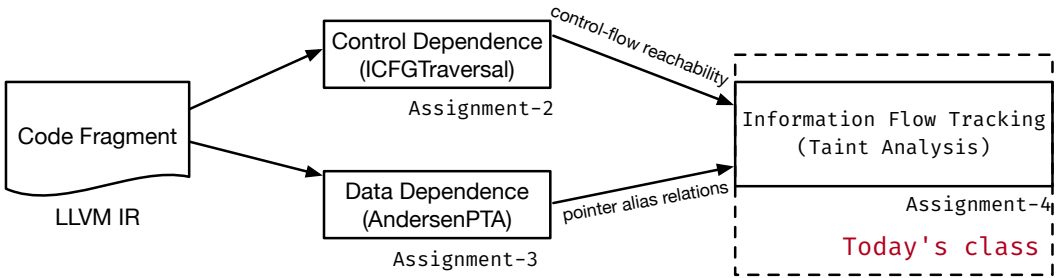
Information Flow Tracking

Yulei Sui

University of Technology Sydney, Australia

Today's Class



What is Taint Analysis?

- Taint analysis aims to reason about the control and data dependence from a source (statement/node) to a sink (statement/node).
- Taint analysis can also be seen as information flow tracking analysis.
  - Static taint analysis: taint tracking at compile time (**this subject**)
  - Dynamic taint analysis: taint tracking during runtime.

What is Taint Analysis?

- Taint analysis aims to reason about the control and data dependence from a source (statement/node) to a sink (statement/node).
- Taint analysis can also be seen as information flow tracking analysis.
  - Static taint analysis: taint tracking at compile time (**this subject**)
  - Dynamic taint analysis: taint tracking during runtime.

Why learn Taint Analysis?

- Detect information leakage
  - sensitive data stored in a heap object and manipulated by pointers can be passed around and stored to an unchecked memory (untrusted third-party APIs)
- Detect code vulnerability
  - There is a vulnerability if an unchecked tainted **source** (e.g., return value from an untrusted third party function) flows into one of the following **sinks**, where the tainted variable being used as
    - a parameter passed to a sensitive function or
    - a bound access (array index) or
    - a termination condition (loop condition)
    - . . .

How to Perform Static Taint Analysis?

Let us use what we have learned about control- and data-dependence to develop an information flow checker to validate tainted flows from a source to a sink.

- A **source** $\mathbf{v_{src}@s_{src}}$ is a tuple consisting of a variable $\mathbf{v_{src}}$ and a statement $\mathbf{s_{src}}$ where $\mathbf{v_{src}}$ is defined.
- A **sink** $\mathbf{v_{snk}@s_{snk}}$ is also a tuple consisting of a variable $\mathbf{v_{snk}}$ and a statement $\mathbf{s_{snk}}$ where $\mathbf{v_{snk}}$ is used.
- In SVF, variables $\mathbf{v_{src}}$ and $\mathbf{v_{snk}}$ are `PAGNodes`. Statements $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are `ICFGNodes`.

How to Perform Static Taint Analysis?

Let us use what we have learned about control- and data-dependence to develop an information flow checker to validate tainted flows from a source to a sink.

- A **source** $\mathbf{v_{src}@s_{src}}$ is a tuple consisting of a variable $\mathbf{v_{src}}$ and a statement $\mathbf{s_{src}}$ where $\mathbf{v_{src}}$ is defined.
- A **sink** $\mathbf{v_{snk}@s_{snk}}$ is also a tuple consisting of a variable $\mathbf{v_{snk}}$ and a statement $\mathbf{s_{snk}}$ where $\mathbf{v_{snk}}$ is used.
- In SVF, variables $\mathbf{v_{src}}$ and $\mathbf{v_{snk}}$ are `PAGNodes`. Statements $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are `ICFGNodes`.
- Given a **tainted** source $\mathbf{v_{src}@s_{src}}$, we say that a sink $\mathbf{v_{snk}@s_{snk}}$ is also **tainted** if both of the following two conditions satisfy:
  - (1) $\mathbf{s_{src}}$ reaches $\mathbf{s_{snk}}$ on the ICFG (**Assignment 2**), and
  - (2) $\mathbf{v_{src}}$ is aliased with $\mathbf{v_{snk}}$, i.e., $pts(v_{src}) \cap pts(v_{snk}) \neq \varnothing$ (**Assignment 3**)

Taint Analysis

Example 1

```
1  int main(){
2      char* secretToken = tgetstr();   // source
3      char* a = secretToken;
4      char* b = a;
5      broadcast(b);                    // sink
6  }
```

What is the tainted flow?

Taint Analysis

Example 1

```
1  int main(){
2      char* secretToken = tgetstr();   // source
3      char* a = secretToken;
4      char* b = a;
5      broadcast(b);                    // sink
6  }
```

What is the tainted flow?

