Taint Analysis via CFL-Reachability

Mostafa Hassanein

03, January 2026

Agenda

- What is taint analysis?

Agenda

- What is taint analysis?
- Why is it important?

Agenda

- What is taint analysis?
- Why is it important?
- Why context sensitivity is needed?

Agenda

- What is taint analysis?
- Why is it important?
- Why context sensitivity is needed?
- CFL-reachability formulation

Agenda

- What is taint analysis?
- Why is it important?
- Why context sensitivity is needed?
- CFL-reachability formulation
- Examples

Static Analysis

- Static analysis reasons about *all executions* without running the program

Static Analysis

- Static analysis reasons about *all executions* without running the program
- It computes a sound over-approximation of behaviors

Static Analysis

- Static analysis reasons about *all executions* without running the program
- It computes a sound over-approximation of behaviors
- Taint analysis is a data-flow–based static analysis

What Is Taint Analysis?

- Tracks flow of untrusted (tainted) data

What Is Taint Analysis?

- Tracks flow of untrusted (tainted) data
- Detects whether tainted data reaches sensitive operations

What Is Taint Analysis?

- Tracks flow of untrusted (tainted) data
- Detects whether tainted data reaches sensitive operations
- Abstracts away concrete values, tracking dependencies

What Is Taint Analysis?

- Tracks flow of untrusted (tainted) data
- Detects whether tainted data reaches sensitive operations
- Abstracts away concrete values, tracking dependencies

Core Question:

Does tainted data flow from a source to a sink along a valid execution?

Example: SQL Query Template

Intended Query

```
 SELECT balance
FROM AcctData
WHERE name = ’:n’ AND password = ’:p’
```

Malicious User Input

$n =$ ‘‘Charles Dickens’ --’’ $p =$ ‘‘who cares’’

Malicious User Input

$n =$ `‘‘Charles Dickens’ --’’` $p =$ `‘‘who cares’’`

Resulting Query

```
 SELECT balance FROM AcctData
WHERE name = ’Charles Dickens’ --’
AND password = ’who cares’
```

Malicious User Input

$n =$ ‘‘Charles Dickens’ --’’ $p =$ ‘‘who cares’’

Resulting Query

```
 SELECT balance FROM AcctData
WHERE name = ’Charles Dickens’ --’
AND password = ’who cares’
```

- Password check is commented out

Malicious User Input

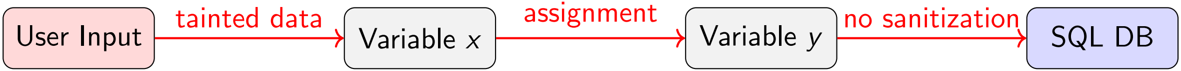
$n =$ `‘‘Charles Dickens’ --’’` $p =$ `‘‘who cares’’`

Resulting Query

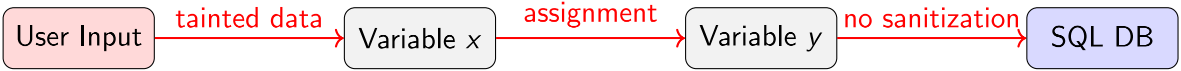
```
 SELECT balance FROM AcctData
WHERE name = ’Charles Dickens’ --’
AND password = ’who cares’
```

- Password check is commented out
- Sensitive data is leaked

Taint Propagation Graph

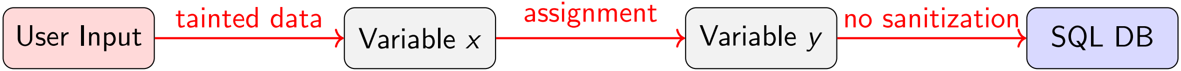


Taint Propagation Graph



- Nodes represent program entities (variables or functions)

Taint Propagation Graph



- Nodes represent program entities (variables or functions)
- Edges represent labeled data-flow relations

