

# Static Program Analysis For Security

Cambridge IB Tech Talks

Zayne Zhang

zz513@cam.ac.uk

March 12, 2025

# Overview

---

1. Introduction to Static Program Analysis
2. Lattices and Fixed Points
3. Data Flow Analysis
4. CodeQL

# How to find bugs in code?

---

## Dynamic Analysis (e.g. Unit Testing, Fuzzing)

- Done on a limited number of different inputs
- Often reveals the presence of errors but **cannot guarantee their absence**
- 100% test coverage  $\neq$  bug-free code
- Many tests are regressive and only added after a bug is found

## Static Analysis: analyze code without executing it

- Can check all possible executions and provide guarantees about its behavior
- With the right tools, can catch bugs early in the development process
- Particularly useful for testing the absence of security vulnerabilities

# How to find bugs in code?

---



**Brenan Keller**  
@brenankeller



A QA engineer walks into a bar.  
Orders a beer. Orders 0 beers.  
Orders 999999999999 beers.  
Orders a lizard. Orders -1 beers.  
Orders a ueicbksjdhd.

First real customer walks in  
and asks where the bathroom  
is. The bar bursts into flames,  
killing everyone.

1:21 PM · 30 Nov 18

# Data Flow Analysis

---

A particular form of static analysis that **examines how data moves through a program** to answer questions such as:

- What values can reach this point in the code?
- Is this variable always initialized before it is used?
- **Does untrusted data ever reach an unsafe function?**

# Partial Orders

## Definition

A **partial order**  $(S, \sqsubseteq)$  is a set  $S$  equipped with a binary relation  $\sqsubseteq$  that is:

- **Reflexive:**  $\forall x \in S, x \sqsubseteq x$
  - **Transitive:**  $\forall x, y, z \in S, x \sqsubseteq y \wedge y \sqsubseteq z \Rightarrow x \sqsubseteq z$
  - **Antisymmetric:**  $\forall x, y \in S, x \sqsubseteq y \wedge y \sqsubseteq x \Rightarrow x = y$
- 
- $y \in S$  is an upper bound for  $X$  ( $X \sqsubseteq y$ ) if  $\forall x \in X, x \sqsubseteq y$
  - $y \in S$  is the least upper bound for  $X$  ( $X \sqcup y$ ) if  $y$  is an upper bound for  $X$  and  $\forall z \in S, X \sqsubseteq z \Rightarrow y \sqsubseteq z$
  - $y \in S$  is a lower bound for  $X$  ( $y \sqsubseteq X$ ) if  $\forall x \in X, y \sqsubseteq x$
  - $y \in S$  is the greatest lower bound for  $X$  ( $y \sqcap X$ ) if  $y$  is a lower bound for  $X$  and  $\forall z \in S, z \sqsubseteq X \Rightarrow z \sqsubseteq y$

# Lattices

---

## Definition

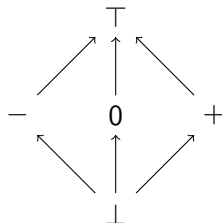
A **lattice**  $(L, \sqsubseteq)$  is a partial order  $(L, \sqsubseteq)$  in which every pair of elements  $x, y \in L$  has a least upper bound  $x \sqcup y$  (join) and a greatest lower bound  $x \sqcap y$  (meet).

A **complete lattice** is a lattice in which every subset has a least upper bound and a greatest lower bound.

# Sign Analysis

As an example, we want to find the possible signs of integer variables and expressions. Consider the following abstract values for the sign of an integer:

- $\top$ : unknown sign
- $+$ : positive
- $-$ : negative
- $0$ : zero
- $\perp$ : not an integer, or unreachable code



This partial order, with edges for  $\sqsubseteq$ , forms a complete lattice. e.g.  $+\sqsubseteq\top$  means  $+$  is *at least as precise as*  $\top$



# Sign Analysis

Let's create a **map lattice**  $State = Var \rightarrow Sign$  that describes the sign of each variable. Derive a system of equations, one per line, using values from the lattice.

