction.
>
> Without full extraction, the classes and predicates described in this section are empty.

Basic usage

The Type class represents a static type, such as `number` or `string`. The type of an expression can be obtained with `Expr.getType()`.

Types that refer to a specific named type can be recognized in various ways:

- `type.(TypeReference).hasQualifiedName(name)` holds if the type refers to the given named type.
- `type.(TypeReference).hasUnderlyingType(name)` holds if the type refers to the given named type or a transitive subtype thereof.
- `type.hasUnderlyingType(name)` is like the above, but additionally holds if the reference is wrapped in a union and/or intersection type.

The `hasQualifiedName` and `hasUnderlyingType` predicates have two overloads:

- The single-argument version takes a qualified name relative to the global scope.
- The two-argument version takes the name of a module and qualified name relative to that module.

Example

The following query can be used to find all `toString` calls on a Node.js `Buffer` object:

```
import javascript

from MethodCallExpr call
where call.getReceiver().getType().hasUnderlyingType("Buffer")
  and call.getMethodName() = "toString"
select call
```

Working with types

`Type` entities are not associated with a specific source location. For instance, there can be many uses of the `number` keyword, but there is only one `number` type.

Some important member predicates of `Type` are:

- `Type.getProperty(name)` gets the type of a named property.
- `Type.getMethod(name)` gets the signature of a named method.
- `Type.getSignature(kind,n)` gets the nth overload of a call or constructor signature.
- `Type.getStringIndexType()` gets the type of the string index signature.

- `Type.getNumberIndexType()` gets the type of the number index signature.

A `Type` entity always belongs to exactly one of the following subclasses:

- `TypeReference`: a named type, possibly with type arguments.
- `UnionType`: a union type such as `string | number`.
- `IntersectionType`: an intersection type such as `T & U`.
- `TupleType`: a tuple type such as `[string, number]`.
- `StringType`: the `string` type.
- `NumberType`: the `number` type.
- `AnyType`: the `any` type.
- `NeverType`: the `never` type.
- `VoidType`: the `void` type.
- `NullType`: the `null` type.
- `UndefinedType`: the `undefined` type.
- `ObjectKeywordType`: the `object` type.
- `SymbolType`: a `symbol` or `unique symbol` type.
- `AnonymousInterfaceType`: an anonymous type such as `{x: number}`.
- `TypeVariableType`: a reference to a type variable.
- `ThisType`: the `this` type within a specific type.
- `TypeofType`: the type of a named value, such as `typeof X`.
- `BooleanLiteralType`: the `true` or `false` type.
- `StringLiteralType`: the type of a string constant.
- `NumberLiteralType`: the type of a number constant.

Additionally, `Type` has the following subclasses which overlap partially with those above:

- `BooleanType`: the type `boolean`, internally represented as the union type `true | false`.
- `PromiseType`: a type that describes a promise such as `Promise<T>`.
- `ArrayType`: a type that describes an array object, possibly a tuple type.
  - `PlainArrayType`: a type of form `Array<T>`.
  - `ReadonlyArrayType`: a type of form `ReadonlyArray<T>`.
- `LiteralType`: a boolean, string, or number literal type.
- `NumberLikeType`: the `number` type or a number literal type.
- `StringLikeType`: the `string` type or a string literal type.
- `BooleanLikeType`: the `true`, `false`, or `boolean` type.

Canonical names and named types

`CanonicalName` is a CodeQL class representing a qualified name relative to a root scope, such as a module or the global scope. It typically represents an entity such as a type, namespace, variable, or function. `TypeName` and `Namespace` are subclasses of this class.

Canonical names can be recognized using the `hasQualifiedName` predicate:

- `hasQualifiedName(name)` holds if the qualified name is `name` relative to the global scope.
- `hasQualifiedName(module,name)` holds if the qualified name is `name` relative to the given module name.

For convenience, this predicate is also available on other classes, such as `TypeReference` and `TypeofType`, where it forwards to the underlying canonical name.

Function types

There is no CodeQL class for function types, as any type with a call or construct signature is usable as a function. The type `CallSignatureType` represents such a signature (with or without the `new` keyword).

Signatures can be obtained in several ways:

- `Type.getFunctionSignature(n)` gets the nth overloaded function signature.
- `Type.getConstructorSignature(n)` gets the nth overloaded constructor signature.
- `Type.getLastFunctionSignature()` gets the last declared function signature.
- `Type.getLastConstructorSignature()` gets the last declared constructor signature.

Some important member predicates of `CallSignatureType` are:

- `CallSignatureType.getParameter(n)` gets the type of the nth parameter.
- `CallSignatureType.getParameterName(n)` gets the name of the nth parameter.
- `CallSignatureType.getReturnType()` gets the return type.

Note that a signature is not associated with a specific declaration site.

Call resolution

Additional type information is available for invocation expressions:

- `InvokeExpr.getResolvedCallee()` gets the callee as a concrete `Function`.
- `InvokeExpr.getResolvedCalleeName()` get the callee as a canonical name.
- `InvokeExpr.getResolvedSignature()` gets the signature of the invoked function, with overloading resolved and type arguments substituted.

Note that these refer to the call target as determined by the type system. The actual call target may differ at runtime, for instance, if the target is a method that has been overridden in a subclass.

Inheritance and subtyping

The declared supertypes of a named type can be obtained using `TypeName.getABaseTypeName()`.

This operates at the level of type names, hence the specific type arguments used in the inheritance chain are not available. However, these can often be deduced using `Type.getProperty` or `Type.getMethod` which both take inheritance into account.

This only accounts for types explicitly mentioned in the `extends` or `implements` clause of a type. There is no predicate that determines subtyping or assignability between types in general.

The following two predicates can be useful for recognising subtypes of a given type:

- `Type.unfold()` unfolds unions and/or intersection types and get the underlying types, or the type itself if it is not a union or intersection.
- `Type.hasUnderlyingType(name)` holds if the type is a reference to the given named type, possibly after unfolding unions/intersections and following declared supertypes.

Example

The following query can be used to find all classes that are React components, along with the type of their `props` property, which generally coincides with its first type argument:

```
import javascript

from ClassDefinition cls, TypeName name
where name = cls.getTypeName()
  and name.getABaseTypeName+().hasQualifiedName("React.Component")
select cls, name.getType().getProperty("props")
```

7.3.4 Name binding

In TypeScript, names can refer to variables, types, and namespaces, or a combination of these.

These concepts are modeled as distinct entities: Variable, TypeName, and Namespace. For example, the class `C` below introduces both a variable and a type:

```
class C {}
let x = C; // refers to the variable C
let y: C;  // refers to the type C
```

The variable `C` and the type `C` are modeled as distinct entities. One is a Variable, the other is a TypeName.

TypeScript also allows you to import types and namespaces, and give them local names in different scopes. For example, the import below introduces a local type name `B`:

```
import {C as B} from "./foo"
```

The local name `B` is represented as a LocalTypeName named `B`, restricted to just the file containing the import. An import statement can also introduce a Variable and a LocalNamespaceName.

The following table shows the relevant classes for working with each kind of name. The classes are described in more detail below.

Kind	Local alias	Canonical name	Definition	Access
Value	Variable			VarAccess
Type	LocalTypeName	TypeName	TypeDefinition	TypeAccess
Namespace	LocalNamespaceName	Namespace	NamespaceDefinition	NamespaceAccess

Note: `TypeName` and `Namespace` are only populated if the database is generated using full TypeScript extraction. `LocalTypeName` and `LocalNamespaceName` are always populated.

Type names

A TypeName is a qualified name for a type and is not bound to a specific lexical scope. The TypeDefinition class represents an entity that defines a type, namely a class, interface, type alias, enum, or enum member. The relevant predicates for working with type names are:

- `TypeAccess.getTypeName()` gets the qualified name being referenced (if any).
- `TypeDefinition.getTypeName()` gets the qualified name of a class, interface, type alias, enum, or enum member.
- `TypeName.getAnAccess()`, gets an access to a given type.
- `TypeName.getADefinition()`, get a definition of a given type. Note that interfaces can have multiple definitions.

A LocalTypeName behaves like a block-scoped variable, that is, it has an unqualified name and is restricted to a specific scope. The relevant predicates are:

- `LocalTypeAccess.getLocalTypeName()` gets the local name referenced by an unqualified type access.
- `LocalTypeName.getAnAccess()` gets an access to a local type name.
- `LocalTypeName.getADeclaration()` gets a declaration of this name.
- `LocalTypeName.getTypeName()` gets the qualified name to which this name refers.

Examples

Find references that omit type arguments to a generic type.

It is best to use TypeName to resolve through imports and qualified names:

```
import javascript

from TypeDefinition def, TypeAccess access
where access.getTypeName().getADefinition() = def
  and def.(TypeParameterized).hasTypeParameters()
  and not access.hasTypeArguments()
select access, "Type arguments are omitted"
```

See this in the query console on LGTM.com.

Find imported names that are used as both a type and a value:

```
import javascript

from ImportSpecifier spec
where exists (LocalTypeAccess access | access.getLocalTypeName().getADeclaration() = spec.
↪getLocal())
  and exists (VarAccess access | access.getVariable().getADeclaration() = spec.getLocal())
select spec, "Used as both variable and type"
```

See this in the query console on LGTM.com.

Namespace names

Namespaces are represented by the classes Namespace and LocalNamespaceName. The NamespaceDefinition class represents a syntactic definition of a namespace, which includes ordinary namespace declarations as well as enum declarations.

Note that these classes deal exclusively with namespaces referenced from inside type annotations, not through expressions.

A Namespace is a qualified name for a namespace, and is not bound to a specific scope. The relevant predicates for working with namespaces are:

- `NamespaceAccess.getNamespace()` gets the namespace being referenced by a namespace access.
- `NamespaceDefinition.getNamespace()` gets the namespace defined by a namespace or enum declaration.
- `Namespace.getAnAccess()` gets an access to a namespace from inside a type.
- `Namespace.getADefinition()` gets a definition of this namespace. Note that namespaces can have multiple definitions.
- `Namespace.getNamespaceMember(name)` gets an inner namespace with a given name.
- `Namespace.getTypeMember(name)` gets a type exported under a given name.
- `Namespace.getAnExportingContainer()` gets a StmtContainer whose exports contribute to this namespace. This can be a the body of a namespace declaration or the top-level of a module. Enums have no exporting containers.

A LocalNamespaceName behaves like a block-scoped variable, that is, it has an unqualified name and is restricted to a specific scope. The relevant predicates are:

- `LocalNamespaceAccess.getLocalNamespaceName()` gets the local name referenced by an identifier.
- `LocalNamespaceName.getAnAccess()` gets an identifier that refers to this local name.
- `LocalNamespaceName.getADeclaration()` gets an identifier that declares this local name.
- `LocalNamespaceName.getNamespace()` gets the namespace to which this name refers.

7.3.5 Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- QL language reference
- CodeQL tools

7.4 Analyzing data flow in JavaScript and TypeScript

This topic describes how data flow analysis is implemented in the CodeQL libraries for JavaScript/TypeScript and includes examples to help you write your own data flow queries.

7.4.1 Overview

The various sections in this article describe how to utilize the libraries for local data flow, global data flow, and taint tracking. As our running example, we will develop a query that identifies command-line arguments that are passed as a file path to the standard Node.js `readFile` function. While this is not a problematic pattern as such, it is typical of the kind of reasoning that is frequently used in security queries.

For a more general introduction to modeling data flow, see *About data flow analysis*.

7.4.2 Data flow nodes

Both local and global data flow, as well as taint tracking, work on a representation of the program known as the *data flow graph*. Nodes on the data flow flow graph may also correspond to nodes on the abstract syntax tree, but they are not the same. While AST nodes belong to class `ASTNode` and its subclasses, data flow nodes belong to class `DataFlow::Node` and its subclasses:

- `DataFlow::ValueNode`: a *value node*, that is, a data flow node that corresponds either to an expression, or to a declaration of a function, class, TypeScript namespace, or TypeScript enum.
- `DataFlow::SsaDefinitionNode`: a data flow node that corresponds to an SSA variable, that is, a local variable with additional information to reason more precisely about different assignments to the same variable. This kind of data flow node does not correspond to an AST node.
- `DataFlow::PropRef`: a data flow node that corresponds to a read or a write of an object property, for example, in an assignment, in an object literal, or in a destructuring assignment.
- `DataFlow::PropRead`, `DataFlow::PropWrite`: subclasses of `DataFlow::PropRef` that correspond to reads and writes, respectively.

Apart from these fairly general classes, there are some more specialized classes:

- `DataFlow::ParameterNode`: a data flow node that corresponds to a function parameter.
- `DataFlow::InvokeNode`: a data flow node that corresponds to a function call; its subclasses `DataFlow::NewNode` and `DataFlow::CallNode` represent calls with and without `new` respectively, while `DataFlow::MethodCallNode` represents method calls. Note that these classes also model reflective calls using `.call` and `.apply`, which do not correspond to any AST nodes.
- `DataFlow::ThisNode`: a data flow node that corresponds to the value of `this` in a function or top level. This kind of data flow node also does not correspond to an AST node.
- `DataFlow::GlobalVarRefNode`: a data flow node that corresponds to a direct reference to a global variable. This class is rarely used directly, instead you would normally use the predicate `globalVarRef` (introduced below), which also considers indirect references through `window` or global `this`.
- `DataFlow::FunctionNode`, `DataFlow::ObjectLiteralNode`, `DataFlow::ArrayLiteralNode`: a data flow node that corresponds to a function (expression or declaration), an object literal, or an array literal, respectively.
- `DataFlow::ClassNode`: a data flow node corresponding to a class, either defined using an ECMAScript 2015 `class` declaration or an old-style constructor function.
- `DataFlow::ModuleImportNode`: a data flow node corresponding to an ECMAScript 2015 import or an AMD or CommonJS `require` import.

The following predicates are available for mapping from AST nodes and other elements to their corresponding data flow nodes:

- `DataFlow::valueNode(x)`: maps `x`, which must be an expression or a declaration of a function, class, namespace or enum, to its corresponding `DataFlow::ValueNode`.
- `DataFlow::ssaDefinitionNode(ssa)`: maps an SSA definition `ssa` to its corresponding `DataFlow::SsaDefinitionNode`.
- `DataFlow::parameterNode(p)`: maps a function parameter `p` to its corresponding `DataFlow::ParameterNode`.
- `DataFlow::thisNode(s)`: maps a function or top-level `s` to the `DataFlow::ThisNode` representing the value of `this` in `s`.

Class `DataFlow::Node` also has a member predicate `asExpr()` that you can use to map from a `DataFlow::ValueNode` to the expression it corresponds to. Note that this predicate is undefined for other kinds of nodes, and for value nodes that do not correspond to expressions.

There are also some other predicates available for accessing commonly used data flow nodes:

- `DataFlow::globalVarRef(g)`: gets a data flow node corresponding to an access to global variable `g`, either directly or through `window` or (top-level) `this`. For example, you can use `DataFlow::globalVarRef("document")` to find references to the DOM `document` object.
- `DataFlow::moduleMember(p, m)`: gets a data flow node that references a member `m` of a module loaded from path `p`. For example, you can use `DataFlow::moduleMember("fs", "readFile")` to find references to the `fs.readFile` function from the Node.js standard library.

7.4.3 Local data flow

Local data flow is data flow within a single function. Data flow through function calls and returns or through property writes and reads is not modeled.

Local data flow is faster to compute and easier to use than global data flow, but less complete. It is, however, sufficient for many purposes.

To reason about local data flow, use the member predicates `getAPredecessor` and `getASuccessor` on `DataFlow::Node`. For a data flow node `nd`, `nd.getAPredecessor()` returns all data flow nodes from which data flows to `nd` in one local step. Conversely, `nd.getASuccessor()` returns all nodes to which data flows from `nd` in one local step.

To follow one or more steps of local data flow, use the transitive closure operator `+`, and for zero or more steps the reflexive transitive closure operator `*`.

For example, the following query finds all data flow nodes `source` whose value may flow into the first argument of a call to a method with name `readFile`:

```
import javascript

from DataFlow::MethodCallNode readFile, DataFlow::Node source
where
  readFile.getMethodName() = "readFile" and
  source.getASuccessor*() = readFile.getArgument(0)
select source
```

Source nodes

Explicit reasoning about data flow edges can be cumbersome and is rare in practice. Typically, we are not interested in flow originating from arbitrary nodes, but from nodes that in some sense are the source of some kind of data, either because they create a new object, such as object literals or functions, or because they represent a point where data enters the local data flow graph, such as parameters or property reads.

The data flow library represents such nodes by the class `DataFlow::SourceNode`, which provides a convenient API to reason about local data flow involving source nodes.

By default, the following kinds of data flow nodes are considered source nodes:

- classes, functions, object and array literals, regular expressions, and JSX elements
- property reads, global variable references and `this` nodes
- function parameters
- function calls
- imports

You can extend the set of source nodes by defining additional subclasses of `DataFlow::SourceNode::Range`.