Why Context Sensitivity Is Needed

- The same function may be called from different sites

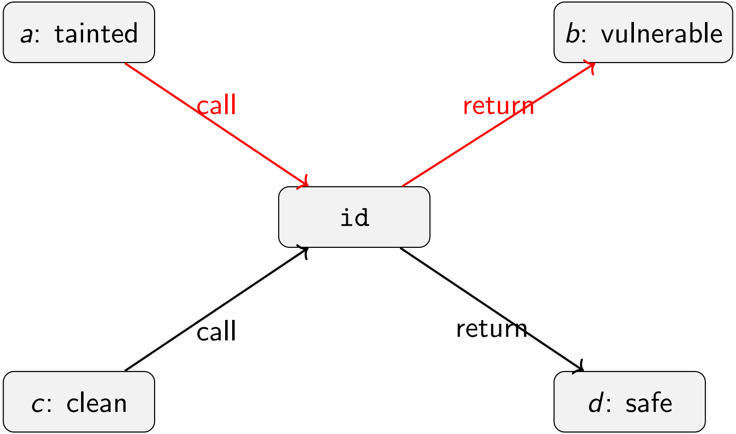
Why Context Sensitivity Is Needed

- The same function may be called from different sites
- Taint behavior depends on the calling context

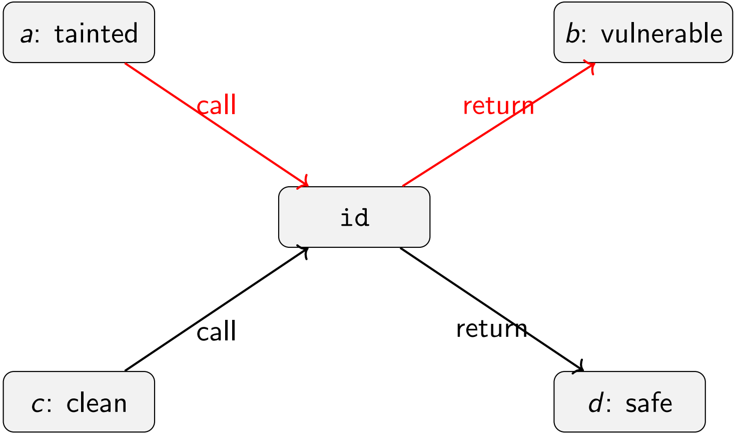
Why Context Sensitivity Is Needed

- The same function may be called from different sites
- Taint behavior depends on the calling context
- Context-insensitive analysis merges incompatible flows

Context-Insensitive Call Graph



Context-Insensitive Call Graph



Spurious taint flow appears due to context merging.

Invalid Reachability Path

- Clean call enters function

Invalid Reachability Path

- Clean call enters function
- Tainted return exits function

Invalid Reachability Path

- Clean call enters function
- Tainted return exits function
- Path exists in graph but not in execution

Invalid Reachability Path

- Clean call enters function
- Tainted return exits function
- Path exists in graph but not in execution

$$\text{call}_{\text{clean}} \;\; \text{return}_{\text{tainted}} \quad \text{(invalid)}$$

Program Graph Model

A program is modeled as a labeled graph

$$G = (V, E, \Sigma)$$

Program Graph Model

A program is modeled as a labeled graph

$$G = (V, E, \Sigma)$$

- V: program entities

Program Graph Model

A program is modeled as a labeled graph

$$G = (V, E, \Sigma)$$

- V: program entities
- $E \subseteq V \times \Sigma \times V$: labeled edges

Program Graph Model

A program is modeled as a labeled graph

$$G = (V, E, \Sigma)$$

- V: program entities
- $E \subseteq V \times \Sigma \times V$: labeled edges
- Σ: flow actions (call, return, assign)

Paths and Labels

A path

$$\pi = v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} v_k$$

Paths and Labels

A path

$$\pi = v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} v_k$$

induces a word

$$\ell(\pi) = a_1 a_2 \cdots a_k$$

Paths and Labels

A path

$$\pi = v_0 \xrightarrow{a_1} v_1 \xrightarrow{a_2} \cdots \xrightarrow{a_k} v_k$$

induces a word

$$\ell(\pi) = a_1 a_2 \cdots a_k$$

Only some words correspond to valid executions.

