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 != 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 99999999999 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**: $\forall x, y, z \in S, x \sqsubseteq y \land y \sqsubseteq z \Rightarrow x \sqsubseteq z$
- **Antisymmetric**: $\forall x, y \in S, x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$

- $y \in S$ 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 $(X \bigsqcup y)$ 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 $(y \bigsqcap X)$ 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
- $\bot$: not an integer, or unreachable code

$$
\begin{tikzcd}
 & \top & \\
- \arrow[ur] & 0 \arrow[u] & + \arrow[ul] \\
 & \bot \arrow[ul] \arrow[u] \arrow[ur] &
\end{tikzcd}
$$

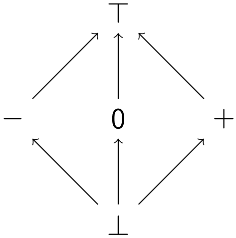
This partial order, with edges for $\sqsubseteq$, forms a complete lattice. e.g. $+ \sqsubseteq \top$ means $+$ is *at least as precise* as $\top$

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
```

$$
\begin{aligned}
x_1 &= [a \mapsto \top, b \mapsto \top] \\
x_2 &= x_1[a \mapsto +] \\
x_3 &= x_2[b \mapsto x_2(a) + \top] \\
x_4 &= x_3[a \mapsto x_3(a) - x_3(b)]
\end{aligned}
$$



Sign

Sign Analysis

$$\begin{aligned} x_1 &= [a \mapsto \top, b \mapsto \top] \\ x_2 &= x_1[a \mapsto +] \\ x_3 &= x_2[b \mapsto x_2(a) + \top] \\ x_4 &= x_3[a \mapsto x_3(a) - x_3(b)] \end{aligned} \qquad \begin{aligned} f_1(x_1, \ldots, x_n) &= [a \mapsto \top, b \mapsto \top] \\ f_2(x_1, \ldots, x_n) &= x_1[a \mapsto +] \\ f_3(x_1, \ldots, x_n) &= x_2[b \mapsto x_2(a) + \top] \\ f_4(x_1, \ldots, x_n) &= x_3[a \mapsto x_3(a) - x_3(b)] \end{aligned}$$

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

$$\begin{aligned} x_1 &= f_1(x_1, \ldots, x_n) \\ x_2 &= f_2(x_1, \ldots, x_n) \\ &\vdots \\ x_n &= f_n(x_1, \ldots, x_n) \end{aligned}$$

Monotonicity and Fixed Points

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

$$
\begin{aligned}
x_1 &= f_1(x_1, \ldots, x_n) \\
x_2 &= f_2(x_1, \ldots, x_n) \\
&\vdots \\
x_n &= f_n(x_1, \ldots, x_n)
\end{aligned}
$$

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

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

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

Monotonicity and Fixed Points

Definition

A function $f : L_1 \to L_2$ 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 $f : L \to L$ has a unique least fixed point $\bigsqcup_{i=0}^{\infty} f^i(\bot)$*

These results generalise to functions that take multiple arguments $f : L^n \to L$ 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

procedure $\textsc{NaiveFixedPoint}(F)$
    $x := \bot$
    while $x \neq F(x)$ **do**
        $x := F(x)$
    end while
    return x
end procedure

In each iteraction, all of $f_1, \ldots, f_4$ 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.

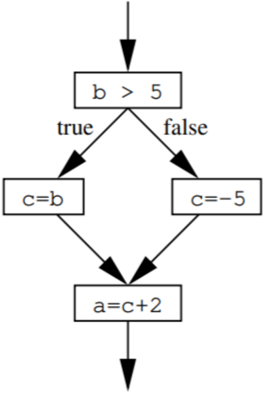
Data Flow Analysis

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)

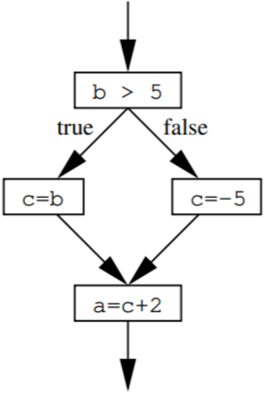
Control Flow Graph

- 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

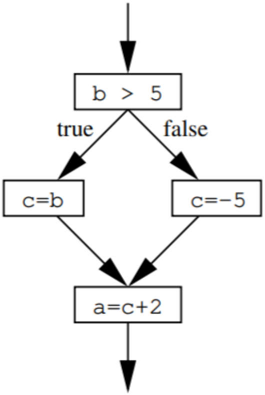
- Recall: each element of the lattice $State = Var \to Sign$ is an abstract state that maps variables to signs
- For each CFG node v, let the **constraint variable** $[[v]]$ be the abstract state at the program point *immediately after* v
- We have a lattice $State^n$ of abstract states, where n is the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

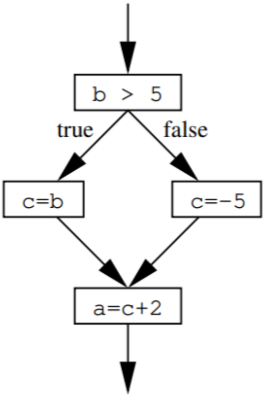
$$\mathrm{JOIN}(v) = \bigsqcup_{u \in \mathrm{pred}(v)} [[u]]$$



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

Constraint Rules

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



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

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

Solving Data Flow Equations

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

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

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

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

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

A More Efficient Algorithm

Algorithm 2 Simple Worklist Algorithm

```
procedure SimpleWorkList(F)
    (x_1, ..., x_n) := (⊥, ..., ⊥)
    W := {v_1, ..., v_n}
    while W ≠ ∅ do
        v_i := W.pop()
        y := f_{v_i}(x_1, ..., x_n)
        if y ≠ x_i then
            x_i := y
            for v_j ∈ dep(v_i) do
                W.add(v_j)
            end for
        end if
    end while
    return (x_1, ..., x_n)
end procedure
```

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

$dep(v_i)$ 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.
  - $\bot$: Unreachable.
- **Ordering:** $\bot \sqsubseteq U \sqsubseteq T \sqsubseteq \top$
- **Abstract state:** A mapping $\sigma : \mathrm{Var} \to \{\bot, U, T, \top\}$.
- **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 ≈ 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.