The `DataFlow::SourceNode` class defines a number of member predicates that can be used to track where data originating from a source node flows, and to find places where properties are accessed or methods are called on them.

For example, the following query finds all references to properties of `process.argv`, the array through which Node.js applications receive their command-line arguments:

```
import javascript

select DataFlow::globalVarRef("process").getAPropertyRead("argv").getAPropertyReference()
```

First, we use `DataFlow::globalVarRef` (mentioned above) to find all references to the global variable `process`. Since global variable references are source nodes, we can then use the predicate `getAPropertyRead` (defined in class `DataFlow::SourceNode`) to find all places where the property `argv` of that global variable is read. The results of this predicate are again source nodes, so we can chain it with a call to `getAPropertyReference`, which is a predicate that finds all references to any property (even references with a computed name) on its base source node.

Note that many predicates on `DataFlow::SourceNode` have source nodes as their result in turn, allowing calls to be chained to concisely express the relationship between several data flow nodes.

Most importantly, predicates like `getAPropertyRead` implicitly follow local data flow, so the above query not only finds direct property references like `process.argv[2]`, but also more indirect ones as in this example:

```
var args = process.argv;
var firstArg = args[2];
```

Analogous to `getAPropertyRead` there is also a predicate `getAPropertyWrite` for identifying property writes.

Another common task is to find calls to a function originating from a source node. For this purpose, `DataFlow::SourceNode` offers predicates `getACall`, `getAnInstantiation` and `getAnInvocation`: the first one only considers invocations without `new`, the second one only invocations with `new`, and the third one considers all invocations.

We can use these predicates in combination with `DataFlow::moduleMember` (mentioned above) to find calls to the function `readFile` imported from the standard Node.js `fs` library:

```
import javascript

select DataFlow::moduleMember("fs", "readFile").getACall()
```

For identifying method calls there is also a predicate `getAMethodCall`, and the slightly more general `getAMemberCall`. The difference between the two is that the former only finds calls that have the syntactic shape of a method call such as `x.m(...)`, while the latter also finds calls where `x.m` is first stored into a local variable `f` and then invoked as `f(...)`.

Finally, the predicate `flowsTo(nd)` holds for any node `nd` into which data originating from the source node may flow. Conversely, `DataFlow::Node` offers a predicate `getALocalSource()` that can be used to find any source node that flows to it.

Putting all of the above together, here is a query that finds (local) data flow from command line arguments to `readFile` calls:

```
import javascript

from DataFlow::SourceNode arg, DataFlow::CallNode call
where
  arg = DataFlow::globalVarRef("process").getAPropertyRead("argv").getAPropertyReference() and
  call = DataFlow::moduleMember("fs", "readFile").getACall() and
  arg.flowsTo(call.getArgument(0))
select arg, call
```

There are two points worth making about the source node API:

1. All data flow tracking is purely local, and in particular flow through global variables is not tracked. If `args` in our `process.argv` example above is a global variable, then the query will not find the reference through `args[2]`.
2. Strings are not source nodes and cannot be tracked using this API. You can, however, use the `mayHaveStringValue` predicate on class `DataFlow::Node` to reason about the possible string values flowing into a data flow node.

For a full description of the `DataFlow::SourceNode` API, see the JavaScript standard library.

Exercises

Exercise 1: Write a query that finds all hard-coded strings used as the `tagName` argument to the `createElement` function from the DOM `document` object, using local data flow. (*Answer*).

7.4.4 Global data flow

Global data flow tracks data flow throughout the entire program, and is therefore more powerful than local data flow. However, global data flow is less precise than local data flow. That is, the analysis may report spurious flows that cannot in fact happen. Moreover, global data flow analysis typically requires significantly more time and memory than local analysis.

Note

You can model data flow paths in CodeQL by creating path queries. To view data flow paths generated by a path query in CodeQL for VS Code, you need to make sure that it has the correct metadata and `select` clause. For more information, see Creating path queries.

Using global data flow

For performance reasons, it is not generally feasible to compute all global data flow across the entire program. Instead, you can define a data flow *configuration*, which specifies *source* data flow nodes and *sink* data flow nodes (sources and sinks for short) of interest. The data flow library provides a generic data flow solver that can check whether there is (global) data flow from a source to a sink.

Optionally, configurations may specify extra data flow edges to be added to the data flow graph, and may also specify *barriers*. Barriers are data flow nodes or edges through which data should not be tracked for the purposes of this analysis.

To define a configuration, extend the class `DataFlow::Configuration` as follows:

```
class MyDataFlowConfiguration extends DataFlow::Configuration {
  MyDataFlowConfiguration() { this = "MyDataFlowConfiguration" }

  override predicate isSource(DataFlow::Node source) { /* ... */ }

  override predicate isSink(DataFlow::Node sink) { /* ... */ }

  // optional overrides:
  override predicate isBarrier(DataFlow::Node nd) { /* ... */ }
  override predicate isBarrierEdge(DataFlow::Node pred, DataFlow::Node succ) { /* ... */ }
  override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ) { /* ... */ }
}
```

The characteristic predicate `MyDataFlowConfiguration()` defines the name of the configuration, so `"MyDataFlowConfiguration"` should be replaced by a suitable name describing your particular analysis configuration.

The data flow analysis is performed using the predicate `hasFlow(source, sink)`:

```
from MyDataFlowConfiguration dataflow, DataFlow::Node source, DataFlow::Node sink
where dataflow.hasFlow(source, sink)
select source, "Data flow from $@ to $@.", source, source.toString(), sink, sink.toString()
```

Using global taint tracking

Global taint tracking extends global data flow with additional non-value-preserving steps, such as flow through string-manipulating operations. To use it, simply extend `TaintTracking::Configuration` instead of `DataFlow::Configuration`:

```
class MyTaintTrackingConfiguration extends TaintTracking::Configuration {
  MyTaintTrackingConfiguration() { this = "MyTaintTrackingConfiguration" }

  override predicate isSource(DataFlow::Node source) { /* ... */ }
```

(continues on next page)

(continued from previous page)

```
  override predicate isSink(DataFlow::Node sink) { /* ... */ }
}
```

Analogous to `isAdditionalFlowStep`, there is a predicate `isAdditionalTaintStep` that you can override to specify custom flow steps to consider in the analysis. Instead of the `isBarrier` and `isBarrierEdge` predicates, the taint tracking configuration includes `isSanitizer` and `isSanitizerEdge` predicates that specify data flow nodes or edges that act as taint sanitizers and hence stop flow from a source to a sink.

Similar to global data flow, the characteristic predicate `MyTaintTrackingConfiguration()` defines the unique name of the configuration, so `"MyTaintTrackingConfiguration"` should be replaced by an appropriate descriptive name.

The taint tracking analysis is again performed using the predicate `hasFlow(source, sink)`.

Examples

The following taint-tracking configuration is a generalization of our example query above, which tracks flow from command-line arguments to `readFile` calls, this time using global taint tracking.

```
import javascript

class CommandLineFileNameConfiguration extends TaintTracking::Configuration {
  CommandLineFileNameConfiguration() { this = "CommandLineFileNameConfiguration" }

  override predicate isSource(DataFlow::Node source) {
    DataFlow::globalVarRef("process").getAPropertyRead("argv").getAPropertyRead() = source
  }

  override predicate isSink(DataFlow::Node sink) {
    DataFlow::moduleMember("fs", "readFile").getACall().getArgument(0) = sink
  }
}

from CommandLineFileNameConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

This query will now find flows that involve inter-procedural steps, like in the following example (where the individual steps have been marked with comments `#1` to `#4`):

```
const fs = require('fs'),
      path = require('path');

function readFileHelper(p) {     // #2
  p = path.resolve(p);           // #3
  fs.readFile(p,                 // #4
    'utf8', (err, data) => {
    if (err) throw err;
    console.log(data);
  });
}
```

(continues on next page)

(continued from previous page)

```
readFileHelper(process.argv[2]); // #1
```

Note that for step #3 we rely on the taint-tracking librarys built-in model of the Node.js `path` library, which adds a taint step from `p` to `path.resolve(p)`. This step is not value preserving, but it preserves taint in the sense that if `p` is user-controlled, then so is `path.resolve(p)` (at least partially).

Other standard taint steps include flow through string-manipulating operations such as concatenation, `JSON.parse` and `JSON.stringify`, array transformations, promise operations, and many more.

Sanitizers

The above JavaScript program allows the user to read any file, including sensitive system files like `/etc/passwd`. If the program may be invoked by an untrusted user, this is undesirable, so we may want to constrain the path. For example, instead of using `path.resolve` we could implement a function `checkPath` that first makes the path absolute and then checks that it starts with the current working directory, aborting the program with an error if it does not. We could then use that function in `readFileHelper` like this:

```
function readFileHelper(p) {
  p = checkPath(p);
  ...
}
```

For the purposes of our above analysis, `checkPath` is a *sanitizer*: its output is always untainted, even if its input is tainted. To model this we can add an override of `isSanitizer` to our taint-tracking configuration like this:

```
class CommandLineFileNameConfiguration extends TaintTracking::Configuration {

  // ...

  override predicate isSanitizer(DataFlow::Node nd) {
    nd.(DataFlow::CallNode).getCalleeName() = "checkPath"
  }
}
```

This says that any call to a function named `checkPath` is to be considered a sanitizer, so any flow through this node is blocked. In particular, the query would no longer flag the flow from `process.argv[2]` to `fs.readFile` in our updated example above.

Sanitizer guards

A perhaps more natural way of implementing the path check in our example would be to have `checkPath` return a Boolean value indicating whether the path is safe to read (instead of returning the path if it is safe and aborting otherwise). We could then use it in `readFileHelper` like this:

```
function readFileHelper(p) {
  if (!checkPath(p))
    return;
  ...
}
```

Note that `checkPath` is now no longer a sanitizer in the sense described above, since the flow from `process.argv[2]` to `fs.readFile` does not go through `checkPath` any more. The flow is, however, *guarded* by `checkPath` in the sense that the expression `checkPath(p)` has to evaluate to `true` (or, more precisely, to a truthy value) in order for the flow to happen.

Such sanitizer guards can be supported by defining a new subclass of `TaintTracking::SanitizerGuardNode` and overriding the predicate `isSanitizerGuard` in the taint-tracking configuration class to add all instances of this class as sanitizer guards to the configuration.

For our above example, we would begin by defining a subclass of `SanitizerGuardNode` that identifies guards of the form `checkPath(...)`:

```
class CheckPathSanitizerGuard extends TaintTracking::SanitizerGuardNode, DataFlow::CallNode {
  CheckPathSanitizerGuard() { this.getCalleeName() = "checkPath" }

  override predicate sanitizes(boolean outcome, Expr e) {
    outcome = true and
    e = getArgument(0).asExpr()
  }
}
```

The characteristic predicate of this class checks that the sanitizer guard is a call to a function named `checkPath`. The overriding definition of `sanitizes` says such a call sanitizes its first argument (that is, `getArgument(0)`) if it evaluates to `true` (or rather, a truthy value).

Now we can override `isSanitizerGuard` to add these sanitizer guards to our configuration:

```
class CommandLineFileNameConfiguration extends TaintTracking::Configuration {

  // ...

  override predicate isSanitizerGuard(TaintTracking::SanitizerGuardNode nd) {
    nd instanceof CheckPathSanitizerGuard
  }
}
```

With these two additions, the query recognizes the `checkPath(p)` check as sanitizing `p` after the `return`, since execution can only reach there if `checkPath(p)` evaluates to a truthy value. Consequently, there is no longer a path from `process.argv[2]` to `readFile`.

Additional taint steps

Sometimes the default data flow and taint steps provided by `DataFlow::Configuration` and `TaintTracking::Configuration` are not sufficient and we need to add additional flow or taint steps to our configuration to make it find the expected flow. For example, this can happen because the analyzed program uses a function from an external library whose source code is not available to the analysis, or because it uses a function that is too difficult to analyze.

In the context of our running example, assume that the JavaScript program we are analyzing uses a (fictitious) npm package `resolve-symlinks` to resolve any symlinks in the path `p` before passing it to `readFile`:

```
const resolveSymlinks = require('resolve-symlinks');

function readFileHelper(p) {
  p = resolveSymlinks(p);
  fs.readFile(p,
  ...
}
```

Resolving symlinks does not make an unsafe path any safer, so we would still like our query to flag this, but since the standard library does not have a model of `resolve-symlinks` it will no longer return any results.

We can fix this quite easily by adding an overriding definition of the `isAdditionalTaintStep` predicate to our configuration, introducing an additional taint step from the first argument of `resolveSymlinks` to its result:

```
class CommandLineFileNameConfiguration extends TaintTracking::Configuration {

  // ...

  override predicate isAdditionalTaintStep(DataFlow::Node pred, DataFlow::Node succ) {
    exists(DataFlow::CallNode c |
      c = DataFlow::moduleImport("resolve-symlinks").getACall() and
      pred = c.getArgument(0) and
      succ = c
    )
  }
}
```

We might even consider adding this as a default taint step to be used by all taint-tracking configurations. In order to do this, we need to wrap it in a new subclass of `TaintTracking::AdditionalTaintStep` like this:

```
class StepThroughResolveSymlinks extends TaintTracking::AdditionalTaintStep, DataFlow::CallNode {
  StepThroughResolveSymlinks() { this = DataFlow::moduleImport("resolve-symlinks").getACall() }

  override predicate step(DataFlow::Node pred, DataFlow::Node succ) {
    pred = this.getArgument(0) and
    succ = this
  }
}
```

If we add this definition to the standard library, it will be picked up by all taint-tracking configurations. Obviously, one has to be careful when adding such new additional taint steps to ensure that they really make sense for *all* configurations.

Analogous to `TaintTracking::AdditionalTaintStep`, there is also a class `DataFlow::AdditionalFlowStep` that can be extended to add extra steps to all data-flow configurations, and hence also to all taint-tracking configurations.

Exercises

Exercise 2: Write a query that finds all hard-coded strings used as the `tagName` argument to the `createElement` function from the DOM `document` object, using global data flow. (*Answer*).

Exercise 3: Write a class which represents flow sources from the array elements of the result of a call, for example the expression `myObject.myMethod(myArgument)[myIndex]`. Hint: array indices are properties with numeric names; you can use regular expression matching to check this. (*Answer*)

Exercise 4: Using the answers from 2 and 3, write a query which finds all global data flows from array elements of the result of a call to the `tagName` argument to the `createElement` function. (*Answer*)

7.4.5 Answers

Exercise 1

```
import javascript

from DataFlow::CallNode create, string name
where
  create = DataFlow::globalVarRef("document").getAMethodCall("createElement") and
  create.getArgument(0).mayHaveStringValue(name)
select name
```

Exercise 2

```
import javascript

class HardCodedTagNameConfiguration extends DataFlow::Configuration {
  HardCodedTagNameConfiguration() { this = "HardCodedTagNameConfiguration" }

  override predicate isSource(DataFlow::Node source) { source.asExpr() instanceof ConstantString }

  override predicate isSink(DataFlow::Node sink) {
    sink = DataFlow::globalVarRef("document").getAMethodCall("createElement").getArgument(0)
  }
}

from HardCodedTagNameConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

Exercise 3

```
import javascript

class ArrayEntryCallResult extends DataFlow::Node {
  ArrayEntryCallResult() {
    exists(DataFlow::CallNode call, string index |
      this = call.getAPropertyRead(index) and
      index.regexpMatch("\\d+")
    )
  }
}
```

Exercise 4

```
import javascript

class ArrayEntryCallResult extends DataFlow::Node {
  ArrayEntryCallResult() {
    exists(DataFlow::CallNode call, string index |
      this = call.getAPropertyRead(index) and
      index.regexpMatch("\\d+")
    )
  }
}

class HardCodedTagNameConfiguration extends DataFlow::Configuration {
  HardCodedTagNameConfiguration() { this = "HardCodedTagNameConfiguration" }

  override predicate isSource(DataFlow::Node source) { source instanceof ArrayEntryCallResult }

  override predicate isSink(DataFlow::Node sink) {
    sink = DataFlow::globalVarRef("document").getAMethodCall("createElement").getArgument(0)
  }
}

from HardCodedTagNameConfiguration cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select source, sink
```

7.4.6 Further reading

- Exploring data flow with path queries
- CodeQL queries for Java
- Example queries for Java
- CodeQL library reference for Java
- QL language reference
- CodeQL tools

Contents

– *Further reading*

7.5 Using flow labels for precise data flow analysis

You can associate flow labels with each value tracked by the flow analysis to determine whether the flow contains potential vulnerabilities.

7.5.1 Overview

You can use basic inter-procedural data-flow analysis and taint tracking as described in *Analyzing data flow in JavaScript and TypeScript* to check whether there is a path in the data-flow graph from some source node to a sink node that does not pass through any sanitizer nodes. Another way of thinking about this is that it statically models the flow of data through the program, and associates a flag with every data value telling us whether it might have come from a source node.