Grammar for Valid Taint Flows

$$F \;\rightarrow\; FF \;\mid\; call_i \; F \; return_i \;\mid\; assign \;\mid\; \varepsilon$$

Grammar for Valid Taint Flows

$$F \rightarrow FF \mid call_i\ F\ return_i \mid assign \mid \varepsilon$$

- $call_i/return_i$: Procedure invocation and return matching call site i

Grammar for Valid Taint Flows

$$F \rightarrow FF \mid call_i\ F\ return_i \mid assign \mid \varepsilon$$

- $call_i / return_i$: Procedure invocation and return matching call site i
- `assign`: Local data flow within a procedure

Grammar for Valid Taint Flows

$$F \;\rightarrow\; FF \;\mid\; \mathit{call}_i \; F \; \mathit{return}_i \;\mid\; \mathit{assign} \;\mid\; \varepsilon$$

- $\mathit{call}_i / \mathit{return}_i$: Procedure invocation and return matching call site i
- `assign`: Local data flow within a procedure
- **Result:** Only matching pairs $(\mathit{call}_i, \mathit{return}_i)$ are derivable. This filters out paths that do not correspond to feasible call stacks.

Taint Analysis as CFL-Reachability

Problem

Given $s, t \in V$, does there exist a path π from s to t such that

$$\ell(\pi) \in L(\mathcal{G})?$$

Taint Analysis as CFL-Reachability

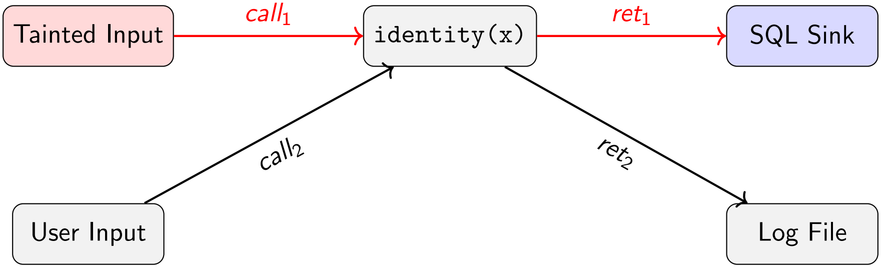
Problem

Given $s, t \in V$, does there exist a path π from s to t such that

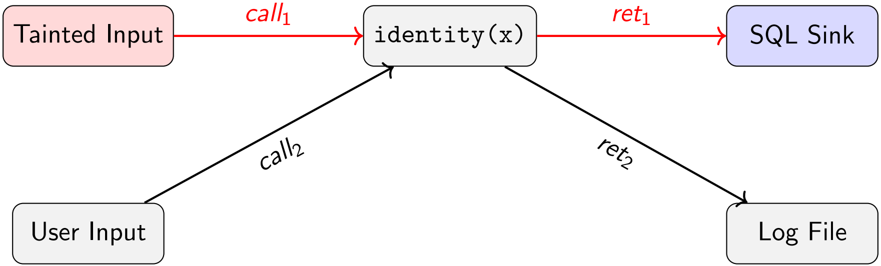
$$\ell(\pi) \in L(\mathcal{G})?$$

This exactly captures context-sensitive taint analysis.

Example 1: The Identity Crisis (Context Matching)

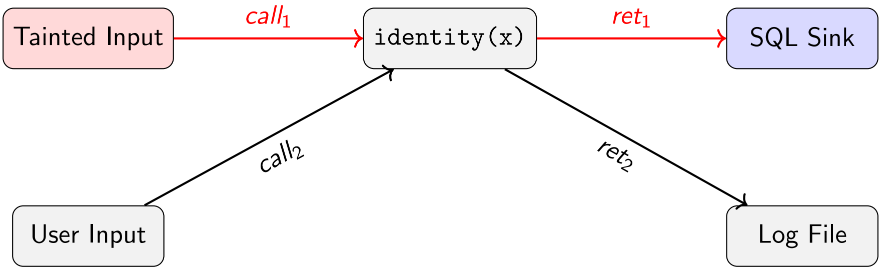


Example 1: The Identity Crisis (Context Matching)