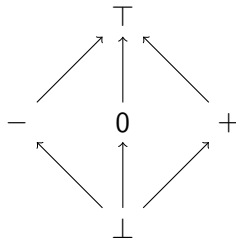
```
var a, b; // 1
a = 42; // 2
b = a + input(); // 3
a = a - b; // 4
```

$$x_1 = [a \mapsto \top, b \mapsto \top]$$

$$x_2 = x_1[a \mapsto +]$$

$$x_3 = x_2[b \mapsto x_2(a) + \top]$$

$$x_4 = x_3[a \mapsto x_3(a) - x_3(b)]$$



*Sign*

# Sign Analysis

---

$$x_1 = [a \mapsto \top, b \mapsto \top]$$

$$x_2 = x_1[a \mapsto +]$$

$$x_3 = x_2[b \mapsto x_2(a) + \top]$$

$$x_4 = x_3[a \mapsto x_3(a) - x_3(b)]$$

$$f_1(x_1, \dots, x_n) = [a \mapsto \top, b \mapsto \top]$$

$$f_2(x_1, \dots, x_n) = x_1[a \mapsto +]$$

$$f_3(x_1, \dots, x_n) = x_2[b \mapsto x_2(a) + \top]$$

$$f_4(x_1, \dots, x_n) = x_3[a \mapsto x_3(a) - x_3(b)]$$

Generalised equation system over a lattice  $L$ , with functions  $f_i : L^n \rightarrow L$ :

$$x_1 = f_1(x_1, \dots, x_n)$$

$$x_2 = f_2(x_1, \dots, x_n)$$

$$\vdots$$

$$x_n = f_n(x_1, \dots, x_n)$$

# Monotonicity and Fixed Points

---

Generalised equation system over a lattice  $L$ , with functions  $f_i : L^n \rightarrow L$ :

$$x_1 = f_1(x_1, \dots, x_n)$$

$$x_2 = f_2(x_1, \dots, x_n)$$

$$\vdots$$

$$x_n = f_n(x_1, \dots, x_n)$$

Combine the  $n$  functions into  $F : L^n \rightarrow L^n$ :

$$\begin{aligned} F(x_1, \dots, x_n) &= (f_1(x_1, \dots, x_n), \dots, f_n(x_1, \dots, x_n)) \\ &= (x_1, \dots, x_n) \end{aligned}$$

Then we are looking for  $x = F(x)$ , i.e. a fixed point of  $F$ .

# Monotonicity and Fixed Points

## Definition

A function  $f : L_1 \rightarrow L_2$  is **monotone** if  $\forall x, y \in L_1, x \sqsubseteq y \Rightarrow f(x) \sqsubseteq f(y)$

*More precise input leads to more precise output*

## Theorem

**Kleene's Fixed Point Theorem:** *In a complete lattice  $L$  with finite height, every monotone function  $f : L \rightarrow L$  has a unique least fixed point  $\bigsqcup_{i=0}^{\infty} f^i(\perp)$*

These results generalise to functions that take multiple arguments  $f : L^n \rightarrow L$  that are monotone in each argument – such as the ones we derived for sign analysis.

## Corollary

*For an equation system over complete lattices of finite height with monotone constraint functions, a **unique, most precise solution** always exists*

# Computing the Least Fixed Point

---

---

**Algorithm 1** Naive Fixed Point Algorithm

---

```
1: procedure NAIVEFIXEDPOINT( $F$ )  
2:    $x := \perp$   
3:   while  $x \neq F(x)$  do  
4:      $x := F(x)$   
5:   end while  
6:   return  $x$   
7: end procedure
```

---

In each iteration, all of  $f_1, \dots, f_4$  are applied. But  $f_2$  depends only on  $x_1$ , and the value of  $x_1$  is unchanged in most iterations. We'll see a more efficient way later.

# Data Flow Analysis

---

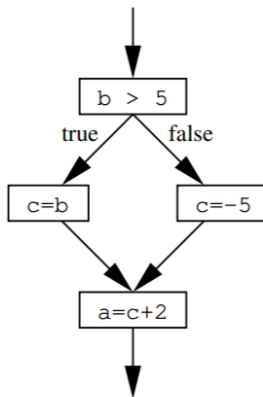
Main idea: we want to **find the possible values of variables at each point in the program.**

- In compilers: used for optimisations (e.g. constant propagation)
- In security: used to find vulnerabilities (e.g. untrusted data reaching a sink)

# Control Flow Graph

---

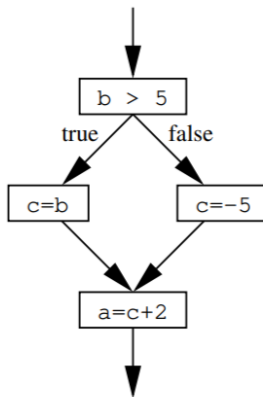
- In our previous example, we had a sequence of statements with no branches
- In general, we have a **control flow graph (CFG)** with basic blocks and edges