In some cases, you may want to track more detailed information about data values. This can be done by associating flow labels with data values, as shown in this tutorial. We will first discuss the general idea behind flow labels and then show how to use them in practice. Finally, we will give an overview of the API involved and provide some pointers to standard queries that use flow labels.

7.5.2 Limitations of basic data-flow analysis

In many applications we are interested in tracking more than just the reachability information provided by inter-procedural data flow analysis.

For example, when tracking object values that originate from untrusted input, we might want to remember whether the entire object is tainted or whether only part of it is tainted. The former happens, for example, when parsing a user-controlled string as JSON, meaning that the entire resulting object is tainted. A typical example of the latter is assigning a tainted value to a property of an object, which only taints that property but not the rest of the object.

While reading a property of a completely tainted object yields a tainted value, reading a property of a partially tainted object does not. On the other hand, JSON-encoding even a partially tainted object and including it in an HTML document is not safe.

Another example where more fine-grained information about tainted values is needed is for tracking partial sanitization. For example, before interpreting a user-controlled string as a file-system path, we generally want to make sure that it is neither an absolute path (which could refer to any file on the file system) nor a relative path containing `..` components (which still could refer to any file). Usually, checking both of these properties would involve two separate checks. Both checks taken together should count as a sanitizer, but each individual check is not by itself enough to make the string safe for use as a path. To handle this case precisely, we want to associate two bits of information with each tainted value, namely whether it may be absolute, and whether it may contain `..` components. Untrusted user input has both bits set initially, individual checks turn off individual bits, and if a value that has at least one bit set is interpreted as a path, a potential vulnerability is flagged.

7.5.3 Using flow labels

You can handle these cases and others like them by associating a set of *flow labels* (sometimes also referred to as *taint kinds*) with each value being tracked by the analysis. Value-preserving data-flow steps (such as flow steps from writes to a variable to its reads) preserve the set of flow labels, but other steps may add or remove flow

labels. Sanitizers, in particular, are simply flow steps that remove some or all flow labels. The initial set of flow labels for a value is determined by the source node that gives rise to it. Similarly, sink nodes can specify that an incoming value needs to have a certain flow label (or one of a set of flow labels) in order for the flow to be flagged as a potential vulnerability.

7.5.4 Example

As an example of using flow labels, we will show how to write a query that flags property accesses on JSON values that come from user-controlled input where we have not checked whether the value is `null`, so that the property access may cause a runtime exception.

For example, we would like to flag this code:

```
var data = JSON.parse(str);
if (data.length > 0) {  // problematic: `data` may be `null`
  ...
}
```

This code, on the other hand, should not be flagged:

```
var data = JSON.parse(str);
if (data && data.length > 0) { // unproblematic: `data` is first checked for nullness
  ...
}
```

We will first try to write a query to find this kind of problem without flow labels, and use the difficulties we encounter as a motivation for bringing flow labels into play, which will make the query much easier to implement.

To get started, lets write a query that simply flags any flow from `JSON.parse` into the base of a property access:

```
import javascript

class JsonTrackingConfig extends DataFlow::Configuration {
  JsonTrackingConfig() { this = "JsonTrackingConfig" }

  override predicate isSource(DataFlow::Node nd) {
    exists(JsonParserCall jpc |
      nd = jpc.getOutput()
    )
  }

  override predicate isSink(DataFlow::Node nd) {
    exists(DataFlow::PropRef pr |
      nd = pr.getBase()
    )
  }
}

from JsonTrackingConfig cfg, DataFlow::Node source, DataFlow::Node sink
where cfg.hasFlow(source, sink)
select sink, "Property access on JSON value originating $@.", source, "here"
```

Note that we use the `JsonParserCall` class from the standard library to model various JSON parsers, including the standard `JSON.parse` API as well as a number of popular npm packages.

Of course, as written this query flags both the good and the bad example above, since we have not introduced any sanitizers yet.

There are many ways of checking for nullness directly or indirectly. Since this is not the main focus of this tutorial, we will only show how to model one specific case: if some variable `v` is known to be truthy, it cannot be `null`. This kind of condition is easily expressed using a `BarrierGuardNode` (or its counterpart `SanitizerGuardNode` for taint-tracking configurations). A barrier guard node is a data-flow node `b` that blocks flow through some other node `nd`, provided that some condition checked at `b` is known to hold, that is, evaluate to a truthy value.

In our case, the barrier guard node is a use of some variable `v`, and the condition is that use itself: it blocks flow through any use of `v` where the guarding use is known to evaluate to a truthy value. In our second example above, the use of `data` on the left-hand side of the `&&` is a barrier guard blocking flow through the use of `data` on the right-hand side of the `&&`. At this point we know that `data` has evaluated to a truthy value, so it cannot be `null` anymore.

Implementing this additional condition is easy. We implement a subclass of `DataFlow::BarrierGuardNode`:

```
class TruthinessCheck extends DataFlow::BarrierGuardNode, DataFlow::ValueNode {
  SsaVariable v;

  TruthinessCheck() {
    astNode = v.getAUse()
  }

  override predicate blocks(boolean outcome, Expr e) {
    outcome = true and
    e = astNode
  }
}
```

and then use it to override predicate `isBarrierGuard` in our configuration class:

```
override predicate isBarrierGuard(DataFlow::BarrierGuardNode guard) {
  guard instanceof TruthinessCheck
}
```

With this change, we now flag the problematic case and dont flag the unproblematic case above.

However, as it stands our analysis has many false negatives: if we read a property of a JSON object, our analysis will not continue tracking it, so property accesses on the resulting value will not be checked for null-guardedness:

```
var root = JSON.parse(str);
if (root) {
  var payload = root.data;   // unproblematic: `root` cannot be `null` here
  if (payload.length > 0) {  // problematic: `payload` may be `null` here
    ...
  }
}
```

We could try to remedy the situation by overriding `isAdditionalFlowStep` in our configuration class to track values through property reads:

```
override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ) {
  succ.(DataFlow::PropRead).getBase() = pred
}
```

But this does not actually allow us to flag the problem above as once we have checked `root` for truthiness, all further uses are considered to be sanitized. In particular, the reference to `root` in `root.data` is sanitized, so no flow tracking through the property read happens.

The problem is, of course, that our sanitizer sanitizes too much. It should not stop flow altogether, it should simply record the fact that `root` itself is known to be non-null. Any property read from `root`, on the other hand, may well be null and needs to be checked separately.

We can achieve this by introducing two different flow labels, `json` and `maybe-null`. The former means that the value we are dealing with comes from a JSON object, the latter that it may be `null`. The result of any call to `JSON.parse` has both labels. A property read from a value with label `json` also has both labels. Checking truthiness removes the `maybe-null` label. Accessing a property on a value that has the `maybe-null` label should be flagged.

To implement this, we start by defining two new subclasses of the class `DataFlow::FlowLabel`:

```
class JsonLabel extends DataFlow::FlowLabel {
  JsonLabel() {
    this = "json"
  }
}

class MaybeNullLabel extends DataFlow::FlowLabel {
  MaybeNullLabel() {
    this = "maybe-null"
  }
}
```

Then we extend our `isSource` predicate from above to track flow labels by overriding the two-argument version instead of the one-argument version:

```
override predicate isSource(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
  exists(JsonParserCall jpc |
    nd = jpc.getOutput() and
    (lbl instanceof JsonLabel or lbl instanceof MaybeNullLabel)
  )
}
```

Similarly, we make `isSink` flow-label aware and require the base of the property read to have the `maybe-null` label:

```
override predicate isSink(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
  exists(DataFlow::PropRef pr |
    nd = pr.getBase() and
    lbl instanceof MaybeNullLabel
  )
}
```

Our overriding definition of `isAdditionalFlowStep` now needs to specify two flow labels, a predecessor label `predlbl` and a successor label `succlbl`. In addition to specifying flow from the predecessor node `pred` to the successor node `succ`, it requires that `pred` has label `predlbl`, and adds label `succlbl` to `succ`. In our case, we use this to add both the `json` label and the `maybe-null` label to any property read from a value labeled with `json` (no matter whether it has the `maybe-null` label):

```
override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ,
                                        DataFlow::FlowLabel predlbl, DataFlow::FlowLabel succlbl) {
  succ.(DataFlow::PropRead).getBase() = pred and
  predlbl instanceof JsonLabel and
  (succlbl instanceof JsonLabel or succlbl instanceof MaybeNullLabel)
}
```

Finally, we turn `TruthinessCheck` from a `BarrierGuardNode` into a `LabeledBarrierGuardNode`, specifying that it only removes the `maybe-null` label (but not the `json` label) from the sanitized value:

```
class TruthinessCheck extends DataFlow::LabeledBarrierGuardNode, DataFlow::ValueNode {
  ...

  override predicate blocks(boolean outcome, Expr e, DataFlow::FlowLabel lbl) {
    outcome = true and
    e = astNode and
    lbl instanceof MaybeNullLabel
  }
}
```

Here is the final query, expressed as a *path query* so we can examine paths from sources to sinks step by step in the UI:

```
/** @kind path-problem */

import javascript
import DataFlow::PathGraph

class JsonLabel extends DataFlow::FlowLabel {
  JsonLabel() {
    this = "json"
  }
}

class MaybeNullLabel extends DataFlow::FlowLabel {
  MaybeNullLabel() {
    this = "maybe-null"
  }
}

class TruthinessCheck extends DataFlow::LabeledBarrierGuardNode, DataFlow::ValueNode {
  SsaVariable v;

  TruthinessCheck() {
    astNode = v.getAUse()
  }
```

(continues on next page)

(continued from previous page)

```
  override predicate blocks(boolean outcome, Expr e, DataFlow::FlowLabel lbl) {
    outcome = true and
    e = astNode and
    lbl instanceof MaybeNullLabel
  }
}

class JsonTrackingConfig extends DataFlow::Configuration {
  JsonTrackingConfig() { this = "JsonTrackingConfig" }

  override predicate isSource(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
    exists(JsonParserCall jpc |
      nd = jpc.getOutput() and
      (lbl instanceof JsonLabel or lbl instanceof MaybeNullLabel)
    )
  }

  override predicate isSink(DataFlow::Node nd, DataFlow::FlowLabel lbl) {
    exists(DataFlow::PropRef pr |
      nd = pr.getBase() and
      lbl instanceof MaybeNullLabel
    )
  }

  override predicate isAdditionalFlowStep(DataFlow::Node pred, DataFlow::Node succ,
                              DataFlow::FlowLabel predlbl, DataFlow::FlowLabel succlbl) {
    succ.(DataFlow::PropRead).getBase() = pred and
    predlbl instanceof JsonLabel and
    (succlbl instanceof JsonLabel or succlbl instanceof MaybeNullLabel)
  }

  override predicate isBarrierGuard(DataFlow::BarrierGuardNode guard) {
    guard instanceof TruthinessCheck
  }
}

from JsonTrackingConfig cfg, DataFlow::PathNode source, DataFlow::PathNode sink
where cfg.hasFlowPath(source, sink)
select sink, source, sink, "Property access on JSON value originating $@.", source, "here"
```

Here is a run of this query on the plexus-interop project on LGTM.com. Many of the 19 results are false positives since we currently do not model many ways in which a value can be checked for nullness. In particular, after a property reference `x.p` we implicitly know that `x` cannot be null anymore, since otherwise the reference would have thrown an exception. Modeling this would allow us to get rid of most of the false positives, but is beyond the scope of this tutorial.

7.5.5 API

Plain data-flow configurations implicitly use a single flow label data, which indicates that a data value originated from a source. You can use the predicate `DataFlow::FlowLabel::data()`, which returns this flow label, as a

symbolic name for it.

Taint-tracking configurations add a second flow label taint (`DataFlow::FlowLabel::taint()`), which is similar to data, but includes values that have passed through non-value preserving steps such as string operations.

Each of the three member predicates `isSource`, `isSink` and `isAdditionalFlowStep`/`isAdditionalTaintStep` has one version that uses the default flow labels, and one version that allows specifying custom flow labels through additional arguments.

For `isSource`, there is one additional argument specifying which flow label(s) should be associated with values originating from this source. If multiple flow labels are specified, each value is associated with *all* of them.

For `isSink`, the additional argument specifies which flow label(s) a value that flows into this source may be associated with. If multiple flow labels are specified, then any value that is associated with *at least one* of them will be considered by the configuration.

For `isAdditionalFlowStep` there are two additional arguments `predlbl` and `succlbl`, which allow flow steps to act as flow label transformers. If a value associated with `predlbl` arrives at the start node of the additional step, it is propagated to the end node and associated with `succlbl`. Of course, `predlbl` and `succlbl` may be the same, indicating that the flow step preserves this label. There can also be multiple values of `succlbl` for a single `predlbl` or vice versa.

Note that if you do not restrict `succlbl` then it will be allowed to range over all flow labels. This may cause labels that were previously blocked on a path to reappear, which is not usually what you want.

The flow label-aware version of `isBarrier` is called `isLabeledBarrier`: unlike `isBarrier`, which prevents any flow past the given node, it only blocks flow of values associated with one of the specified flow labels.

7.5.6 Standard queries using flow labels

Some of our standard security queries use flow labels. You can look at their implementation to get a feeling for how to use flow labels in practice.

In particular, both of the examples mentioned in the section on limitations of basic data flow above are from standard security queries that use flow labels. The Prototype pollution query uses two flow labels to distinguish completely tainted objects from partially tainted objects. The Uncontrolled data used in path expression query uses four flow labels to track whether a user-controlled string may be an absolute path and whether it may contain `..` components.

7.5.7 Further reading

- Exploring data flow with path queries
- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- QL language reference
- CodeQL tools

7.6 Using type tracking for API modeling

You can track data through an API by creating a model using the CodeQL type-tracking library for JavaScript.

7.6.1 Overview

The type-tracking library makes it possible to track values through properties and function calls, usually to recognize method calls and properties accessed on a specific type of object.

This is an advanced topic and is intended for readers already familiar with the SourceNode class as well as taint tracking. For TypeScript analysis also consider reading about static type information first.

7.6.2 The problem of recognizing method calls

Well start with a simple model of the Firebase API and gradually build on it to use type tracking. Knowledge of Firebase is not required.

Suppose we wish to find places where data is written to a Firebase database, as in the following example:

```
var ref = firebase.database().ref("forecast");
ref.set("Rain"); // <-- find this call
```

A simple way to do this is just to find all method calls named `set`:

```
import javascript
import DataFlow

MethodCallNode firebaseSetterCall() {
  result.getMethodName() = "set"
}
```

The obvious problem with this is that it finds calls to *all* methods named `set`, many of which are unrelated to Firebase.

Another approach is to use local data flow to match the chain of calls that led to this call:

```
MethodCallNode firebaseSetterCall() {
  result = globalVarRef("firebase")
      .getAMethodCall("database")
      .getAMethodCall("ref")
      .getAMethodCall("set")
}
```

This will find the `set` call from the example, but no spurious, unrelated `set` method calls. We can split it up so each step is its own predicate:

```
SourceNode firebase() {
  result = globalVarRef("firebase")
}

SourceNode firebaseDatabase() {
  result = firebase().getAMethodCall("database")
}

SourceNode firebaseRef() {
  result = firebaseDatabase().getAMethodCall("ref");
}
```

(continues on next page)

(continued from previous page)

```
MethodCallNode firebaseSetterCall() {
  result = firebaseRef().getAMethodCall("set")
}
```

The code above is equivalent to the previous version, but its easier to tinker with the individual steps.

The downside is that the model relies entirely on local data flow, which means it wont look through properties and function calls. For instance, `firebaseSetterCall()` fails to find anything in this example:

```
function getDatabase() {
  return firebase.database();
}
var ref = getDatabase().ref("forecast");
ref.set("Rain");
```

Notice that the predicate `firebaseDatabase()` still finds the call to `firebase.database()`, but not the `getDatabase()` call. This means `firebaseRef()` has no result, which in turn means `firebaseSetterCall()` has no result.

As a simple remedy, lets try to make `firebaseDatabase()` recognize the `getDatabase()` call:

```
SourceNode firebaseDatabase() {
  result = firebase().getAMethodCall("database")
  or
  result.(CallNode).getACallee().getAReturn().getALocalSource() = firebaseDatabase()
}
```

The second clause ensures `firebaseDatabase()` finds not only `firebase.database()` calls, but also calls to functions that *return* `firebase.database()`, such as `getDatabase()` seen above. Its recursive, so it handles flow out of any number of nested function calls.

However, it still only tracks *out* of functions, not *into* functions through parameters, nor through properties. Instead of adding these steps by hand, well use type tracking.

7.6.3 Type tracking in general

Type tracking is a generalization of the above pattern, where a predicate matches the value to track, and has a recursive clause that tracks the flow of that value. But instead of us having to deal with function calls/returns and property reads/writes, all of these steps are included in a single predicate, SourceNode.track, to be used with the companion class TypeTracker.

Predicates that use type tracking usually conform to the following general pattern, which we explain below:

```
SourceNode myType(TypeTracker t) {
  t.start() and
  result = /* SourceNode to track */
  or
  exists(TypeTracker t2 |
    result = myType(t2).track(t2, t)
  )
```

(continues on next page)

(continued from previous page)

```
}

SourceNode myType() {
  result = myType(TypeTracker::end())
}
```

Well apply the pattern to our example model and use that to explain whats going on.