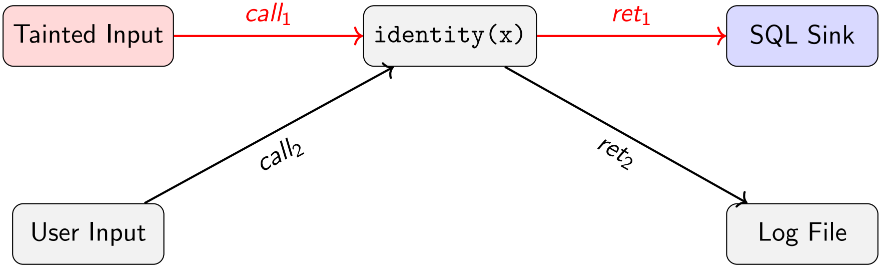
- **Graph Path:** A path exists from *Tainted Input* to *Log File* via $call_1 \rightarrow ret_2$.

Example 1: The Identity Crisis (Context Matching)



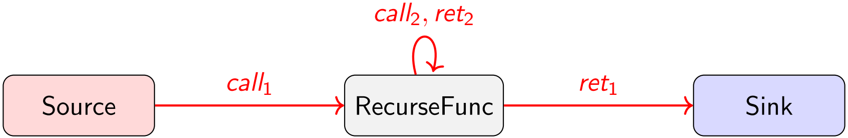
- **Graph Path:** A path exists from *Tainted Input* to *Log File* via $call_1 \to ret_2$.
- **CFL Check:** $call_1 ret_2$ is **not** in $L(\mathcal{G})$ because indices 1 and 2 don't match.

Example 1: The Identity Crisis (Context Matching)

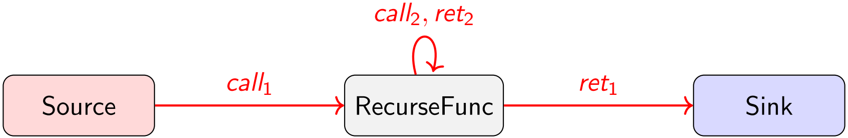


- **Graph Path:** A path exists from *Tainted Input* to *Log File* via $call_1 \rightarrow ret_2$.
- **CFL Check:** $call_1 ret_2$ is **not** in $L(\mathcal{G})$ because indices 1 and 2 don't match.
- **Insight:** Grammar prevents tainted data from "leaking" into different call sites.

Example 2: Deep Recursion (Self-Referential Flow)

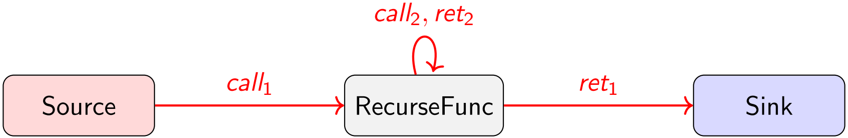


Example 2: Deep Recursion (Self-Referential Flow)



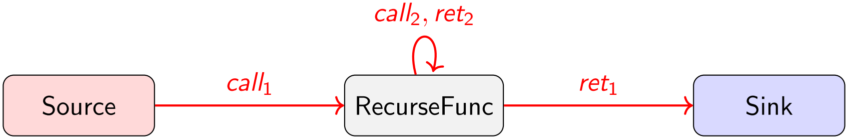
- **Scenario:** A function calls itself. Taint flows through an arbitrary number of recursive steps.

Example 2: Deep Recursion (Self-Referential Flow)



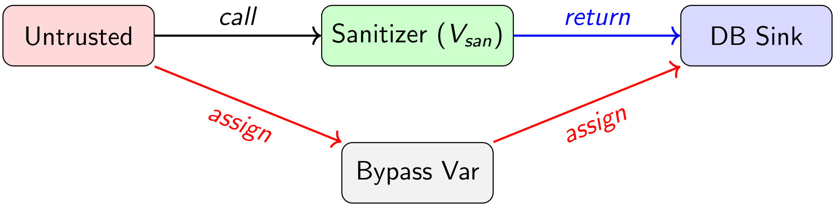
- **Scenario:** A function calls itself. Taint flows through an arbitrary number of recursive steps.
- **Word:** $\ell(\pi) = call_1(call_2)^n(ret_2)^n ret_1$.