# Abstract States

---

- Recall: each element of the lattice  $State = Var \rightarrow Sign$  is an abstract state that maps variables to signs
- For each CFG node  $v$ , let the **constraint variable**  $[[v]]$  be the abstract state at the program point *immediately after*  $v$
- We have a lattice  $State^n$  of abstract states, where  $n$  is the number of CFG nodes

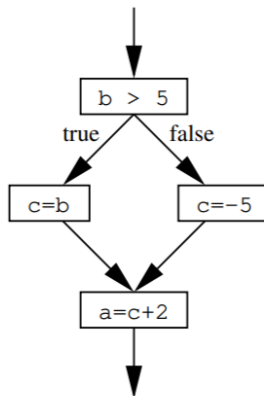




# Constraint Rules

We need to combine the abstract states of the predecessors of a node to get the abstract state of the node itself.

$$\text{JOIN}(v) = \bigsqcup_{u \in \text{pred}(v)} [[u]]$$



$$\text{JOIN}([[a = c + 2]]) = [[c = b]] \sqcup [[c = -5]]$$

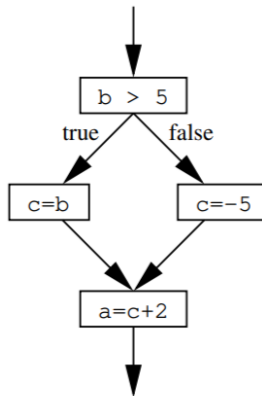
# Constraint Rules

$$[[c = b]] = [b \mapsto +, c \mapsto +]$$

$$[[c = -5]] = [b \mapsto \top, c \mapsto -]$$

$$\begin{aligned} \text{JOIN}([a = c + 2]) &= [[c = b]] \sqcup [[c = -5]] \\ &= [b \mapsto \top, c \mapsto \top] \end{aligned}$$

$$\begin{aligned} [[a = c + 2]] &= \text{JOIN}([a = c + 2])[a \mapsto \text{eval}(\text{JOIN}([a = c + 2]), c + 2)] \\ &= [a \mapsto \top, b \mapsto \top, c \mapsto \top] \end{aligned}$$



# Solving Data Flow Equations

---

Generalised equation system over a lattice  $L$ , with functions  $f_i : L^n \rightarrow L$ :

$$\begin{aligned} [[v_1]] &= f_{v_1}([v_1], \dots, [v_n]) \\ [[v_2]] &= f_{v_2}([v_1], \dots, [v_n]) \\ &\vdots \\ [[v_n]] &= f_{v_n}([v_1], \dots, [v_n]) \end{aligned}$$

Combine the  $n$  functions into  $F : L^n \rightarrow L^n$ :

$$\begin{aligned} F([v_1], \dots, [v_n]) &= (f_{v_1}([v_1], \dots, [v_n]), \dots, f_{v_n}([v_1], \dots, [v_n])) \\ &= ([v_1], \dots, [v_n]) \end{aligned}$$

Then we are looking for  $x = F(x)$ , i.e. a fixed point of  $F$ .

# A More Efficient Algorithm

---

## Algorithm 2 Simple Worklist Algorithm

---

```
1: procedure SIMPLEWORKLIST( $F$ )
2:    $(x_1, \dots, x_n) := (\perp, \dots, \perp)$ 
3:    $W := \{v_1, \dots, v_n\}$ 
4:   while  $W \neq \emptyset$  do
5:      $v_i := W.\text{pop}()$ 
6:      $y := f_{v_i}(x_1, \dots, x_n)$ 
7:     if  $y \neq x_i$  then
8:        $x_i := y$ 
9:       for  $v_j \in \text{dep}(v_i)$  do
10:         $W.\text{add}(v_j)$ 
11:       end for
12:     end if
13:   end while
14:   return  $(x_1, \dots, x_n)$ 
15: end procedure
```

---

Insight: most  $f_{v_i}$  will only read the values from a few other variables, instead of all  $[[v_1]], \dots, [[v_n]]$ .

$\text{dep}(v_i)$  is the set of nodes that depend on  $v_i$  (i.e. the successors of  $v_i$ )

# Taint Tracking

---

- **Can we tell which computations may involve “tainted” data?**
- i.e. data that comes from an untrusted source
- e.g. user input, HTTP responses, environment variables

# Taint Tracking

---

The approach is similar! We just define different abstract values and equations.