7.6.4 Tracking the database instance

Applying the above pattern to the `firebaseDatabase()` predicate we get the following:

```
SourceNode firebaseDatabase(TypeTracker t) {
  t.start() and
  result = firebase().getAMethodCall("database")
  or
  exists(TypeTracker t2 |
    result = firebaseDatabase(t2).track(t2, t)
  )
}

SourceNode firebaseDatabase() {
  result = firebaseDatabase(TypeTracker::end())
}
```

There are now two predicates named `firebaseDatabase`. The one with the `TypeTracker` parameter is the one actually doing the global data flow tracking – the other predicate exposes the result in a convenient way.

The new `TypeTracker t` parameter is a summary of the steps needed to track the value of interest to the resulting data flow node.

In the base case, when matching `firebase.database()`, we use `t.start()` to indicate that no steps were needed, that is, this is the starting point of type tracking:

```
t.start() and
result = firebase().getAMethodCall("database")
```

In the recursive case, we apply the `track` predicate on a previously-found Firebase database node, such as `firebase.database()`. The `track` predicate maps this to a successor of that node, such as `getDatabase()`, and binds `t` to the continuation of `t2` with this extra step included:

```
exists(TypeTracker t2 |
  result = firebaseDatabase(t2).track(t2, t)
)
```

To understand the role of `t` here, note that type tracking can step *into* a property, which means the data flow node returned from `track` is not necessarily a Firebase database instance, it could be an object *containing* a Firebase database in one of its properties.

For example, in the program below, the `firebaseDatabase(t)` predicate includes the `obj` node in its result, but with `t` recording the fact that the actual value being tracked is inside the `DB` property:

```
let obj = { DB: firebase.database() };
let db = obj.DB;
```

This brings us to the last predicate. This uses `TypeTracker::end()` to filter out the paths where the Firebase database instance ended up inside a property of another object, so it includes `db` but not `obj`:

```
SourceNode firebaseDatabase() {
  result = firebaseDatabase(TypeTracker::end())
}
```

Heres see an example of what this can handle now:

```
class Firebase {
  constructor() {
    this.db = firebase.database();
  }

  getDatabase() { return this.db; }

  setForecast(value) {
    this.getDatabase().ref("forecast").set(value); // found by firebaseSetterCall()
  }
}
```

7.6.5 Tracking in the whole model

We applied this pattern to `firebaseDatabase()` in the previous section, and it can just as easily apply to the other predicates. For reference, heres our simple Firebase model with type tracking on every predicate:

```
SourceNode firebase(TypeTracker t) {
  t.start() and
  result = globalVarRef("firebase")
  or
  exists(TypeTracker t2 |
    result = firebase(t2).track(t2, t)
  )
}

SourceNode firebase() {
  result = firebase(TypeTracker::end())
}

SourceNode firebaseDatabase(TypeTracker t) {
  t.start() and
  result = firebase().getAMethodCall("database")
  or
  exists(TypeTracker t2 |
    result = firebaseDatabase(t2).track(t2, t)
  )
}
```

(continues on next page)

(continued from previous page)

```
SourceNode firebaseDatabase() {
  result = firebaseDatabase(TypeTracker::end())
}

SourceNode firebaseRef(TypeTracker t) {
  t.start() and
  result = firebaseDatabase().getAMethodCall("ref")
  or
  exists(TypeTracker t2 |
    result = firebaseRef(t2).track(t2, t)
  )
}

SourceNode firebaseRef() {
  result = firebaseRef(TypeTracker::end())
}

MethodCallNode firebaseSetterCall() {
  result = firebaseRef().getAMethodCall("set")
}
```

Here is a run of an example query using the model to find *set* calls on one of the Firebase sample projects. Its been modified slightly to handle a bit more of the API, which is beyond the scope of this tutorial.

7.6.6 Tracking associated data

By adding extra parameters to the type-tracking predicate, we can carry along extra bits of information about the result.

For example, heres a type-tracking version of `firebaseRef()`, which tracks the string that was passed to the `ref` call:

```
SourceNode firebaseRef(string name, TypeTracker t) {
  t.start() and
  exists(CallNode call |
    call = firebaseDatabase().getAMethodCall("ref") and
    name = call.getArgument(0).getStringValue() and
    result = call
  )
  or
  exists(TypeTracker t2 |
    result = firebaseRef(name, t2).track(t2, t)
  )
}

SourceNode firebaseRef(string name) {
  result = firebaseRef(name, TypeTracker::end())
}

MethodCallNode firebaseSetterCall(string refName) {
  result = firebaseRef(refName).getAMethodCall("set")
```

(continues on next page)

(continued from previous page)

```
}
```

So now we can use `firebaseSetterCall("forecast")` to find assignments to the forecast.

7.6.7 Back-tracking callbacks

The type-tracking predicates weve seen above all use *forward* tracking. That is, they all start with some value of interest and ask where does this flow?.

Sometimes its more useful to work backwards, starting at the desired end-point and asking what flows to here?.

As a motivating example, well extend our model to look for places where we *read* a value from the database, as opposed to writing it. Reading is an asynchronous operation and the result is obtained through a callback, for example:

```
function fetchForecast(callback) {
  firebase.database().ref("forecast").once("value", callback);
}

function updateReminders() {
  fetchForecast((snapshot) => {
    let forecast = snapshot.val(); // <-- find this call
    addReminder(forecast === "Rain" ? "Umbrella" : "Sunscreen");
  })
}
```

The actual forecast is obtained by the call to `snapshot.val()`.

Looking for all method calls named `val` will in practice find many unrelated methods, so well use type tracking again to take the receiver type into account.

The receiver `snapshot` is a parameter to a callback function, which ultimately escapes into the `once()` call. Well extend our model from above to use back-tracking to find all functions that flow into the `once()` call. Backwards type tracking is not too different from forwards type tracking. The differences are:

- The `TypeTracker` parameter instead has type `TypeBackTracker`.
- The call to `.track()` is instead a call to `.backtrack()`.
- To ensure the initial value is a source node, a call to `getALocalSource()` is usually required.

```
SourceNode firebaseSnapshotCallback(string refName, TypeBackTracker t) {
  t.start() and
  result = firebaseRef(refName).getAMethodCall("once").getArgument(1).getALocalSource()
  or
  exists(TypeBackTracker t2 |
    result = firebaseSnapshotCallback(refName, t2).backtrack(t2, t)
  )
}

FunctionNode firebaseSnapshotCallback(string refName) {
  result = firebaseSnapshotCallback(refName, TypeBackTracker::end())
}
```

Now, `firebaseSnapshotCallback("forecast")` finds the function being passed to `fetchForecast`. Based on that we can track the `snapshot` value and find the `val()` call itself:

```
SourceNode firebaseSnapshot(string refName, TypeTracker t) {
  t.start() and
  result = firebaseSnapshotCallback(refName).getParameter(0)
  or
  exists(TypeTracker t2 |
    result = firebaseSnapshot(refName, t2).track(t2, t)
  )
}

SourceNode firebaseSnapshot(string refName) {
  result = firebaseSnapshot(refName, TypeTracker::end())
}

MethodCallNode firebaseDatabaseRead(string refName) {
  result = firebaseSnapshot(refName).getAMethodCall("val")
}
```

With this addition, `firebaseDatabaseRead("forecast")` finds the call to `snapshot.val()` that contains the value of the forecast.

Here is a run of an example query using the model to find *val* calls.

7.6.8 Summary

We have covered how to use the type-tracking library. To recap, use this template to define forward type-tracking predicates:

```
SourceNode myType(TypeTracker t) {
  t.start() and
  result = /* SourceNode to track */
  or
  exists(TypeTracker t2 |
    result = myType(t2).track(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeTracker::end())
}
```

Use this template to define backward type-tracking predicates:

```
SourceNode myType(TypeBackTracker t) {
  t.start() and
  result = (/* argument to track */).getALocalSource()
  or
  exists(TypeBackTracker t2 |
    result = myType(t2).backtrack(t2, t)
  )
```

(continues on next page)

(continued from previous page)

```
}

SourceNode myType() {
  result = myType(TypeBackTracker::end())
}
```

Note that these predicates all return `SourceNode`, so attempts to track a non-source node, such as an identifier or string literal, will not work. If this becomes an issue, see TypeTracker.smallstep.

Also note that the predicates taking a `TypeTracker` or `TypeBackTracker` can often be made `private`, as they are typically only used as an intermediate result to compute the other predicate.

7.6.9 Limitations

As mentioned, type tracking will track values in and out of function calls and properties, but only within some limits.

For example, type tracking does not always track *through* functions. That is, if a value flows into a parameter and back out of the return value, it might not be tracked back out to the call site again. Heres an example that the model from this tutorial wont find:

```
function wrapDB(database) {
  return { db: database }
}
let wrapper = wrapDB(firebase.database())
wrapper.db.ref("forecast"); // <-- not found
```

This is an example of where data-flow configurations are more powerful.

7.6.10 When to use type tracking

Type tracking and data-flow configurations are different solutions to the same problem, each with their own tradeoffs.

Type tracking can be used in any number of predicates, which may depend on each other in fairly unrestricted ways. The result of one predicate may be the starting point for another. Type-tracking predicates may be mutually recursive. Type-tracking predicates can have any number of extra parameters, making it possible, but optional, to construct source/sink pairs. Omitting source/sink pairs can be useful when there is a huge number of sources and sinks.

Data-flow configurations have more restricted dependencies but are more powerful in other ways. For performance reasons, the sources, sinks, and steps of a configuration should not depend on whether a flow path has been found using that configuration or any other configuration. In that sense, the sources, sinks, and steps must be configured up front and cant be discovered on-the-fly. The upside is that they track flow through functions and callbacks in some ways that type tracking doesnt, which is particularly important for security queries. Also, path queries can only be defined using data-flow configurations.

Prefer type tracking when:

- Disambiguating generically named methods or properties.
- Making reusable library components to be shared between queries.

- The set of source/sink pairs is too large to compute or has insufficient information.
- The information is needed as input to a data-flow configuration.

Prefer data-flow configurations when:

- Tracking user-controlled data – use taint tracking.
- Differentiating between different kinds of user-controlled data – see *Using flow labels for precise data flow analysis*.
- Tracking transformations of a value through generic utility functions.
- Tracking values through string manipulation.
- Generating a path from source to sink – see *Creating path queries*.

Lastly, depending on the code base being analyzed, some alternatives to consider are:

- Using static type information, if analyzing TypeScript code.
- Relying on local data flow.
- Relying on syntactic heuristics such as the name of a method, property, or variable.

7.6.11 Type tracking in the standard libraries

Type tracking is used in a few places in the standard libraries:

- The DOM predicates, documentRef, locationRef, and domValueRef, are implemented with type tracking.
- The HTTP server models, such as Express, use type tracking to track the installation of router handler functions.
- The Firebase and Socket.io models use type tracking to track objects coming from their respective APIs.

7.6.12 Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- QL language reference
- CodeQL tools

7.7 Abstract syntax tree classes for working with JavaScript and TypeScript programs

CodeQL has a large selection of classes for representing the abstract syntax tree of JavaScript and TypeScript programs.

The abstract syntax tree (AST) represents the syntactic structure of a program. Nodes on the AST represent elements such as statements and expressions.

7.7.1 Statement classes

This table lists subclasses of Stmt representing ECMAScript and TypeScript statements.

Statement syntax	CodeQL class	Superclasses
Expr ;	ExprStmt	
Label : Stmt	LabeledStmt	
;	EmptyStmt	
break Label ;	BreakStmt	JumpStmt, BreakOrContinueStmt
case Expr : Stmt	Case	
catch(Identifier) { Stmt }	CatchClause	ControlStmt
class Identifier extends Expr { MemberDeclaration }	ClassDeclStmt	ClassDefinition, ClassOrInterface,
const Identifier = Expr ;	ConstDeclStmt	DeclStmt
continue Label ;	ContinueStmt	JumpStmt, BreakOrContinueStmt
debugger;	DebuggerStmt	
declare global { Stmt }	GlobalAugmentationDeclaration	
declare module StringLiteral { Stmt }	ExternalModuleDeclaration	
default: Stmt	Case	
do Stmt while (Expr)	DoWhileStmt	ControlStmt, LoopStmt
enum Identifier { MemberDeclaration }	EnumDeclaration	NamespaceDefinition
export * from StringLiteral	BulkReExportDeclaration	ReExportDeclaration, ExportDeclar
export default ClassDeclStmt	ExportDefaultDeclaration	ExportDeclaration
export default Expr ;	ExportDefaultDeclaration	ExportDeclaration
export default FunctionDeclStmt	ExportDefaultDeclaration	ExportDeclaration
export { ExportSpecifier };	ExportNamedDeclaration	ExportDeclaration
export DeclStmt	ExportNamedDeclaration	ExportDeclaration
export = Expr ;	ExportAssignDeclaration	
export as namespace Identifier ;	ExportAsNamespaceDeclaration	
for (Expr ; Expr ; Expr) Stmt	ForStmt	ControlStmt, LoopStmt
for (VarAccess in Expr) Stmt	ForInStmt	ControlStmt, LoopStmt, Enhanced
for (VarAccess of Expr) Stmt	ForOfStmt	ControlStmt, LoopStmt, Enhanced
function Identifier (Parameter) { Stmt }	FunctionDeclStmt	Function
if (Expr) Stmt else Stmt	IfStmt	ControlStmt
import { ImportSpecifier from StringLiteral	ImportDeclaration	Import
import Identifier = Expr ;	ImportEqualsDeclaration	
interface Identifier { MemberDeclaration }	InterfaceDeclaration	InterfaceDefinition, ClassOrInterfa
let Identifier = Expr ;	LetStmt	DeclStmt
namespace Identifier { Stmt }	NamespaceDeclaration	NamespaceDefinition
return Expr ;	ReturnStmt	JumpStmt
switch (Expr) { Case }	SwitchStmt	ControlStmt
throw Expr ;	ThrowStmt	JumpStmt
try { Stmt } CatchClause finally { Stmt }	TryStmt	ControlStmt
type Identifier = TypeExpr ;	TypeAliasDeclaration	TypeParameterized
var Identifier = Expr ;	VarDeclStmt	DeclStmt
while (Expr) Stmt	WhileStmt	ControlStmt, LoopStmt
with (Expr) Stmt	WithStmt	ControlStmt

Table 1 – continued from pre

Statement syntax	CodeQL class	Superclasses
{ Stmt }	BlockStmt	

7.7.2 Expression classes

There is a large number of expression classes, so we present them by category. All classes in this section are subclasses of Expr, except where noted otherwise.

Literals

All classes in this subsection are subclasses of Literal.

Expression syntax	CodeQL class
`true`	BooleanLiteral
`23`	NumberLiteral
`4.2`	NumberLiteral
`"Hello"`	StringLiteral
`/ab*c?/`	RegExpLiteral
`null`	NullLiteral

Identifiers

All identifiers are represented by the class Identifier, which has subclasses to represent specific kinds of identifiers:

- VarAccess: an identifier that refers to a variable
- VarDecl: an identifier that declares a variable, for example `x` in `var x = "hi"` or in `function(x) { }`
- VarRef: a VarAccess or a VarDecl
- Label: an identifier that refers to a statement label or a property, not a variable; in the following examples, `l` and `p` are labels:
  - `break l;`
  - `l: for(;;) {}`
  - `x.p`
  - `{ p: 42 }`

Primary expressions

All classes in this subsection are subclasses of Expr.

Expression syntax	CodeQL class	Superclasses	Remarks
`this`	ThisExpr		
`[` Expr `]`	ArrayExpr		
`{` Property `}`	ObjectExpr		
`function (` Parameter `) {` Stmt `}`	FunctionExpr	Function	
`(` Parameter `) =>` Expr	ArrowFunctionExpr	Function	
`(` Expr `)`	ParExpr		
`` ` ` ``	TemplateLiteral		an element in a TemplateLiteral is either a TemplateElement representing a constant template element, or some other expression representing an interpolated expression of the form `${` Expr `}`
Expr `` ` ` ``	TaggedTemplateExpr		an element in a TaggedTemplateExpr is either a TemplateElement representing a constant template element, or some other expression representing an interpolated expression of the form `${` Expr `}`

Properties

All classes in this subsection are subclasses of Property. Note that Property is not a subclass of Expr.

Property syntax	CodeQL class	Superclasses
Identifier `:` Expr	ValueProperty	
`get` Identifier `() {` Stmt `}`	PropertyGetter	PropertyAccessor
`set` Identifier `(` Identifier `) {` Stmt `}`	PropertySetter	PropertyAccessor

Property accesses

All classes in this subsection are subclasses of PropAccess.

Expression syntax	CodeQL class
Expr `.` Identifier	DotExpr
Expr `[` Expr `]`	IndexExpr

Function calls and `new`

All classes in this subsection are subclasses of InvokeExpr.