Example 2: Deep Recursion (Self-Referential Flow)



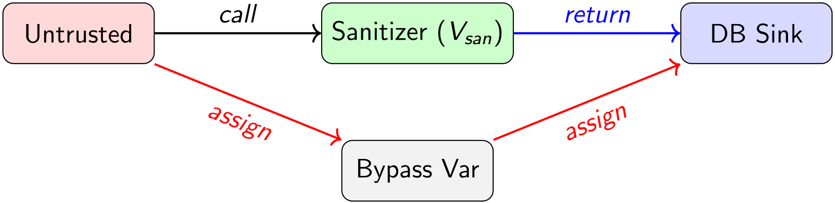
- **Scenario:** A function calls itself. Taint flows through an arbitrary number of recursive steps.
- **Word:** $\ell(\pi) = call_1(call_2)^n(ret_2)^n ret_1$.
- **CFL Validation:** This is a classic $a^n b^n$ structure. The grammar $F \rightarrow call_i F ret_i$ naturally accepts balanced recursive calls, ensuring the data eventually returns to the correct caller.

Example 3: Sanitization Logic (Path Selection)



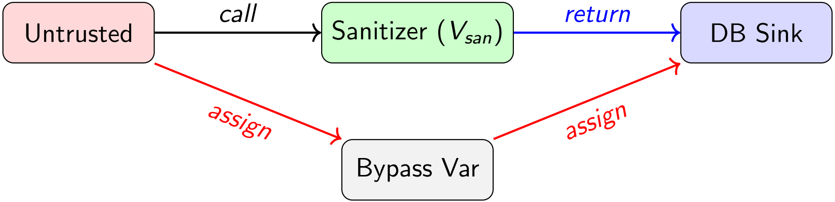
- **Formal Condition:** A path π is a violation if $\ell(\pi) \in L(\mathcal{G})$ and $V(\pi) \cap V_{san} = \emptyset$.

Example 3: Sanitization Logic (Path Selection)



- **Formal Condition:** A path π is a violation if $\ell(\pi) \in L(\mathcal{G})$ and $V(\pi) \cap V_{san} = \emptyset$.
- **Sanitized Path:** $\pi_{top} =$ (Untrusted $\rightarrow$ Sanitizer $\rightarrow$ Sink). Since Sanitizer $\in V_{san}$, this path is **discarded** despite being in $L(\mathcal{G})$.

Example 3: Sanitization Logic (Path Selection)



- **Formal Condition:** A path π is a violation if $\ell(\pi) \in L(\mathcal{G})$ and $V(\pi) \cap V_{san} = \emptyset$.
- **Sanitized Path:** $\pi_{top} = (\text{Untrusted} \to \text{Sanitizer} \to \text{Sink})$. Since Sanitizer $\in V_{san}$, this path is **discarded** despite being in $L(\mathcal{G})$.
- **Vulnerable Path:** $\pi_{bot} = (\text{Untrusted} \to \text{Bypass} \to \text{Sink})$. $\ell(\pi_{bot}) = assign \cdot assign \in L(\mathcal{G})$ and avoids V_{san}. **Violation detected.**

Summary

- Taint analysis tracks information flow

Summary

- Taint analysis tracks information flow
- Context insensitivity yields false positives

Summary

- Taint analysis tracks information flow
- Context insensitivity yields false positives
- Valid executions form a context-free language

Summary

- Taint analysis tracks information flow
- Context insensitivity yields false positives
- Valid executions form a context-free language
- CFL-reachability provides principled context sensitivity

References

- Aho, Alfred V., et al. Compilers: Principles, Techniques, and Tools. 2nd ed., Pearson Education, 2006.

Thank you

Questions?