```
import javascript

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

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

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

from DataFlow::PathNode source, DataFlow::PathNode sink, UnsafeRouterPushConfiguration config
where config.hasFlowPath(source, sink)
select sink.getNode(), "Potentially unsafe router.push with $@.", source.getNode(),
  "untrusted input"

```

Defining Sources

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

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

Defining Sinks

```
override predicate isSink(DataFlow::Node sink) {
  exists(DataFlow::MethodCallNode call, DataFlow::Node receiver |
    call.getMethodName() = "push" and
    call.getReceiver() = receiver and
    receiver.getALocalSource().(DataFlow::InvokeNode).getCalleeName() = "useRouter" and
    sink = call.getArgument(0)
  )
}
```

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

Changing the Transfer Function

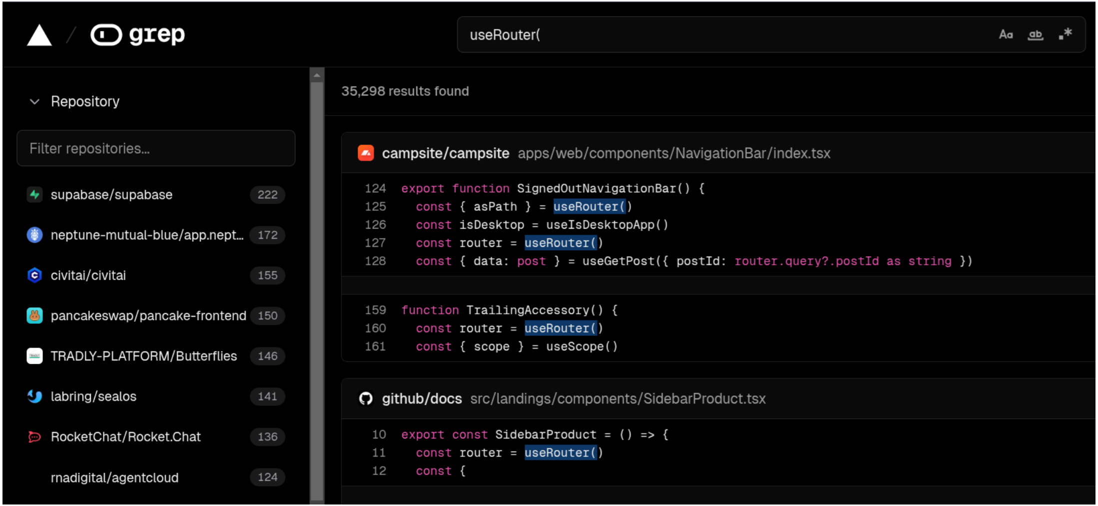
```
override predicate isAdditionalTaintStep(DataFlow::Node pred, DataFlow::Node succ) {
  exists(DataFlow::ArrayCreationNode array |
    pred = array.getAnElement() and
    succ = array
  )
  or
  exists(DataFlow::MethodCallNode call |
    call.getMethodName().regexpMatch("find|filter|some|every|map") and
    pred = call.getReceiver() and
    succ = call.getABoundCallbackParameter(1, 0)
  )
}
```

Changing the Transfer Function

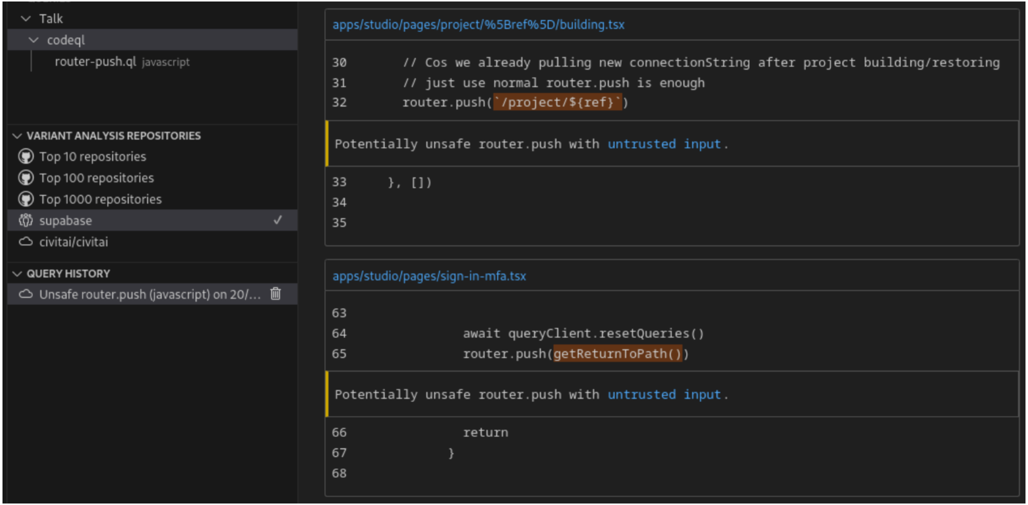
```
override predicate isSanitizer(DataFlow::Node node) {
  node = DataFlow::moduleImport("dompurify").getAMemberCall("sanitize")
}
```

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

Let’s Go Hunting



Let’s Go Hunting



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 Hajiyev, 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.