Expression syntax	CodeQL class	Remarks
Expr (Expr)	CallExpr	
Expr . Identifier (Expr)	MethodCallExpr	this also includes calls of the form Expr [Expr] (Expr)
new Expr (Expr)	NewExpr	

Unary expressions

All classes in this subsection are subclasses of UnaryExpr.

Expression syntax	CodeQL class
~ Expr	BitNotExpr
- Expr	NegExpr
+ Expr	PlusExpr
! Expr	LogNotExpr
typeof Expr	TypeofExpr
void Expr	VoidExpr
delete Expr	DeleteExpr
... Expr	SpreadElement

Binary expressions

All classes in this subsection are subclasses of BinaryExpr.

Expression syntax	CodeQL class	Superclasses
Expr * Expr	MulExpr	
Expr / Expr	DivExpr	
Expr % Expr	ModExpr	
Expr ** Expr	ExpExpr	
Expr + Expr	AddExpr	
Expr - Expr	SubExpr	
Expr << Expr	LShiftExpr	
Expr >> Expr	RShiftExpr	
Expr >>> Expr	URShiftExpr	
Expr && Expr	LogAndExpr	
Expr || Expr	LogOrExpr	
Expr < Expr	LTExpr	Comparison
Expr > Expr	GTExpr	Comparison
Expr <= Expr	LEExpr	Comparison
Expr >= Expr	GEExpr	Comparison
Expr == Expr	EqExpr	EqualityTest, Comparison
Expr != Expr	NEqExpr	EqualityTest, Comparison
Expr === Expr	StrictEqExpr	EqualityTest, Comparison
Expr !== Expr	StrictNEqExpr	EqualityTest, Comparison
Expr & Expr	BitAndExpr	
Expr | Expr	BitOrExpr	
Expr ^ Expr	XOrExpr	
Expr in Expr	InExpr	
Expr instanceof Expr	InstanceofExpr	

Assignment expressions

All classes in this table are subclasses of Assignment.

Expression syntax	CodeQL class	Superclasses
Expr = Expr	AssignExpr	
Expr += Expr	AssignAddExpr	CompoundAssignExpr
Expr -= Expr	AssignSubExpr	CompoundAssignExpr
Expr *= Expr	AssignMulExpr	CompoundAssignExpr
Expr **= Expr	AssignExpExpr	CompoundAssignExpr
Expr /= Expr	AssignDivExpr	CompoundAssignExpr
Expr %= Expr	AssignModExpr	CompoundAssignExpr
Expr &= Expr	AssignAndExpr	CompoundAssignExpr
Expr |= Expr	AssignOrExpr	CompoundAssignExpr
Expr ^= Expr	AssignXOrExpr	CompoundAssignExpr
Expr <<= Expr	AssignLShiftExpr	CompoundAssignExpr
Expr >>= Expr	AssignRShiftExpr	CompoundAssignExpr
Expr >>>= Expr	AssignURShiftExpr	CompoundAssignExpr

Update expressions

All classes in this table are subclasses of UpdateExpr.

Expression syntax	CodeQL class
Expr `++`	PostIncExpr
Expr `--`	PostDecExpr
`++` Expr	PreIncExpr
`--` Expr	PreDecExpr

Miscellaneous

All classes in this table are subclasses of Expr.

Expression syntax	CodeQL class
Expr `?` Expr `:` Expr	ConditionalExpr
Expr `,` `,` Expr	SeqExpr
`await` Expr	AwaitExpr
`yield` Expr	YieldExpr

7.7.3 Further reading

- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- QL language reference
- CodeQL tools

7.8 Data flow cheat sheet for JavaScript

This article describes parts of the JavaScript libraries commonly used for variant analysis and in data flow queries.

7.8.1 Taint tracking path queries

Use the following template to create a taint tracking path query:

```
/**
 * @kind path-problem
 */
import javascript
import DataFlow
import DataFlow::PathGraph

class MyConfig extends TaintTracking::Configuration {
  MyConfig() { this = "MyConfig" }
  override predicate isSource(Node node) { ... }
  override predicate isSink(Node node) { ... }
```

(continues on next page)

(continued from previous page)

```
  override predicate isAdditionalTaintStep(Node pred, Node succ) { ... }
}

from MyConfig cfg, PathNode source, PathNode sink
where cfg.hasFlowPath(source, sink)
select sink.getNode(), source, sink, "taint from $@.", source.getNode(), "here"
```

This query reports flow paths which:

- Begin at a node matched by isSource.
- Step through variables, function calls, properties, strings, arrays, promises, exceptions, and steps added by isAdditionalTaintStep.
- End at a node matched by isSink.

See also: Global data flow and *Creating path queries*.

7.8.2 DataFlow module

Use data flow nodes to match program elements independently of syntax. See also: *Analyzing data flow in JavaScript and TypeScript*.

Predicates in the `DataFlow::` module:

- moduleImport – finds uses of a module
- moduleMember – finds uses of a module member
- globalVarRef – finds uses of a global variable

Classes and member predicates in the `DataFlow::` module:

- **Node – something that can have a value, such as an expression, declaration, or SSA variable**
  - getALocalSource – find the node that this came from
  - getTopLevel – top-level scope enclosing this node
  - getFile – file containing this node
  - getIntValue – value of this node if its is an integer constant
  - getStringValue – value of this node if its is a string constant
  - mayHaveBooleanValue – check if the value is `true` or `false`
- **SourceNode extends Node – function call, parameter, object creation, or reference to a property or global variable**
  - getACall – find calls with this as the callee
  - getAnInstantiation – find `new`-calls with this as the callee
  - getAnInvocation – find calls or `new`-calls with this as the callee
  - getAMethodCall – find method calls with this as the receiver
  - getAMemberCall – find calls with a member of this as the receiver
  - getAPropertyRead – find property reads with this as the base

  - getAPropertyWrite – find property writes with this as the base
  - getAPropertySource – find nodes flowing into a property of this node
- **InvokeNode, NewNode, CallNode, MethodCallNode extends SourceNode – call to a function or constructor**
  - getArgument – an argument to the call
  - getCalleeNode – node being invoked as a function
  - getCalleeName – name of the variable or property being called
  - getOptionArgument – a named argument passed in through an object literal
  - getCallback – a function passed as a callback
  - getACallee - a function being called here
  - (MethodCallNode).getMethodName – name of the method being invoked
  - (MethodCallNode).getReceiver – receiver of the method call
- **FunctionNode extends SourceNode – definition of a function, including closures, methods, and class constructors**
  - getName – name of the function, derived from a variable or property name
  - getParameter – a parameter of the function
  - getReceiver – the node representing the value of `this`
  - getAReturn – get a returned expression
- **ParameterNode extends SourceNode – parameter of a function**
  - getName – the parameter name, if it has one
- **ClassNode extends SourceNode – class declaration or function that acts as a class**
  - getName – name of the class, derived from a variable or property name
  - getConstructor – the constructor function
  - getInstanceMethod – get an instance method by name
  - getStaticMethod – get a static method by name
  - getAnInstanceReference – find references to an instance of the class
  - getAClassReference – find references to the class itself
- **ObjectLiteralNode extends SourceNode – object literal**
  - getAPropertyWrite – a property in the object literal
  - getAPropertySource – value flowing into a property
- **ArrayCreationNode extends SourceNode – array literal or call to `Array` constructor**
  - getElement – an element of the array
- **PropRef, PropRead, PropWrite – read or write of a property**
  - getPropertyName – name of the property, if it is constant

  - getPropertyNameExpr – expression holding the name of the property
  - getBase – object whose property is accessed
  - (PropWrite).getRhs – right-hand side of the property assignment

7.8.3 StringOps module

- StringOps::Concatenation – string concatenation, using a plus operator, template literal, or array join call
- StringOps::StartsWith – check if a string starts with something
- StringOps::EndsWith – check if a string ends with something
- StringOps::Includes – check if a string contains something

7.8.4 Utility

- ExtendCall – call that copies properties from one object to another
- JsonParserCall – call that deserializes a JSON string
- PropertyProjection – call that extracts nested properties by name

7.8.5 System and Network

- ClientRequest – outgoing network request
- DatabaseAccess – query being submitted to a database
- FileNameSource – reference to a filename
- **FileSystemAccess – file system operation**
  - FileSystemReadAccess – reading the contents of a file
  - FileSystemWriteAccess – writing to the contents of a file
- PersistentReadAccess – reading from persistent storage, like cookies
- PersistentWriteAccess – writing to persistent storage
- RemoteFlowSource – source of untrusted user input
- SystemCommandExecution – execution of a system command

7.8.6 Files

- File, Folder extends Container – file or folder in the database
  - getBaseName – the name of the file or folder
  - getRelativePath – path relative to the database root

7.8.7 AST nodes

See also: *Abstract syntax tree classes for working with JavaScript and TypeScript programs*.

Conversion between DataFlow and AST nodes:

- Node.asExpr() – convert node to an expression, if possible

- Expr.flow() – convert expression to a node (always possible)
- DataFlow::valueNode – convert expression or declaration to a node
- DataFlow::parameterNode – convert a parameter to a node
- DataFlow::thisNode – get the receiver node of a function

7.8.8 String matching

- x.matches(escape%) – holds if x starts with escape
- x.regexpMatch(escape.*) – holds if x starts with escape
- x.regexpMatch((?i).*escape.*) – holds if x contains escape (case insensitive)

7.8.9 Type tracking

See also: *Using type tracking for API modeling*.

Use the following template to define forward type tracking predicates:

```
import DataFlow

SourceNode myType(TypeTracker t) {
  t.start() and
  result = /* SourceNode to track */
  or
  exists(TypeTracker t2 |
    result = myType(t2).track(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeTracker::end())
}
```

Use the following template to define backward type tracking predicates:

```
import DataFlow

SourceNode myType(TypeBackTracker t) {
  t.start() and
  result = (/* argument to track */).getALocalSource()
  or
  exists(TypeBackTracker t2 |
    result = myType(t2).backtrack(t2, t)
  )
}

SourceNode myType() {
  result = myType(TypeBackTracker::end())
}
```

7.8.10 Troubleshooting

- Using a call node as as sink? Try using getArgument to get an *argument* of the call node instead.
- Trying to use moduleImport or moduleMember as a call node? Try using getACall to get a *call* to the imported function, instead of the function itself.
- Compilation fails due to incompatible types? Make sure AST nodes and DataFlow nodes are not mixed up. Use asExpr() or flow() to convert.

7.8.11 Further reading

- Exploring data flow with path queries
- CodeQL queries for JavaScript
- Example queries for JavaScript
- CodeQL library reference for JavaScript
- QL language reference
- CodeQL tools
- *Basic query for JavaScript code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for JavaScript*: When youre analyzing a JavaScript program, you can make use of the large collection of classes in the CodeQL library for JavaScript.
- *CodeQL library for TypeScript*: When youre analyzing a TypeScript program, you can make use of the large collection of classes in the CodeQL library for TypeScript.
- *Analyzing data flow in JavaScript and TypeScript*: This topic describes how data flow analysis is implemented in the CodeQL libraries for JavaScript/TypeScript and includes examples to help you write your own data flow queries.
- *Using flow labels for precise data flow analysis*: You can associate flow labels with each value tracked by the flow analysis to determine whether the flow contains potential vulnerabilities.
- *Using type tracking for API modeling*: You can track data through an API by creating a model using the CodeQL type-tracking library for JavaScript.
- *Abstract syntax tree classes for working with JavaScript and TypeScript programs*: CodeQL has a large selection of classes for representing the abstract syntax tree of JavaScript and TypeScript programs.
- *Data flow cheat sheet for JavaScript*: This article describes parts of the JavaScript libraries commonly used for variant analysis and in data flow queries.

CHAPTER

EIGHT

CODEQL FOR PYTHON

Experiment and learn how to write effective and efficient queries for CodeQL databases generated from Python codebases.

8.1 Basic query for Python code

Learn to write and run a simple CodeQL query using LGTM.

8.1.1 About the query

The query were going to run performs a basic search of the code for `if` statements that are redundant, in the sense that they only include a `pass` statement. For example, code such as:

```
if error: pass
```

8.1.2 Running the query

1. In the main search box on LGTM.com, search for the project you want to query. For tips, see Searching.
2. Click the project in the search results.
3. Click **Query this project**.

   This opens the query console. (For information about using this, see Using the query console.)

   > Note
   >
   > Alternatively, you can go straight to the query console by clicking **Query console** (at the top of any page), selecting **Python** from the **Language** drop-down list, then choosing one or more projects to query from those displayed in the **Project** drop-down list.

4. Copy the following query into the text box in the query console:

```
import python

from If ifstmt, Stmt pass
where pass = ifstmt.getStmt(0) and
  pass instanceof Pass
select ifstmt, "This 'if' statement is redundant."
```

LGTM checks whether your query compiles and, if all is well, the **Run** button changes to green to indicate that you can go ahead and run the query.

5. Click **Run**.

   The name of the project you are querying, and the ID of the most recently analyzed commit to the project, are listed below the query box. To the right of this is an icon that indicates the progress of the query operation:

   Note

   Your query is always run against the most recently analyzed commit to the selected project.

   The query will take a few moments to return results. When the query completes, the results are displayed below the project name. The query results are listed in two columns, corresponding to the two expressions in the `select` clause of the query. The first column corresponds to the expression `ifstmt` and is linked to the location in the source code of the project where `ifstmt` occurs. The second column is the alert message.

   Example query results

   Note

   An ellipsis () at the bottom of the table indicates that the entire list is not displayedclick it to show more results.

6. If any matching code is found, click a link in the `ifstmt` column to view the `if` statement in the code viewer.

   The matching `if` statement is highlighted with a yellow background in the code viewer. If any code in the file also matches a query from the standard query library for that language, you will see a red alert message at the appropriate point within the code.

About the query structure

After the initial `import` statement, this simple query comprises three parts that serve similar purposes to the FROM, WHERE, and SELECT parts of an SQL query.

Query part	Purpose	Details
`import python`	Imports the standard CodeQL libraries for Python.	Every query begins with one or more `import` statements.
`from If ifstmt, Stmt pass`	Defines the variables for the query. Declarations are of the form: `<type> <variable name>`	We use: • an `If` variable for `if` statements • a `Stmt` variable for the statement
`where pass = ifstmt.getStmt(0) and pass instanceof Pass`	Defines a condition on the variables.	`pass = ifstmt.getStmt(0)`: `pass` is the first statement in the `if` statement. `pass instanceof Pass`: `pass` must be a pass statement. In other words, the first statement contained in the `if` statement is a `pass` statement.
`select ifstmt, "This 'if' statement is redundant."`	Defines what to report for each match. `select` statements for queries that are used to find instances of poor coding practice are always in the form: `select <program element>, "<alert message>"`	Reports the resulting `if` statement with a string that explains the problem.

8.1.3 Extend the query

Query writing is an inherently iterative process. You write a simple query and then, when you run it, you discover examples that you had not previously considered, or opportunities for improvement.

Remove false positive results

Browsing the results of our basic query shows that it could be improved. Among the results you are likely to find examples of `if` statements with an `else` branch, where a `pass` statement does serve a purpose. For example:

```
if cond():
  pass
else:
  do_something()
```

In this case, identifying the `if` statement with the `pass` statement as redundant is a false positive. One solution to this is to modify the query to ignore `pass` statements if the `if` statement has an `else` branch.

To exclude `if` statements that have an `else` branch:

1. Extend the `where` clause to include the following extra condition:

   ```
   and not exists(ifstmt.getOrelse())
   ```

   The `where` clause is now:

```
where pass = ifstmt.getStmt(0) and
  pass instanceof Pass and
  not exists(ifstmt.getOrelse())
```

2. Click **Run**.

   There are now fewer results because `if` statements with an `else` branch are no longer included.

See this in the query console

8.1.4 Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- QL language reference
- CodeQL tools

8.2 CodeQL library for Python

When you need to analyze a Python program, you can make use of the large collection of classes in the CodeQL library for Python.

8.2.1 About the CodeQL library for Python

The CodeQL library for each programming language uses classes with abstractions and predicates to present data in an object-oriented form.

Each CodeQL library is implemented as a set of QL modules, that is, files with the extension `.qll`. The module `python.qll` imports all the core Python library modules, so you can include the complete library by beginning your query with:

```
import python
```

The CodeQL library for Python incorporates a large number of classes. Each class corresponds either to one kind of entity in Python source code or to an entity that can be derived from the source code using static analysis. These classes can be divided into four categories:

- **Syntactic** - classes that represent entities in the Python source code.
- **Control flow** - classes that represent entities from the control flow graphs.
- **Type inference** - classes that represent the inferred values and types of entities in the Python source code.
- **Taint tracking** - classes that represent the source, sinks and kinds of taint used to implement taint-tracking queries.