- **Abstract taint values:**
  - $\top$ : Unknown taint status.
  - $T$ : Tainted.
  - $U$ : Untainted.
  - $\perp$ : Unreachable.
- **Ordering:**  $\perp \sqsubseteq U \sqsubseteq T \sqsubseteq \top$
- **Abstract state:** A mapping  $\sigma : \text{Var} \rightarrow \{\perp, U, T, \top\}$ .
- **Transfer functions:** e.g. for an assignment  $x := y \text{ op } z$ , define

$$f(\sigma) = \sigma[x \mapsto \sigma(y) \sqcup \sigma(z)]$$

i.e. if either operand is tainted, the result is tainted

# In Practice: CodeQL

---

A tool developed by Semmle (a spin-out company from Oxf\*rd), now acquired by GitHub. Used with CLI or GitHub integration (free for all public repos!)

- The source code is compiled into a relational database, which includes information about the control flow graph, data flow, and other properties of the code.
- The user writes queries in a high-level language called QL, which is executed by the CodeQL engine.
- The engine uses fixed-point algorithms to perform data flow analysis.
- Results are exported into the SARIF format which can be consumed by CI tools or custom integrations.

# Let's Go On A Little Adventure

---

- Next.js, the most popular React framework, has some weird, poorly documented URL parsing semantics that does not conform to the widely accepted WHATWG URL standard
- This is unexpected behaviour, and often results in wrong URL validation
- Made a responsible disclosure  $\approx$  1 year ago, still not fixed

Let's query open-source GitHub projects to find instances of this bug!

- Common design pattern: unauthenticated user visits `/admin`, gets redirected to `/login?next=/admin`, logs in, and gets redirected back to `/admin`
- Use Next.js URL parsing trickery to turn a “normal” URL into `javascript:sendToAttacker(authToken)` at the final step



# Taint Tracking in CodeQL

We want to find all instances where untrusted user input (source) reaches a sensitive function (sink) without being sanitized.

○ ○ ○

```
1 import javascript
2
3 class UnsafeRouterPushConfiguration extends TaintTracking::Configuration {
4   UnsafeRouterPushConfiguration() { this = "UnsafeRouterPushConfiguration" }
5
6   override predicate isSource(DataFlow::Node source) {}
7
8   override predicate isSink(DataFlow::Node sink) {}
9 }
10
11 from DataFlow::PathNode source, DataFlow::PathNode sink, UnsafeRouterPushConfiguration config
12 where config.hasFlowPath(source, sink)
13 select sink.getNode(), "Potentially unsafe router.push with $@.", source.getNode(),
14   "untrusted input"
15
```

# Defining Sources

---

○ ○ ○

```
1 override predicate isSource(DataFlow::Node source) {  
2   source instanceof RemoteFlowSource or    // user input, e.g. https://example.com?x=1  
3   source instanceof ClientRequest::Range  // results of HTTP requests  
4 }
```

You can also extend this with custom logic, to incorporate codebase-specific patterns e.g. RPC calls, deserialization, etc.

# Defining Sinks

○ ○ ○

```
1 override predicate isSink(DataFlow::Node sink) {  
2   exists(DataFlow::MethodCallNode call, DataFlow::Node receiver |  
3     call.getMethodName() = "push" and  
4     call.getReceiver() = receiver and  
5     receiver.getLocalSource().(DataFlow::InvokeNode).getCalleeName() = "useRouter" and  
6     sink = call.getArgument(0)  
7   }  
8 }
```

$\text{isSink}(\text{node}) \triangleq \exists \text{ call, receiver . call invokes the } \mathbf{.push} \text{ method on receiver} \wedge$   
 $\exists \text{ invocation . invocation is } \mathbf{useRouter()} \wedge \text{invocation} \rightarrow^* \text{receiver} \wedge$   
 $\text{node} = \mathbf{args}(\text{call})[0]$

# Changing the Transfer Function

---

○ ○ ○

```
1  override predicate isAdditionalTaintStep(DataFlow::Node pred, DataFlow::Node succ) {
2      exists(DataFlow::ArrayCreationNode array |
3          pred = array.getAnElement() and
4          succ = array
5      )
6      or
7      exists(DataFlow::MethodCallNode call |
8          call.getMethodName().regexMatch("find|filter|some|every|map") and
9          pred = call.getReceiver() and
10         succ = call.getABoundCallbackParameter(1, 0)
11     )
12 }
```

# Changing the Transfer Function

---

○ ○ ○

```
1 override predicate isSanitizer(DataFlow::Node node) {  
2   node = DataFlow::moduleImport("dompurify").getAMemberCall("sanitize")  
3 }
```

Any results from DOMPurify.sanitize are treated as untainted. Know your assumptions!