- Line 2 reaches Line 5 along the ICFG (control-dependence holds) `secretToken` and `b` are aliases (data-dependence holds)
- Both control-dependence and data-dependence hold. Therefore, `secretToken`@Line 2 flows to `b`@Line 5.

Taint Analysis

Example 2

```
1  int main(){
2      char* secretToken = tgetstr(...);    // source
3      char* a = secretToken;
4      char* b = a;
5      char* publicToken = "hello";
6      broadcast(publicToken);          // sink
7  }
```

Do we have a tainted flow from source to sink?

Taint Analysis

Example 2

```
1  int main(){
2      char* secretToken = tgetstr(...);    // source
3      char* a = secretToken;
4      char* b = a;
5      char* publicToken = "hello";
6      broadcast(publicToken);          // sink
7  }
```

Do we have a tainted flow from source to sink?

- Line 2 reaches Line 6 along the ICFG (control-dependence holds),
- `secretToken` and `publicToken` are not aliases (data-dependence does not hold),
- `secretToken`@Line 2 does not flow to `publicToken`@Line 6.

Taint Analysis

Example 3

```
1  char* foo(char* token){ return token; }
2  int main(){
3      if(condition){
4          char*  secretToken = tgetstr(...);     // source
5          char* b = foo(secretToken);
6      }
7      else{
8          char* publicToken = "hello";
9          char* a = foo(publicToken);
10         broadcast(a);                          // sink
11     }
12 }
```

Do we have a tainted flow from source to sink?

Taint Analysis

Example 3

```
1  char* foo(char* token){ return token; }
2  int main(){
3      if(condition){
4          char*  secretToken = tgetstr(...);     // source
5          char* b = foo(secretToken);
6      }
7      else{
8          char* publicToken = "hello";
9          char* a = foo(publicToken);
10         broadcast(a);                          // sink
11     }
12 }
```

Do we have a tainted flow from source to sink?

- `secretToken` and `a` are aliases due to callee `foo` (data-dependence holds),
- Line 4 does not reach Line 10 on ICFG (control-dependence does not hold),
- `secretToken`@Line 4 does not flow to `a`@Line 10.

Taint Analysis

Example 4

```
1  int main(){
2      char* secretToken = tgetstr(...);                    // source
3      while(loopCondition){
4          if(BranchCondition){
5              char* a = secretToken;
6              broadcast(a);                                // sink
7          }
8          else{
9              char* b = "hello";
10         }
11     }
12 }
```

How many tainted flows from source to sink?

Taint Analysis

Example 4

```
1  int main(){
2      char* secretToken = tgetstr(...);                    // source
3      while(loopCondition){
4          if(BranchCondition){
5              char* a = secretToken;
6              broadcast(a);                                // sink
7          }
8          else{
9              char* b = "hello";
10         }
11     }
12 }
```

How many tainted flows from source to sink?

- (At least) two paths from Line 2 to Line 6 on ICFG (control-dependence holds),
- `secretToken` and `a` are aliases (data-dependence holds),
- `secretToken`@Line 2 has two tainted paths flowing to `a`@Line 6.

Configuring Sources and Sinks for Taint Analysis

Aim: enable different taint tracking patterns by defining/configuring sources and sinks.

- Given a source $\mathbf{v_{src}@s_{src}}$ and a sink $\mathbf{v_{snk}@s_{snk}}$, in this class, we are interested in the case that $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are both API calls, i.e., `CallBlockNode` in SVF.

Configuring Sources and Sinks for Taint Analysis

Aim: enable different taint tracking patterns by defining/configuring sources and sinks.

- Given a source $\mathbf{v_{src}@s_{src}}$ and a sink $\mathbf{v_{snk}@s_{snk}}$, in this class, we are interested in the case that $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are both API calls, i.e., `CallBlockNode` in SVF.
- $\mathbf{v_{src}}$ is a return value from the call statement $\mathbf{s_{src}}$.
- $\mathbf{v_{snk}}$ is a parameter being passed to a call statement $\mathbf{s_{snk}}$.

Configuring Sources and Sinks for Taint Analysis

Aim: enable different taint tracking patterns by defining/configuring sources and sinks.

- Given a source $\mathbf{v_{src}@s_{src}}$ and a sink $\mathbf{v_{snk}@s_{snk}}$, in this class, we are interested in the case that $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ are both API calls, i.e., `CallBlockNode` in SVF.
- $\mathbf{v_{src}}$ is a return value from the call statement $\mathbf{s_{src}}$.
- $\mathbf{v_{snk}}$ is a parameter being passed to a call statement $\mathbf{s_{snk}}$.
- We can identify $\mathbf{s_{src}}$ and $\mathbf{s_{snk}}$ according to different APIs, so as to configure sources and sinks.
- In our Example 1, variable `secretToken` is $\mathbf{v_{src}}$ and `b` is $\mathbf{v_{snk}}$. The call statement `tgetstr(..)` represents $\mathbf{s_{src}}$ and `broadcast(..)` are used for $\mathbf{s_{snk}}$.