8.2.2 Syntactic classes

This part of the library represents the Python source code. The `Module`, `Class`, and `Function` classes correspond to Python modules, classes, and functions respectively, collectively these are known as `Scope` classes. Each `Scope`

contains a list of statements each of which is represented by a subclass of the class `Stmt`. Statements themselves can contain other statements or expressions which are represented by subclasses of `Expr`. Finally, there are a few additional classes for the parts of more complex expressions such as list comprehensions. Collectively these classes are subclasses of `AstNode` and form an Abstract syntax tree (AST). The root of each AST is a `Module`. Symbolic information is attached to the AST in the form of variables (represented by the class `Variable`). For more information, see Abstract syntax tree and Symbolic information on Wikipedia.

Scope

A Python program is a group of modules. Technically a module is just a list of statements, but we often think of it as composed of classes and functions. These top-level entities, the module, class, and function are represented by the three CodeQL classes Module, Class and Function which are all subclasses of `Scope`.

- `Scope`
  - `Module`
  - `Class`
  - `Function`

All scopes are basically a list of statements, although `Scope` classes have additional attributes such as names. For example, the following query finds all functions whose scope (the scope in which they are declared) is also a function:

```
import python

from Function f
where f.getScope() instanceof Function
select f
```

See this in the query console on LGTM.com. Many projects have nested functions.

Statement

A statement is represented by the Stmt class which has about 20 subclasses representing the various kinds of statements, such as the `Pass` statement, the `Return` statement or the `For` statement. Statements are usually made up of parts. The most common of these is the expression, represented by the `Expr` class. For example, take the following Python `for` statement:

```
for var in seq:
    pass
else:
    return 0
```

The For class representing the `for` statement has a number of member predicates to access its parts:

- `getTarget()` returns the `Expr` representing the variable `var`.
- `getIter()` returns the `Expr` resenting the variable `seq`.
- `getBody()` returns the statement list body.
- `getStmt(0)` returns the pass `Stmt`.
- `getOrElse()` returns the `StmtList` containing the return statement.

Expression

Most statements are made up of expressions. The Expr class is the superclass of all expression classes, of which there are about 30 including calls, comprehensions, tuples, lists and arithmetic operations. For example, the Python expression `a+2` is represented by the `BinaryExpr` class:

- `getLeft()` returns the `Expr` representing the `a`.
- `getRight()` returns the `Expr` representing the `2`.

As an example, to find expressions of the form `a+2` where the left is a simple name and the right is a numeric constant we can use the following query:

Finding expressions of the form a+2

```
import python

from BinaryExpr bin
where bin.getLeft() instanceof Name and bin.getRight() instanceof Num
select bin
```

See this in the query console on LGTM.com. Many projects include examples of this pattern.

Variable

Variables are represented by the Variable class in the CodeQL library. There are two subclasses, `LocalVariable` for function-level and class-level variables and `GlobalVariable` for module-level variables.

Other source code elements

Although the meaning of the program is encoded by the syntactic elements, `Scope`, `Stmt` and `Expr` there are some parts of the source code not covered by the abstract syntax tree. The most useful of these is the Comment class which describes comments in the source code.

Examples

Each syntactic element in Python source is recorded in the CodeQL database. These can be queried via the corresponding class. Let us start with a couple of simple examples.

1. Finding all `finally` blocks

For our first example, we can find all `finally` blocks by using the `Try` class:

Find all `finally` blocks

```
import python

from Try t
select t.getFinalbody()
```

See this in the query console on LGTM.com. Many projects include examples of this pattern.

2. Finding `except` blocks that do nothing

For our second example, we can use a simplified version of a query from the standard query set. We look for all `except` blocks that do nothing.

A block that does nothing is one that contains no statements except `pass` statements. We can encode this as:

```
not exists(Stmt s | s = ex.getAStmt() | not s instanceof Pass)
```

where `ex` is an `ExceptStmt` and `Pass` is the class representing `pass` statements. Instead of using the double negative, **no** *statements that are* **not** *pass statements*, this can also be expressed positively, *all statements must be pass statements.* The positive form is expressed using the `forall` quantifier:

```
forall(Stmt s | s = ex.getAStmt() | s instanceof Pass)
```

Both forms are equivalent. Using the positive expression, the whole query looks like this:

Find pass-only `except` blocks

```
import python

from ExceptStmt ex
where forall(Stmt s | s = ex.getAStmt() | s instanceof Pass)
select ex
```

See this in the query console on LGTM.com. Many projects include pass-only `except` blocks.

Summary

The most commonly used standard classes in the syntactic part of the library are organized as follows:

`Module`, `Class`, `Function`, `Stmt`, and `Expr` - they are all subclasses of AstNode.

Abstract syntax tree

- `AstNode`
  - `Module` – A Python module
  - `Class` – The body of a class definition
  - `Function` – The body of a function definition
  - `Stmt` – A statement

    `Assert` – An `assert` statement

    `Assign` – An assignment

    - `AssignStmt` – An assignment statement, `x = y`
    - `ClassDef` – A class definition statement
    - `FunctionDef` – A function definition statement

    `AugAssign` – An augmented assignment, `x += y`

    `Break` – A `break` statement

Continue – A `continue` statement

Delete – A `del` statement

ExceptStmt – The `except` part of a `try` statement

Exec – An exec statement

For – A `for` statement

If – An `if` statement

Pass – A `pass` statement

Print – A `print` statement (Python 2 only)

Raise – A raise statement

Return – A `return` statement

Try – A `try` statement

While – A `while` statement

With – A `with` statement

- Expr – An expression

  Attribute – An attribute, `obj.attr`

  Call – A function call, `f(arg)`

  IfExp – A conditional expression, `x if cond else y`

  Lambda - A lambda expression

  Yield – A `yield` expression

  Bytes – A bytes literal, `b"x"` or (in Python 2) `"x"`

  Unicode – A unicode literal, `u"x"` or (in Python 3) `"x"`

  Num – A numeric literal, `3` or `4.2`

    - IntegerLiteral
    - FloatLiteral
    - ImaginaryLiteral

  Dict – A dictionary literal, `{'a': 2}`

  Set – A set literal, `{'a', 'b'}`

  List – A list literal, `['a', 'b']`

  Tuple – A tuple literal, `('a', 'b')`

  DictComp – A dictionary comprehension, `{k: v for ...}`

  SetComp – A set comprehension, `{x for ...}`

  ListComp – A list comprehension, `[x for ...]`

  GenExpr – A generator expression, `(x for ...)`

  Subscript – A subscript operation, `seq[index]`

`Name` – A reference to a variable, `var`

`UnaryExpr` – A unary operation, `-x`

`BinaryExpr` – A binary operation, `x+y`

`Compare` – A comparison operation, `0 < x < 10`

`BoolExpr` – Short circuit logical operations, `x and y`, `x or y`

Variables

- `Variable` – A variable
  - `LocalVariable` – A variable local to a function or a class
  - `GlobalVariable` – A module level variable

Other

- `Comment` – A comment

8.2.3 Control flow classes

This part of the library represents the control flow graph of each `Scope` (classes, functions, and modules). Each `Scope` contains a graph of `ControlFlowNode` elements. Each scope has a single entry point and at least one (potentially many) exit points. To speed up control and data flow analysis, control flow nodes are grouped into basic blocks. For more information, see Basic block on Wikipedia.

Example

If we want to find the longest sequence of code without any branches, we need to consider control flow. A `BasicBlock` is, by definition, a sequence of code without any branches, so we just need to find the longest `BasicBlock`.

First of all we introduce a simple predicate `bb_length()` which relates `BasicBlocks` to their length.

```
int bb_length(BasicBlock b) {
    result = max(int i | exists(b.getNode(i))) + 1
}
```

Each `ControlFlowNode` within a `BasicBlock` is numbered consecutively, starting from zero, therefore the length of a `BasicBlock` is equal to one more than the largest index within that `BasicBlock`.

Using this predicate we can select the longest `BasicBlock` by selecting the `BasicBlock` whose length is equal to the maximum length of any `BasicBlock`:

Find the longest sequence of code without branches

```
import python

int bb_length(BasicBlock b) {
    result = max(int i | exists(b.getNode(i)) | i) + 1
}
```

(continues on next page)

(continued from previous page)

```
from BasicBlock b
where bb_length(b) = max(bb_length(_))
select b
```

See this in the query console on LGTM.com. When we ran it on the LGTM.com demo projects, the *openstack/nova* and *ytdl-org/youtube-dl* projects both contained source code results for this query.

> Note
>
> The special underscore variable `_` means any value; so `bb_length(_)` is the length of any block.

Summary

The classes in the control-flow part of the library are:

- ControlFlowNode – A control-flow node. There is a one-to-many relation between AST nodes and control-flow nodes.
- BasicBlock – A non branching list of control-flow nodes.

8.2.4 Type-inference classes

The CodeQL library for Python also supplies some classes for accessing the inferred types of values. The classes `Value` and `ClassValue` allow you to query the possible classes that an expression may have at runtime.

Example

For example, which `ClassValues` are iterable can be determined using the query:

Find iterable ClassValues

```
import python

from ClassValue cls
where cls.hasAttribute("__iter__")
select cls
```

See this in the query console on LGTM.com This query returns a list of classes for the projects analyzed. If you want to include the results for `builtin` classes, which do not have any Python source code, show the non-source results. For more information, see builtin classes in the Python documentation.

Summary

- Value
  - `ClassValue`
  - `CallableValue`
  - `ModuleValue`

For more information about these classes, see *Pointer analysis and type inference in Python*.

8.2.5 Taint-tracking classes

The CodeQL library for Python also supplies classes to specify taint-tracking analyses. The `Configuration` class can be overridden to specify a taint-tracking analysis, by specifying source, sinks, sanitizers and additional flow steps. For those analyses that require additional types of taint to be tracked the `TaintKind` class can be overridden.

Summary

- TaintKind
- Configuration

For more information about these classes, see *Analyzing data flow and tracking tainted data in Python*.

8.2.6 Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- QL language reference
- CodeQL tools

8.3 Functions in Python

You can use syntactic classes from the standard CodeQL library to find Python functions and identify calls to them.

These examples use the standard CodeQL class Function. For more information, see *CodeQL library for Python*.

8.3.1 Finding all functions called get

In this example we look for all the getters in a program. Programmers moving to Python from Java are often tempted to write lots of getter and setter methods, rather than use properties. We might want to find those methods.

Using the member predicate `Function.getName()`, we can list all of the getter functions in a database:

> Tip
>
> Instead of copying this query, try typing the code. As you start to write a name that matches a library class, a pop-up is displayed making it easy for you to select the class that you want.

```
import python

from Function f
where f.getName().matches("get%")
select f, "This is a function called get..."
```

See this in the query console on LGTM.com. This query typically finds a large number of results. Usually, many of these results are for functions (rather than methods) which we are not interested in.

8.3.2 Finding all methods called get

You can modify the query above to return more interesting results. As we are only interested in methods, we can use the `Function.isMethod()` predicate to refine the query.

```
import python

from Function f
where f.getName().matches("get%") and f.isMethod()
select f, "This is a method called get..."
```

See this in the query console on LGTM.com. This finds methods whose name starts with `"get"`, but many of those are not the sort of simple getters we are interested in.

8.3.3 Finding one line methods called get

We can modify the query further to include only methods whose body consists of a single statement. We do this by counting the number of lines in each method.

```
import python

from Function f
where f.getName().matches("get%") and f.isMethod()
 and count(f.getAStmt()) = 1
select f, "This function is (probably) a getter."
```

See this in the query console on LGTM.com. This query returns fewer results, but if you examine the results you can see that there are still refinements to be made. This is refined further in *Expressions and statements in Python*.

8.3.4 Finding a call to a specific function

This query uses `Call` and `Name` to find calls to the function `eval` - which might potentially be a security hazard.

```
import python

from Call call, Name name
where call.getFunc() = name and name.getId() = "eval"
select call, "call to 'eval'."
```

See this in the query console on LGTM.com. Some of the demo projects on LGTM.com use this function.

The `Call` class represents calls in Python. The `Call.getFunc()` predicate gets the expression being called. `Name.getId()` gets the identifier (as a string) of the `Name` expression. Due to the dynamic nature of Python, this query will select any call of the form `eval(...)` regardless of whether it is a call to the built-in function `eval` or not. In a later tutorial we will see how to use the type-inference library to find calls to the built-in function `eval` regardless of name of the variable called.

8.3.5 Further reading

- CodeQL queries for Python
- Example queries for Python

- CodeQL library reference for Python
- QL language reference
- CodeQL tools

8.4 Expressions and statements in Python

You can use syntactic classes from the CodeQL library to explore how Python expressions and statements are used in a code base.

8.4.1 Statements

The bulk of Python code takes the form of statements. Each different type of statement in Python is represented by a separate CodeQL class.

Here is the full class hierarchy:

- `Stmt` – A statement
  - `Assert` – An `assert` statement
  - `Assign`

    `AssignStmt` – An assignment statement, `x = y`

    `ClassDef` – A class definition statement

    `FunctionDef` – A function definition statement
  - `AugAssign` – An augmented assignment, `x += y`
  - `Break` – A `break` statement
  - `Continue` – A `continue` statement
  - `Delete` – A `del` statement
  - `ExceptStmt` – The `except` part of a `try` statement
  - `Exec` – An `exec` statement
  - `For` – A `for` statement
  - `Global` – A `global` statement
  - `If` – An `if` statement
  - `ImportStar` – A `from xxx import *` statement
  - `Import` – Any other `import` statement
  - `Nonlocal` – A `nonlocal` statement
  - `Pass` – A `pass` statement
  - `Print` – A `print` statement (Python 2 only)
  - `Raise` – A `raise` statement
  - `Return` – A `return` statement
  - `Try` – A `try` statement

  - `While` – A `while` statement
  - `With` – A `with` statement

Example finding redundant global statements

The `global` statement in Python declares a variable with a global (module-level) scope, when it would otherwise be local. Using the `global` statement outside a class or function is redundant as the variable is already global.

```
import python

from Global g
where g.getScope() instanceof Module
select g
```

See this in the query console on LGTM.com. None of the demo projects on LGTM.com has a global statement that matches this pattern.

The line: `g.getScope() instanceof Module` ensures that the `Scope` of `Global g` is a `Module`, rather than a class or function.

Example finding if statements with redundant branches

An `if` statement where one branch is composed of just `pass` statements could be simplified by negating the condition and dropping the `else` clause.

```
if cond():
    pass
else:
    do_something
```

To find statements like this that could be simplified we can write a query.

```
import python

from If i, StmtList l
where (l = i.getBody() or l = i.getOrelse())
  and forall(Stmt p | p = l.getAnItem() | p instanceof Pass)
select i
```

See this in the query console on LGTM.com. Many projects have some `if` statements that match this pattern.

The line: `(l = i.getBody() or l = i.getOrelse())` restricts the `StmtList l` to branches of the `if` statement.

The line: `forall(Stmt p | p = l.getAnItem() | p instanceof Pass)` ensures that all statements in `l` are `pass` statements.

8.4.2 Expressions

Each kind of Python expression has its own class. Here is the full class hierarchy:

- `Expr` – An expression
  - `Attribute` – An attribute, `obj.attr`

- `BinaryExpr` – A binary operation, `x+y`
- `BoolExpr` – Short circuit logical operations, `x and y`, `x or y`
- `Bytes` – A bytes literal, `b"x"` or (in Python 2) `"x"`
- `Call` – A function call, `f(arg)`
- `Compare` – A comparison operation, `0 < x < 10`
- `Dict` – A dictionary literal, `{'a': 2}`
- `DictComp` – A dictionary comprehension, `{k: v for ...}`
- `Ellipsis` – An ellipsis expression, `...`
- `GeneratorExp` – A generator expression
- `IfExp` – A conditional expression, `x if cond else y`
- `ImportExpr` – An artificial expression representing the module imported
- `ImportMember` – An artificial expression representing importing a value from a module (part of an `from xxx import *` statement)
- `Lambda` – A lambda expression
- `List` – A list literal, `['a', 'b']`
- `ListComp` – A list comprehension, `[x for ...]`
- `Name` – A reference to a variable, `var`
- `Num` – A numeric literal, `3` or `4.2`

  `FloatLiteral`

  `ImaginaryLiteral`

  `IntegerLiteral`
- `Repr` – A backticks expression, `x` (Python 2 only)
- `Set` – A set literal, `{'a', 'b'}`
- `SetComp` – A set comprehension, `{x for ...}`
- `Slice` – A slice; the `0:1` in the expression `seq[0:1]`
- `Starred` – A starred expression, `*x` in the context of a multiple assignment: `y, *x = 1,2,3` (Python 3 only)
- `StrConst` – A string literal. In Python 2 either bytes or unicode. In Python 3 only unicode.
- `Subscript` – A subscript operation, `seq[index]`
- `UnaryExpr` – A unary operation, `-x`
- `Unicode` – A unicode literal, `u"x"` or (in Python 3) `"x"`
- `Yield` – A `yield` expression
- `YieldFrom` – A `yield from` expression (Python 3.3+)

Example finding comparisons to integer or string literals using is

Python implementations commonly cache small integers and single character strings, which means that comparisons such as the following often work correctly, but this is not guaranteed and we might want to check for them.

```
x is 10
x is "A"
```

We can check for these using a query.

```
import python

from Compare cmp, Expr literal
where (literal instanceof StrConst or literal instanceof Num)
  and cmp.getOp(0) instanceof Is and cmp.getComparator(0) = literal
select cmp
```

See this in the query console on LGTM.com. Two of the demo projects on LGTM.com use this pattern: *saltstack/salt* and *openstack/nova*.

