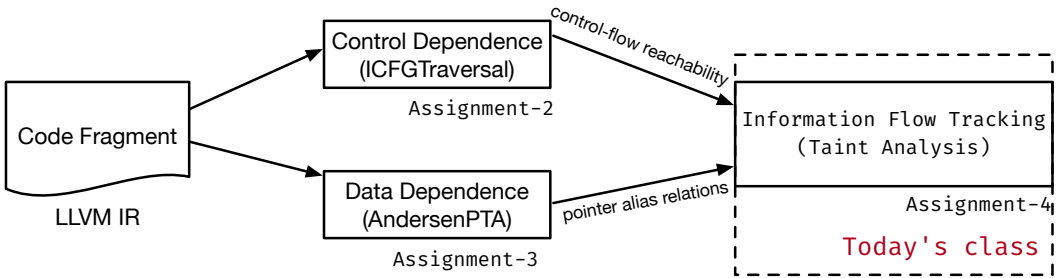
Information Flow Tracking

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
  - A 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 passing 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., $pts(src) \cap pts(snk) \neq \varnothing$ (**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 `b`@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 `b`@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 have two tainted paths flowing to `b`@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 passing 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 $\mathbf{v_{src}}@\mathbf{s_{src}}$ and a sink $\mathbf{v_{snk}}@\mathbf{s_{snk}}$, 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 passing 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 $\mathbf{s_{snk}}$.

Assignment 4

- Code template and specification:
  `https://github.com/SVF-tools/SVF-Teaching/wiki/Assignment-4`
- Make sure your previously implementations in `Assignment-2.cpp` and `Assignment-3.cpp` are in place.
  - Class `TaintGraphTraversal` in Assignment 4 is a **child class** of 'ICFGTraversal'. `TaintGraphTraversal` will use the `DFS` method implemented in Assignment 2 for **control-flow traversal**.
  - Andersen's analysis implemented in Assignment 3 will also be used for **checking aliases** between two pointers.

Assignment 4

- You will need to implement two tasks in Assignment 4 with the second one having 5 bonus points.
  - Task 1 (**Compulsory**)
    - Implement method `readSrcSnkFromFile` in `Assignment-4.cpp` using C++ file reading to configure sources and sinks.
    - Implement method `printICFGPath` to collect the tainted ICFG paths and add each path (a sequence of node IDs) as a string into `std::set<std::string> paths` similar to Assignment 2
    - Implement method `aliasCheck` to check aliases of the variables at source and sink.

Assignment 4

- You will need to implement two tasks in Assignment 4 with the second one having 5 bonus points.
  - Task 1 (**Compulsory**)
    - Implement method `readSrcSnkFromFile` in `Assignment-4.cpp` using C++ file reading to configure sources and sinks.
    - Implement method `printICFGPath` to collect the tainted ICFG paths and add each path (a sequence of node IDs) as a string into `std::set<std::string> paths` similar to Assignment 2
    - Implement method `aliasCheck` to check aliases of the variables at source and sink.
  - Task 2 (**5 bonus points**)
    - Dump the taint program paths into a text file
    - Implement a VSCode extension to annotate and visualize the tainted paths from a source to a sink.

C++ File Reading

Implement method `readSrcSnkFormFile` in `Assignment-4.cpp` to parse the two lines from `SrcSnk.txt` in the form of

```
1 line 1 contains source APIs  "{ api1, api2, api3 }"
2 line 2 contains sink APIs    "{ api1, api2, api3 }"
```

Please refer to the following links (among many others) for C++ file reading:

- https://www.tutorialspoint.com/cplusplus/cpp_files_streams.htm
- https://www.cplusplus.com/doc/tutorial/files/
- https://linuxhint.com/cplusplus_read_write/
- https://opensource.com/article/21/3/ccc-input-output

Visualizing Tainted Paths (5 bonus points)

This task is optional and there is no uniform answer! Some hits as below but you are also encouraged to design and implement your own approach.

- Output taint paths into a text file in the following format for example, ‘{ ln: number cl: number, fl: name } → { ln: number, cl: number, fl: name } → { ln: number, cl: number, fl: name }‘.
- Create a VSCode extension to read the text file
- Annotate the target source file (e.g., example.c) based on the taint paths reading from the text file.

Two VSCode extension examples (note that they are just general examples for references and are **NOT** the solution to the task):

`https://github.com/akshatsinghkaushik/vscode-extension-example`

`https://github.com/spcidealacm/codepointer_js`

VSCode Extension Demo (You feel free to design yours)