# Let's Go Hunting

The screenshot shows the grep search interface with the query `useRouter()` entered in the top right. The search has found 35,298 results. On the left, a sidebar lists repositories with their respective result counts. The main area displays two code snippets where `useRouter()` is used.

**Repository List:**

- supabase/supabase (222)
- neptune-mutual-blue/app.nept... (172)
- civitai/civitai (155)
- pancakeswap/pancake-frontend (150)
- TRADLY-PLATFORM/Butterflies (146)
- labring/sealos (141)
- RocketChat/Rocket.Chat (136)
- rnadigital/agentcloud (124)

**Search Results:**

35,298 results found

**campsite/campsite** apps/web/components/NavigationBar/index.tsx

```
124 export function SignedOutNavigationBar() {
125   const { asPath } = useRouter()
126   const isDesktop = useIsDesktopApp()
127   const router = useRouter()
128   const { data: post } = useGetPost({ postId: router.query?.postId as string })
```

**github/docs** src/landings/components/SidebarProduct.tsx

```
10 export const SidebarProduct = () => {
11   const router = useRouter()
12   const {
```

# Let's Go Hunting

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

🟢 CodeQL query - javascript query on 20 repositories #12

Re-run all jobs

⋮

Summary

Jobs

🟢 setup

🟢 run (supabase/realtime)

🟢 run (supabase/supabase)

🟢 run (supabase/postgres-js)

🟢 run (supabase/supabase-js)

🟢 run (supabase/realtime-js)

🟢 run (supabase/postgres-meta)

🟢 run (supabase/auth-js)

🟢 run (supabase/cli)

🟢 run (supabase/ui)

🟢 run (supabase/storage)

🟢 run (supabase/supabase-ui-w...)

🟢 run (supabase/storage-js)

🟢 run (supabase/auth-helpers)

🟢 run (supabase/supabase-actio...

🟢 run (supabase/setup-cli)

🟢 run (supabase/edge-runtime)

Triggered via dynamic 9 hours ago

zeyu2001 → fc03e35 `main`

Status

Success

Total duration

2m 31s

Artifacts

-

queries

on: dynamic

🟢 setup 0s

Matrix: run

🟢 20 jobs completed  
Show all jobs

🕒 update-repo-tasks-statuses... 0s

🕒 update-repo-tasks-statuses... 0s

🔄

-

+

# Limitations

---

CodeQL is very useful as a CI integration to catch security issues early in the development process, and provide guarantees about your code. But it's really hard to get right . . .

- We need to create custom taint specifications for third-party library APIs.
- False positives: even if tainted data reaches a sink, it may not always be exploitable – some other conditions may need to be met
- Requires a good understanding of the codebase and the problem domain, and lots of fine-tuning to get good results – only as good as the queries you write



# Alternative Approaches

---

- **Symbolic execution:** represent the program inputs symbolically and explore all possible paths through the program, generating constraints on the inputs such that a certain path is taken
- Ziyang Li, Saikat Dutta, and Mayur Naik. LLM-assisted static analysis for detecting security vulnerabilities, 2024

*IRIS leverages LLMs to infer taint specifications and perform contextual analysis, alleviating needs for human specifications and inspection . . .*

*A state-of-the-art static analysis tool CodeQL detects only 27 of these vulnerabilities whereas IRIS with GPT-4 detects 55 (+28) and improves upon CodeQL's average false discovery rate by 5% points.*

# References

---

- [1] Anders Møller and Michael I. Schwartzbach. *Static Program Analysis*. November 2020.
- [2] Oege de Moor, Mathieu Verbaere, Elnar Hajiyeu, Pavel Avgustinov, Torbjorn Ekman, Neil Ongkingco, Damien Sereni, and Julian Tibble. Keynote address: .ql for source code analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 3–16, 2007.
- [3] Ziyang Li, Saikat Dutta, and Mayur Naik. LLM-assisted static analysis for detecting security vulnerabilities, 2024.