The clause `cmp.getOp(0) instanceof Is and cmp.getComparator(0) = literal` checks that the first comparison operator is is and that the first comparator is a literal.

> Tip
>
> We have to use `cmp.getOp(0)` and `cmp.getComparator(0)`as there is no `cmp.getOp()` or `cmp.getComparator()`. The reason for this is that a `Compare` expression can have multiple operators. For example, the expression `3 < x < 7` has two operators and two comparators. You use `cmp.getComparator(0)` to get the first comparator (in this example the `x`) and `cmp.getComparator(1)` to get the second comparator (in this example the `7`).

Example finding duplicates in dictionary literals

If there are duplicate keys in a Python dictionary, then the second key will overwrite the first, which is almost certainly a mistake. We can find these duplicates with CodeQL, but the query is more complex than previous examples and will require us to write a `predicate` as a helper.

```
import python

predicate same_key(Expr k1, Expr k2) {
  k1.(Num).getN() = k2.(Num).getN()
  or
  k1.(StrConst).getText() = k2.(StrConst).getText()
}

from Dict d, Expr k1, Expr k2
where k1 = d.getAKey() and k2 = d.getAKey()
  and k1 != k2 and same_key(k1, k2)
select k1, "Duplicate key in dict literal"
```

See this in the query console on LGTM.com. When we ran this query on LGTM.com, the source code of the *saltstack/salt* project contained an example of duplicate dictionary keys. The results were also highlighted as

alerts by the standard Duplicate key in dict literal query. Two of the other demo projects on LGTM.com refer to duplicate dictionary keys in library files. For more information, see Duplicate key in dict literal on LGTM.com.

The supporting predicate `same_key` checks that the keys have the same identifier. Separating this part of the logic into a supporting predicate, instead of directly including it in the query, makes it easier to understand the query as a whole. The casts defined in the predicate restrict the expression to the type specified and allow predicates to be called on the type that is cast-to. For example:

```
x = k1.(Num).getN()
```

is equivalent to

```
exists(Num num | num = k1 | x = num.getN())
```

The short version is usually used as this is easier to read.

Example finding Java-style getters

Returning to the example from *Functions in Python*, the query identified all methods with a single line of code and a name starting with `get`.

```
import python

from Function f
where f.getName().matches("get%") and f.isMethod()
    and count(f.getAStmt()) = 1
select f, "This function is (probably) a getter."
```

This basic query can be improved by checking that the one line of code is a Java-style getter of the form `return self.attr`.

```
import python

from Function f, Return ret, Attribute attr, Name self
where f.getName().matches("get%") and f.isMethod()
    and ret = f.getStmt(0) and ret.getValue() = attr
    and attr.getObject() = self and self.getId() = "self"
select f, "This function is a Java-style getter."
```

See this in the query console on LGTM.com. Of the demo projects on LGTM.com, only the *openstack/nova* project has examples of functions that appear to be Java-style getters.

```
ret = f.getStmt(0) and ret.getValue() = attr
```

This condition checks that the first line in the method is a return statement and that the expression returned (`ret.getValue()`) is an `Attribute` expression. Note that the equality `ret.getValue() = attr` means that `ret.getValue()` is restricted to `Attributes`, since `attr` is an `Attribute`.

```
attr.getObject() = self and self.getId() = "self"
```

This condition checks that the value of the attribute (the expression to the left of the dot in `value.attr`) is an access to a variable called `"self"`.

8.4.3 Class and function definitions

As Python is a dynamically typed language, class, and function definitions are executable statements. This means that a class statement is both a statement and a scope containing statements. To represent this cleanly the class definition is broken into a number of parts. At runtime, when a class definition is executed a class object is created and then assigned to a variable of the same name in the scope enclosing the class. This class is created from a code-object representing the source code for the body of the class. To represent this the `ClassDef` class (which represents a `class` statement) subclasses `Assign`. The `Class` class, which represents the body of the class, can be accessed via the `ClassDef.getDefinedClass()`. `FunctionDef` and `Function` are handled similarly.

Here is the relevant part of the class hierarchy:

- `Stmt`
  - `Assign`
    `ClassDef`
    `FunctionDef`
- `Scope`
  - `Class`
  - `Function`

8.4.4 Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- QL language reference
- CodeQL tools

8.5 Pointer analysis and type inference in Python

At runtime, each Python expression has a value with an associated type. You can learn how an expression behaves at runtime by using type-inference classes from the standard CodeQL library.

8.5.1 The `Value` class

The `Value` class and its subclasses `FunctionValue`, `ClassValue`, and `ModuleValue` represent the values an expression may hold at runtime.

Summary

Class hierarchy for `Value`:

- Value
  - `ClassValue`
  - `FunctionValue`
  - `ModuleValue`

8.5.2 Points-to analysis and type inference

Points-to analysis, sometimes known as pointer analysis, allows us to determine which objects an expression may point to at runtime. Type inference allows us to infer what the types (classes) of an expression may be at runtime. For more information, see Pointer analysis and Type inference on Wikipedia.

The predicate `ControlFlowNode.pointsTo(...)` shows which object a control flow node may point to at runtime.

`ControlFlowNode.pointsTo(...)` has three variants:

```
predicate pointsTo(Value object)
predicate pointsTo(Value object, ControlFlowNode origin)
predicate pointsTo(Context context, Value object, ControlFlowNode origin)
```

`object` is an object that the control flow node refers to, and `origin` is where the object comes from, which is useful for displaying meaningful results.

The third form includes the `context` in which the control flow node refers to the `object`. This form can usually be ignored.

> Note
>
> `ControlFlowNode.pointsTo()` cannot find all objects that a control flow node might point to as it is impossible to be accurate *and* to find all possible values. We prefer precision (no incorrect values) over recall (finding as many values as possible). We do this so that queries based on points-to analysis have fewer false positive results and are thus more useful.

For complex data flow analyses, involving multiple stages, the `ControlFlowNode` version is more precise, but for simple use cases the `Expr` based version is easier to use. For convenience, the `Expr` class also has the same three predicates. `Expr.pointsTo(...)` also has three variants:

```
predicate pointsTo(Value object)
predicate pointsTo(Value object, AstNode origin)
predicate pointsTo(Context context, Value object, AstNode origin)
```

8.5.3 Using points-to analysis

In this example we use points-to analysis to build a more complex query. This query is included in the standard query set.

We want to find `except` blocks in a `try` statement that are in the wrong order. That is, where a more general exception type precedes a more specific one, which is a problem as the second `except` handler will never be executed.

First we can write a query to find ordered pairs of `except` blocks for a `try` statement.

Ordered except blocks in same `try` statement

```
import python

from Try t, ExceptStmt ex1, ExceptStmt ex2
where
exists(int i, int j |
```

(continues on next page)

(continued from previous page)

```
    ex1 = t.getHandler(i) and ex2 = t.getHandler(j) and i < j
)
select t, ex1, ex2
```

See this in the query console on LGTM.com. Many projects contain ordered `except` blocks in a `try` statement.

Here `ex1` and `ex2` are both `except` handlers in the `try` statement `t`. By using the indices `i` and `j` we can also ensure that `ex1` precedes `ex2`.

The results of this query need to be filtered to return only results where `ex1` is more general than `ex2`. We can use the fact that an `except` block is more general than another block if the class it handles is a superclass of the other.

More general `except` block

```
exists(ClassValue cls1, ClassValue cls2 |
    ex1.getType().pointsTo(cls1) and
    ex2.getType().pointsTo(cls2) |
    not cls1 = cls2 and
    cls1 = cls2.getASuperType()
)
```

The line:

```
ex1.getType().pointsTo(cls1)
```

ensures that `cls1` is a `ClassValue` that the `except` block would handle.

Combining the parts of the query we get this:

More general `except` block precedes more specific

```
import python

from Try t, ExceptStmt ex1, ExceptStmt ex2
where
exists(int i, int j |
    ex1 = t.getHandler(i) and ex2 = t.getHandler(j) and i < j
)
and
exists(ClassValue cls1, ClassValue cls2 |
    ex1.getType().pointsTo(cls1) and
    ex2.getType().pointsTo(cls2) |
    not cls1 = cls2 and
    cls1 = cls2.getASuperType()
)
select t, ex1, ex2
```

See this in the query console on LGTM.com. This query finds only one result in the demo projects on LGTM.com (youtube-dl). The result is also highlighted by the standard Unreachable except block query. For more information, see Unreachable except block on LGTM.com.

Note

If you want to submit a query for use in LGTM, then the format must be of the form `select element message`. For example, you might replace the `select` statement with: `select t, "Incorrect order of except blocks; more general precedes more specific"`

8.5.4 Using type inference

In this example we use type inference to determine when an object is used as a sequence in a `for` statement, but that object might not be an `"iterable"`.

First of all find what object is used in the `for` loop:

```
from For loop, Value iter
where loop.getIter().pointsTo(iter)
select loop, iter
```

Then we need to determine if the object `iter` is iterable. We can test `ClassValue` to see if it has the `__iter__` attribute.

Find non-iterable object used as a loop iterator

```
import python

from For loop, Value iter, ClassValue cls
where loop.getIter().getAFlowNode().pointsTo(iter) and
  cls = iter.getClass() and
  not exists(cls.lookup("__iter__"))
select loop, cls
```

See this in the query console on LGTM.com. Many projects use a non-iterable as a loop iterator.

Many of the results shown will have `cls` as `NoneType`. It is more informative to show where these `None` values may come from. To do this we use the final field of `pointsTo`, as follows:

Find non-iterable object used as a loop iterator 2

```
import python

from For loop, Value iter, ClassValue cls, AstNode origin
where loop.getIter().pointsTo(iter, origin) and
  cls = iter.getClass() and
  not cls.hasAttribute("__iter__")
select loop, cls, origin
```

See this in the query console on LGTM.com. This reports the same results, but with a third column showing the source of the `None` values.

8.5.5 Finding calls using call-graph analysis

The `Value` class has a method `getACall()` which allows us to find calls to a particular function (including builtin functions).

If we wish to restrict the callables to actual functions we can use the `FunctionValue` class, which is a subclass of `Value` and corresponds to function objects in Python, in much the same way as the `ClassValue` class corresponds to class objects in Python.

Returning to an example from *Functions in Python*, we wish to find calls to the `eval` function.

The original query looked this:

```
import python

from Call call, Name name
where call.getFunc() = name and name.getId() = "eval"
select call, "call to 'eval'."
```

See this in the query console on LGTM.com. Some of the demo projects on LGTM.com have calls that match this pattern.

There are two problems with this query:

- It assumes that any call to something named eval is a call to the builtin `eval` function, which may result in some false positive results.
- It assumes that `eval` cannot be referred to by any other name, which may result in some false negative results.

We can get much more accurate results using call-graph analysis. First, we can precisely identify the `FunctionValue` for the `eval` function, by using the `Value::named` predicate as follows:

```
import python

from Value eval
where eval = Value::named("eval")
select eval
```

Then we can use `Value.getACall()` to identify calls to the `eval` function, as follows:

```
import python

from ControlFlowNode call, Value eval
where eval = Value::named("eval") and
      call = eval.getACall()
select call, "call to 'eval'."
```

See this in the query console on LGTM.com. This accurately identifies calls to the builtin `eval` function even when they are referred to using an alternative name. Any false positive results with calls to other `eval` functions, reported by the original query, have been eliminated.

8.5.6 Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- QL language reference
- CodeQL tools

8.6 Analyzing control flow in Python

You can write CodeQL queries to explore the control-flow graph of a Python program, for example, to discover unreachable code or mutually exclusive blocks of code.

8.6.1 About analyzing control flow

To analyze the control-flow graph of a `Scope` we can use the two CodeQL classes `ControlFlowNode` and `BasicBlock`. These classes allow you to ask such questions as can you reach point A from point B? or Is it possible to reach point B *without* going through point A?. To report results we use the class `AstNode`, which represents a syntactic element and corresponds to the source code - allowing the results of the query to be more easily understood. For more information, see Control-flow graph on Wikipedia.

8.6.2 The `ControlFlowNode` class

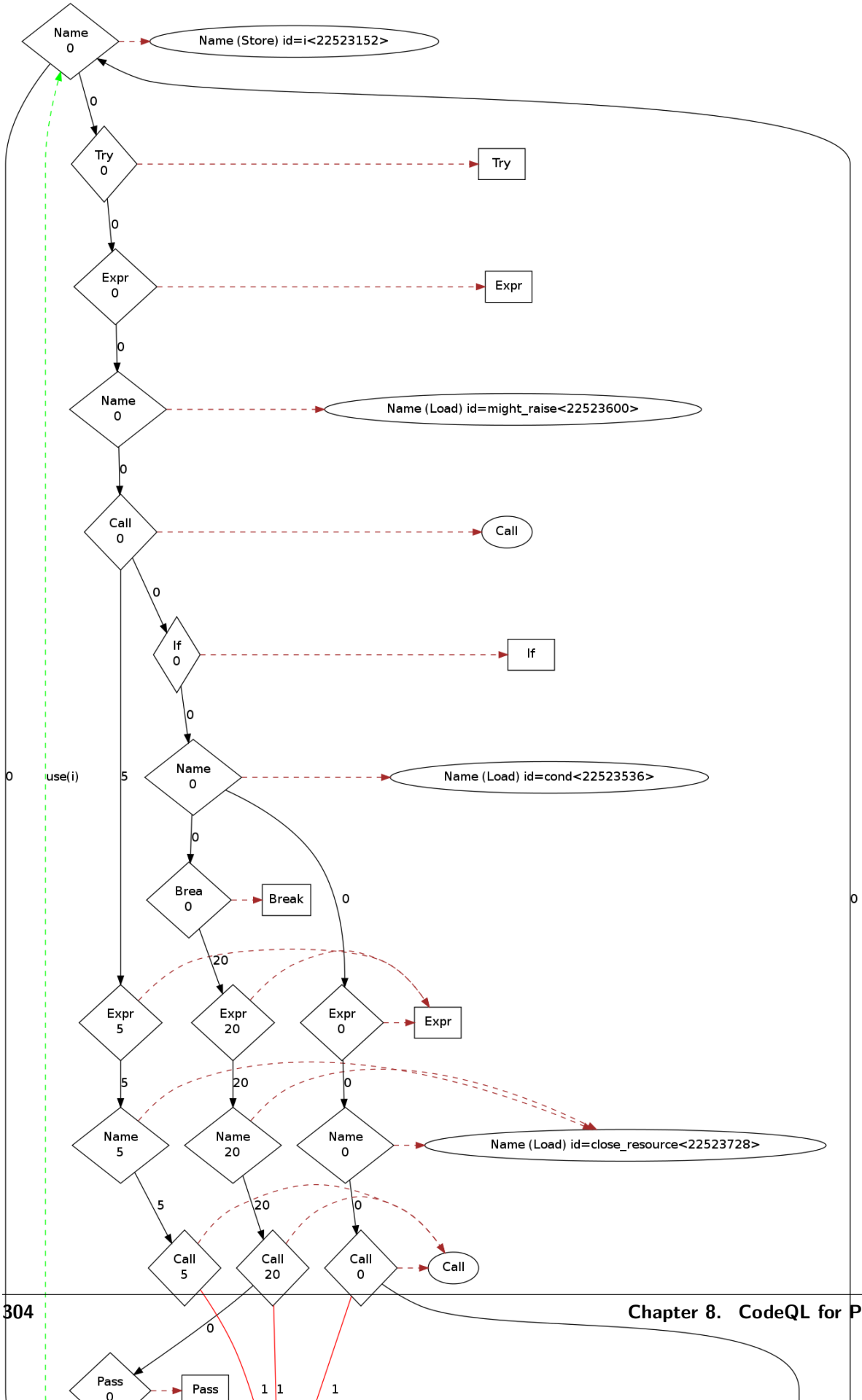
The `ControlFlowNode` class represents nodes in the control flow graph. There is a one-to-many relation between AST nodes and control flow nodes. Each syntactic element, the `AstNode`, maps to zero, one, or many `ControlFlowNode` classes, but each `ControlFlowNode` maps to exactly one `AstNode`.

To show why this complex relation is required consider the following Python code:

```
try:
    might_raise()
    if cond:
        break
finally:
    close_resource()
```

There are many paths through the above code. There are three different paths through the call to `close_resource()`; one normal path, one path that breaks out of the loop, and one path where an exception is raised by `might_raise()`.

