Detection of Malicious Code Using Static Taint Analysis

This is a two-year SEI-funded project, ending in Oct 2024.

Will Klieber

Sep 2023

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM23-0929

Problem

- DoD uses a lot of software produced by various supply chains.
- These supply chains can be compromised by an adversary:
  - Network intrusion
  - Insider threat
- Failing to detect malicious code can be very costly, but detection is difficult.
  - Example: SolarWinds incident of 2020
- We aim to detect two types of malicious code:
  - Exfiltration of potentially sensitive information
  - Timebombs / logic bombs, Remote-Access Trojans, etc.
    - In general: Calling a potentially sensitive system API call (e.g., starting a new process) in response to a potentially questionable trigger (e.g., on a specific date, in response to incoming network packets, etc.)

Our approach

- Scope restriction: We will flag code as *potentially* malicious, but further human analysis is required to determine whether the code is *actually* malicious.
  - Whether behavior is malicious depends on the what the program is supposed to do.
  - Vulnerabilities (e.g., SQL injection) are outside the main focus of this project.
- Goal for our tool: Produce output that concisely and precisely characterizes the potentially malicious behaviors of the codebase, so that a human analyst can quickly and accurately determine whether the behavior is benign or malicious.

Questions for audience discussion

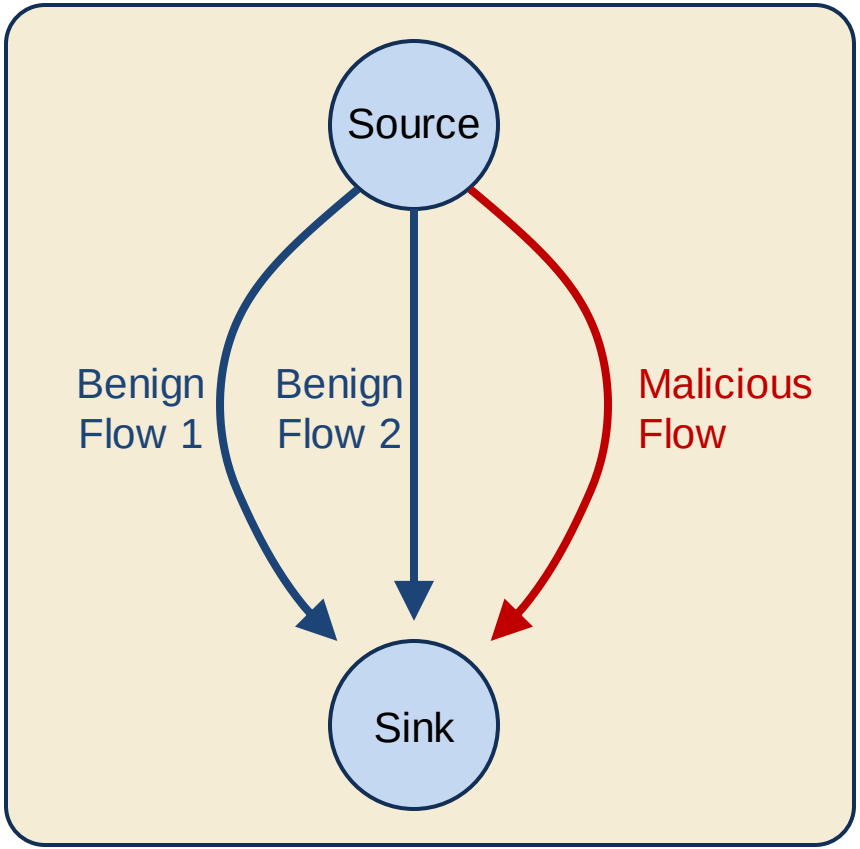
1. Are you currently involved in scanning for malicious code?
   - If so, what tools or practices do you use? How well do they work?
   - Primarily binary, primarily source-code, or a mixture?
2. What tool functionality would you find useful for detecting malicious code?
3. What information or tool functionality would you find most helpful in adjudicating whether a potentially malicious behavior is actually malicious?
4. Analyzing whole codebase for the first time vs analyzing updated version?

More details of our approach

- We will be using only static analysis, not dynamic analysis.
  - So, no need to actually run potentially malicious code.
- We are building on Phasar, an open-source LLVM-based static-analysis framework: https://github.com/secure-software-engineering/phasar
- Initially, we are focusing on C/C++ codebases.
  - We will also have some support for binaries, by lifting to LLVM IR.
  - We can also fairly easily support other languages that compile to LLVM IR.