An annotated flow graph:


Name
0
Name (Store) id=i<22523152>
0
Try
0
Try
0
Expr
0
Expr
0
Name
0
Name (Load) id=might_raise<22523600>
0
Call
0
Call
0
If
0
If
0
0
use(i)
5
Name
0
Name (Load) id=cond<22523536>
0
Brea
0
Break
0
0
20
Expr
5
Expr
20
Expr
0
Expr
5
20
0
Name
5
Name
20
Name
0
Name (Load) id=close_resource<22523728>
5
20
0
Call
5
Call
20
Call
0
Call
0
Pass
0
Pass
1
1
1

The simplest use of the `ControlFlowNode` and `AstNode` classes is to find unreachable code. There is one `ControlFlowNode` per path through any `AstNode` and any `AstNode` that is unreachable has no paths flowing through it. Therefore, any `AstNode` without a corresponding `ControlFlowNode` is unreachable.

Example finding unreachable AST nodes

```
import python

from AstNode node
where not exists(node.getAFlowNode())
select node
```

See this in the query console on LGTM.com. The demo projects on LGTM.com all have some code that has no control flow node, and is therefore unreachable. However, since the `Module` class is also a subclass of the `AstNode` class, the query also finds any modules implemented in C or with no source code. Therefore, it is better to find all unreachable statements.

Example finding unreachable statements

```
import python

from Stmt s
where not exists(s.getAFlowNode())
select s
```

See this in the query console on LGTM.com. This query gives fewer results, but most of the projects have some unreachable nodes. These are also highlighted by the standard Unreachable code query. For more information, see Unreachable code on LGTM.com.

8.6.3 The `BasicBlock` class

The `BasicBlock` class represents a basic block of control flow nodes. The `BasicBlock` class is not that useful for writing queries directly, but is very useful for building complex analyses, such as data flow. The reason it is useful is that it shares many of the interesting properties of control flow nodes, such as, what can reach what, and what dominates what, but there are fewer basic blocks than control flow nodes - resulting in queries that are faster and use less memory. For more information, see Basic block and Dominator on Wikipedia.

Example finding mutually exclusive basic blocks

Suppose we have the following Python code:

```
if condition():
    return 0
pass
```

Can we determine that it is impossible to reach both the `return 0` statement and the `pass` statement in a single execution of this code? For two basic blocks to be mutually exclusive it must be impossible to reach either of them from the other. We can write:

```
import python

from BasicBlock b1, BasicBlock b2
where b1 != b2 and not b1.strictlyReaches(b2) and not b2.strictlyReaches(b1)
select b1, b2
```

However, by that definition, two basic blocks are mutually exclusive if they are in different scopes. To make the results more useful, we require that both basic blocks can be reached from the same function entry point:

```
exists(Function shared, BasicBlock entry |
    entry.contains(shared.getEntryNode()) and
    entry.strictlyReaches(b1) and entry.strictlyReaches(b2)
)
```

Combining these conditions we get:

Example finding mutually exclusive blocks within the same function

```
import python

from BasicBlock b1, BasicBlock b2
where b1 != b2 and not b1.strictlyReaches(b2) and not b2.strictlyReaches(b1) and
exists(Function shared, BasicBlock entry |
    entry.contains(shared.getEntryNode()) and
    entry.strictlyReaches(b1) and entry.strictlyReaches(b2)
)
select b1, b2
```

See this in the query console on LGTM.com. This typically gives a very large number of results, because it is a common occurrence in normal control flow. It is, however, an example of the sort of control-flow analysis that is possible. Control-flow analyses such as this are an important aid to data flow analysis. For more information, see *Analyzing data flow and tracking tainted data in Python*.

8.6.4 Further reading

- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- QL language reference
- CodeQL tools

8.7 Analyzing data flow and tracking tainted data in Python

You can use CodeQL to track the flow of data through a Python program. Tracking user-controlled, or tainted, data is a key technique for security researchers.

8.7.1 About data flow and taint tracking

Taint tracking is used to analyze how potentially insecure, or tainted data flows throughout a program at runtime. You can use taint tracking to find out whether user-controlled input can be used in a malicious way, whether dangerous arguments are passed to vulnerable functions, and whether confidential or sensitive data can leak. You can also use it to track invalid, insecure, or untrusted data in other analyses.

Taint tracking differs from basic data flow in that it considers non-value-preserving steps in addition to normal data flow steps. For example, in the assignment `dir = path + "/"`, if `path` is tainted then `dir` is also tainted, even though there is no data flow from `path` to `path + "/"`.

Separate CodeQL libraries have been written to handle normal data flow and taint tracking in *C/C++*, *C#*, *Java*, and *JavaScript*. You can access the appropriate classes and predicates that reason about these different modes of data flow by importing the appropriate library in your query. In Python analysis, we can use the same taint tracking library to model both normal data flow and taint flow, but we are still able make the distinction between steps that preserve values and those that dont by defining additional data flow properties.

For further information on data flow and taint tracking with CodeQL, see *Introduction to data flow*.

Fundamentals of taint tracking using data flow analysis

The taint tracking library is in the TaintTracking module. Any taint tracking or data flow analysis query has three explicit components, one of which is optional, and an implicit component. The explicit components are:

1. One or more `sources` of potentially insecure or unsafe data, represented by the TaintTracking::Source class.
2. One or more `sinks`, to where the data or taint may flow, represented by the TaintTracking::Sink class.
3. Zero or more `sanitizers`, represented by the Sanitizer class.

A taint tracking or data flow query gives results when there is the flow of data from a source to a sink, which is not blocked by a sanitizer.

These three components are bound together using a TaintTracking::Configuration. The purpose of the configuration is to specify exactly which sources and sinks are relevant to the specific query.

The final, implicit component is the kind of taint, represented by the TaintKind class. The kind of taint determines which non-value-preserving steps are possible, in addition to value-preserving steps that are built into the analysis. In the above example `dir = path + "/"`, taint flows from `path` to `dir` if the taint represents a string, but not if the taint is `None`.

Limitations

Although taint tracking is a powerful technique, it is worth noting that it depends on the underlying data flow graphs. Creating a data flow graph that is both accurate and covers a large enough part of a program is a challenge, especially for a dynamic language like Python. The call graph might be incomplete, the reachability of code is an approximation, and certain constructs, like `eval`, are just too dynamic to analyze.

8.7.2 Using taint-tracking for Python

A simple taint tracking query has the basic form:

```
/**
 * @name ...
 * @description ...
```

(continues on next page)

(continued from previous page)

```
 * @kind problem
 */

import semmle.python.security.TaintTracking

class MyConfiguration extends TaintTracking::Configuration {

    MyConfiguration() { this = "My example configuration" }

    override predicate isSource(TaintTracking::Source src) { ... }

    override predicate isSink(TaintTracking::Sink sink) { ... }

    /* optionally */
    override predicate isExtension(Extension extension) { ... }

}

from MyConfiguration config, TaintTracking::Source src, TaintTracking::Sink sink
where config.hasFlow(src, sink)
select sink, "Alert message, including reference to $@.", src, "string describing the source"
```

Example

As a contrived example, here is a query that looks for flow from a HTTP request to a function called `"unsafe"`. The sources are predefined and accessed by importing library `semmle.python.web.HttpRequest`. The sink is defined by using a custom `TaintTracking::Sink` class.

```
/* Import the string taint kind needed by our custom sink */
import semmle.python.security.strings.Untrusted

/* Sources */
import semmle.python.web.HttpRequest

/* Sink */
/** A class representing any argument in a call to a function called "unsafe" */
class UnsafeSink extends TaintTracking::Sink {

    UnsafeSink() {
        exists(FunctionValue unsafe |
            unsafe.getName() = "unsafe" and
            unsafe.getACall().(CallNode).getAnArg() = this
        )
    }

    override predicate sinks(TaintKind kind) {
        kind instanceof StringKind
    }

}
```

(continues on next page)

(continued from previous page)

```
class HttpToUnsafeConfiguration extends TaintTracking::Configuration {

    HttpToUnsafeConfiguration() {
        this = "Example config finding flow from http request to 'unsafe' function"
    }

    override predicate isSource(TaintTracking::Source src) { src instanceof HttpRequestTaintSource␣
↪}

    override predicate isSink(TaintTracking::Sink sink) { sink instanceof UnsafeSink }

}

from HttpToUnsafeConfiguration config, TaintTracking::Source src, TaintTracking::Sink sink
where config.hasFlow(src, sink)
select sink, "This argument to 'unsafe' depends on $@.", src, "a user-provided value"
```

Converting a taint-tracking query to a path query

Although the taint tracking query above tells which sources flow to which sinks, it doesnt tell us how. For that we need a path query.

A standard taint tracking query can be converted to a path query by changing `@kind problem` to `@kind path-problem`, adding an import and changing the format of the query clauses. The import is simply:

```
import semmle.python.security.Paths
```

And the format of the query becomes:

```
from Configuration config, TaintedPathSource src, TaintedPathSink sink
where config.hasFlowPath(src, sink)
select sink.getSink(), src, sink, "Alert message, including reference to $@.", src.getSource(),
↪"string describing the source"
```

Thus, our example query becomes:

```
/**
 * ...
 * @kind path-problem
 * ...
 */

/* This computes the paths */
import semmle.python.security.Paths

/* Expose the string taint kinds needed by our custom sink */
import semmle.python.security.strings.Untrusted

/* Sources */
```

(continues on next page)

(continued from previous page)

```
import semmle.python.web.HttpRequest

/* Sink */
/** A class representing any argument in a call to a function called "unsafe" */
class UnsafeSink extends TaintTracking::Sink {

    UnsafeSink() {
        exists(FunctionValue unsafe |
            unsafe.getName() = "unsafe" and
            unsafe.getACall().(CallNode).getAnArg() = this
        )
    }

    override predicate sinks(TaintKind kind) {
        kind instanceof StringKind
    }

}

class HttpToUnsafeConfiguration extends TaintTracking::Configuration {

    HttpToUnsafeConfiguration() {
        this = "Example config finding flow from http request to 'unsafe' function"
    }

    override predicate isSource(TaintTracking::Source src) { src instanceof HttpRequestTaintSource␣
↪}

    override predicate isSink(TaintTracking::Sink sink) { sink instanceof UnsafeSink }

}

from HttpToUnsafeConfiguration config, TaintedPathSource src, TaintedPathSink sink
where config.hasFlowPath(src, sink)
select sink.getSink(), src, sink, "This argument to 'unsafe' depends on $@.", src.getSource(), "a␣
↪user-provided value"
```

8.7.3 Tracking custom taint kinds and flows

In the above examples, we have assumed the existence of a suitable `TaintKind`, but sometimes it is necessary to model the flow of other objects, such as database connections, or `None`.

The `TaintTracking::Source` and `TaintTracking::Sink` classes have predicates that determine which kind of taint the source and sink model, respectively.

```
abstract class Source {
    abstract predicate isSourceOf(TaintKind kind);
    ...
}

abstract class Sink {
```

(continues on next page)

(continued from previous page)

```
    abstract predicate sinks(TaintKind taint);
    ...
}
```

The `TaintKind` itself is just a string (a QL string, not a CodeQL entity representing a Python string), which provides methods to extend flow and allow the kind of taint to change along the path. The `TaintKind` class has many predicates allowing flow to be modified. This simplest `TaintKind` does not override any predicates, meaning that it only flows as opaque data. An example of this is the Hard-coded credentials query, which defines the simplest possible taint kind class, `HardcodedValue`, and custom source and sink classes. For more information, see Hard-coded credentials on LGTM.com.

```
class HardcodedValue extends TaintKind {
    HardcodedValue() {
        this = "hard coded value"
    }
}

class HardcodedValueSource extends TaintTracking::Source {
    ...
    override predicate isSourceOf(TaintKind kind) {
        kind instanceof HardcodedValue
    }
}

class CredentialSink extends TaintTracking::Sink {
    ...
    override predicate sinks(TaintKind kind) {
        kind instanceof HardcodedValue
    }
}
```

8.7.4 Further reading

- Exploring data flow with path queries
- CodeQL queries for Python
- Example queries for Python
- CodeQL library reference for Python
- QL language reference
- CodeQL tools
- *Basic query for Python code*: Learn to write and run a simple CodeQL query using LGTM.
- *CodeQL library for Python*: When you need to analyze a Python program, you can make use of the large collection of classes in the CodeQL library for Python.
- *Functions in Python*: You can use syntactic classes from the standard CodeQL library to find Python functions and identify calls to them.

- *Expressions and statements in Python*: You can use syntactic classes from the CodeQL library to explore how Python expressions and statements are used in a codebase.
- *Analyzing control flow in Python*: You can write CodeQL queries to explore the control-flow graph of a Python program, for example, to discover unreachable code or mutually exclusive blocks of code.
- *Pointer analysis and type inference in Python*: At runtime, each Python expression has a value with an associated type. You can learn how an expression behaves at runtime by using type-inference classes from the standard CodeQL library.
- *Analyzing data flow and tracking tainted data in Python*: You can use CodeQL to track the flow of data through a Python program. Tracking user-controlled, or tainted, data is a key technique for security researchers.

CHAPTER

NINE

CODEQL TRAINING AND VARIANT ANALYSIS EXAMPLES

9.1 CodeQL and variant analysis

Variant analysis is the process of using a known vulnerability as a seed to find similar problems in your code. Security engineers typically perform variant analysis to identify possible vulnerabilities and to ensure that these threats are properly fixed across multiple code bases.

CodeQL is the code analysis engine that underpins LGTM, the community driven security analysis platform. Together, CodeQL and LGTM provide continuous monitoring and scalable variant analysis for your projects, even if you dont have your own team of dedicated security engineers. You can read more about using CodeQL and LGTM in variant analysis on the Security Lab research page.

CodeQL is easy to learn, and exploring code using CodeQL is the most efficient way to perform variant analysis.

9.2 Learning CodeQL for variant analysis

Start learning how to use CodeQL in variant analysis for a specific language by looking at the topics below. Each topic links to a short presentation on CodeQL, its libraries, or an example variant discovered using CodeQL.

When you have selected a presentation, use → and to navigate between slides. Press **p** to view the additional notes on slides that have an information icon in the top right corner, and press **f** to enter full-screen mode.

The presentations contain a number of query examples. We recommend that you download CodeQL for Visual Studio Code and add the example database for each presentation so that you can find the bugs mentioned in the slides.

> Information
>
> The presentations listed below are used in CodeQL and variant analysis training sessions run by GitHub engineers. Therefore, be aware that the slides are designed to be presented by an instructor. If you are using the slides without an instructor, please use the additional notes to help guide you through the examples.

9.2.1 CodeQL and variant analysis for C/C++

- Introduction to variant analysis: CodeQL for C/C++–an introduction to variant analysis and CodeQL for C/C++ programmers.
- Example: Bad overflow guard–an example of iterative query development to find bad overflow guards in a C++ project.

- Program representation: CodeQL for C/C++–information on how CodeQL analysis represents C/C++ programs.
- Introduction to local data flow–an introduction to analyzing local data flow in C/C++ using CodeQL, including an example demonstrating how to develop a query to find a real CVE.
- Exercise: snprintf overflow–an example demonstrating how to develop a data flow query.
- Introduction to global data flow–an introduction to analyzing global data flow in C/C++ using CodeQL.
- Analyzing control flow: CodeQL for C/C++–an introduction to analyzing control flow in C/C++ using CodeQL.

9.2.2 CodeQL and variant analysis for Java

- Introduction to variant analysis: CodeQL for Java–an introduction to variant analysis and CodeQL for Java programmers.
- Example: Query injection–an example of iterative query development to find unsanitized SPARQL injections in a Java project.
- Program representation: CodeQL for Java–information on how CodeQL analysis represents Java programs.
- Introduction to local data flow–an introduction to analyzing local data flow in Java using CodeQL, including an example demonstrating how to develop a query to find a real CVE.
- Exercise: Apache Struts–an example demonstrating how to develop a data flow query.
- Introduction to global data flow–an introduction to analyzing global data flow in Java using CodeQL.

9.2.3 Further reading

- GitHub Security Lab

CHAPTER

TEN

RECENT TERMINOLOGY CHANGES

We recently started using new terminology to make it clearer to users what our products do. This note gives some information about what has changed.

10.1 CodeQL

CodeQL is the code analysis platform formerly known as QL. CodeQL treats code as data, and CodeQL analysis is based on running queries against your code to check for errors and find bugs and vulnerabilities. The CodeQL product includes the tools, scripts, queries, and libraries used in CodeQL analysis.

10.2 QL

Previously we used the term QL to refer to the whole code analysis platform, which has been renamed CodeQL. The name QL now only refers to the query language that powers CodeQL analysis.

The CodeQL queries and libraries used to analyze source code are written in QL. These queries and libraries are open source, and can be found in the CodeQL repository. QL is a general-purpose, object-oriented language that can be used to query any kind of data.

10.3 CodeQL databases

QL snapshots have been renamed CodeQL databases. CodeQL databases contain relational data created and analyzed using CodeQL. They are the equivalent of QL snapshots, but have been optimized for use with the CodeQL tools.

CHAPTER

ELEVEN

FURTHER READING

- QL language reference: A description of important concepts in QL and a formal specification of the QL language.