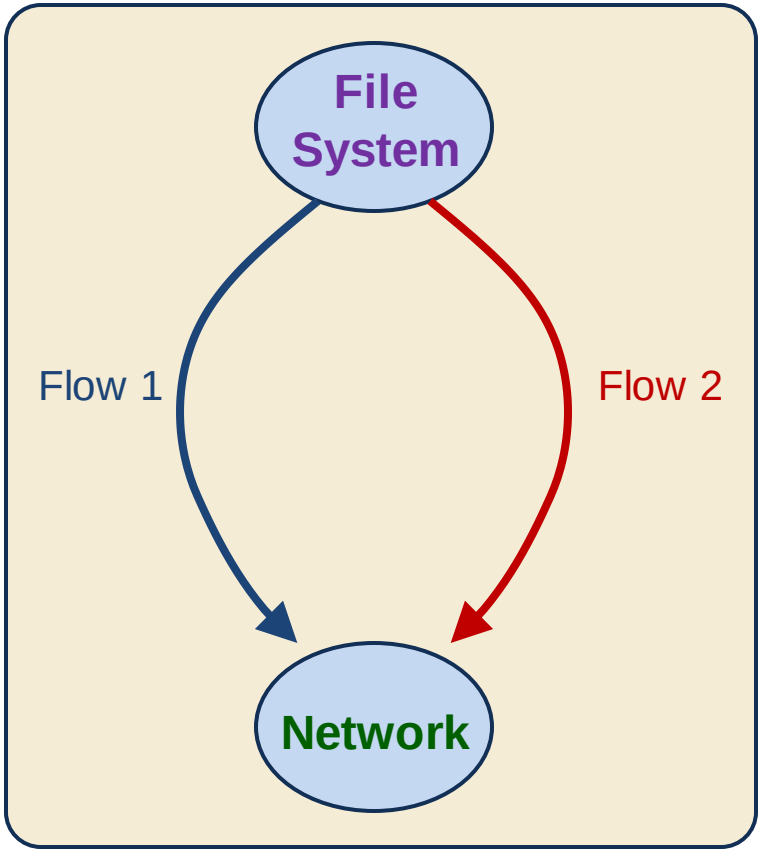
Information Flow Analysis

- Taint analysis using the Interprocedural, Finite, Distributive Subset (IFDS) algorithm
  - has successful track record, e.g., finding malicious flows of information in Android apps.
  - *Sources* are designated system API calls that return potentially sensitive information.
  - *Sinks* are designated system API calls that can be used to exfiltrate information to an attacker.
- Limitation: conflates together all flow paths from a given source to a given sink. So, a malicious flow path can be 'hidden' by a benign flow path.
- **Our idea:** Identify the conditions under which the flows happen, especially conditions that might be indicative of malicious code.



Motivating example E1 (pseudocode)

```
1.   function Flow_1() {
2.     cmd = read_from_keyboard();
3.     if (is_upload_cmd(cmd)) {
4.       name = get_file_name(cmd);
5.       x = read_from_file(name);
6.       send_to_network(x);
7.     }
8.   }
9.
10.  function Flow_2() {
11.    data = read_from_network();
12.    if (is_special_cmd(data)) {
13.      x = read_from_file("secrets.txt");
14.      send_to_network(x);
15.    }
16.  }
```



Motivating example E1 – ideal output

```
1.   function Flow_1() {
2.     cmd = read_from_keyboard();
3.     if (is_upload_cmd(cmd)) {
4.       name = get_file_name(cmd);
5.       x = read_from_file(name);
6.       send_to_network(x);
7.     }
8.   }
9.
10.  function Flow_2() {
11.    data = read_from_network();
12.    if (is_special_cmd(data)) {
13.      x = read_from_file("secrets.txt");
14.      send_to_network(x);
15.    }
16.  }
```

Example of ideal output

- Flow 1:
  - Source: File system
    - Filename is specified by user.
  - Sink: Network
    - IP: 127.0.0.1
    - Port: 12345
  - Condition: User input satisfies `is_upload_cmd` (line 3).
- Flow 2:
  - Source: File system
    - Filename is hardcoded “secrets.txt”.
  - Sink: Network
    - IP: 127.0.0.1
    - Port: 12345
  - Condition: Incoming network data satisfies `is_special_command` (line 12).

Current output of our tool

- Output: list of unique (*source*, *sink*, *conditional_edge*) tuples.
  - The *conditional_edge* field specifies an outgoing edge (in the control-flow graph) of a conditional jump.
    - It is represented as a pair of *(cond_jump, jump_target)*, where *cond_jump* and *jump_target* identify a source-code location in the form of a tuple of *(filename, line number, column number)*.
  - The fields *source* and *sink* hold the names of system API functions, and may also include the call-site source-code location, at the user's choice.
- Example output for E1:

```
[
 {"src":  "read_from_file",
  "sink": "send_to_network",
  "cond_edge": [{"file":"E1.c", "line":3, "col":3},
                {"file":"E1.c", "line":4, "col":5}]},
 {"src":  "read_from_file",
  "sink": "send_to_network",
  "cond_edge": [{"file":"E1.c", "line":12, "col":3},
                {"file":"E1.c", "line":13, "col":5}]},
 ...
]
```

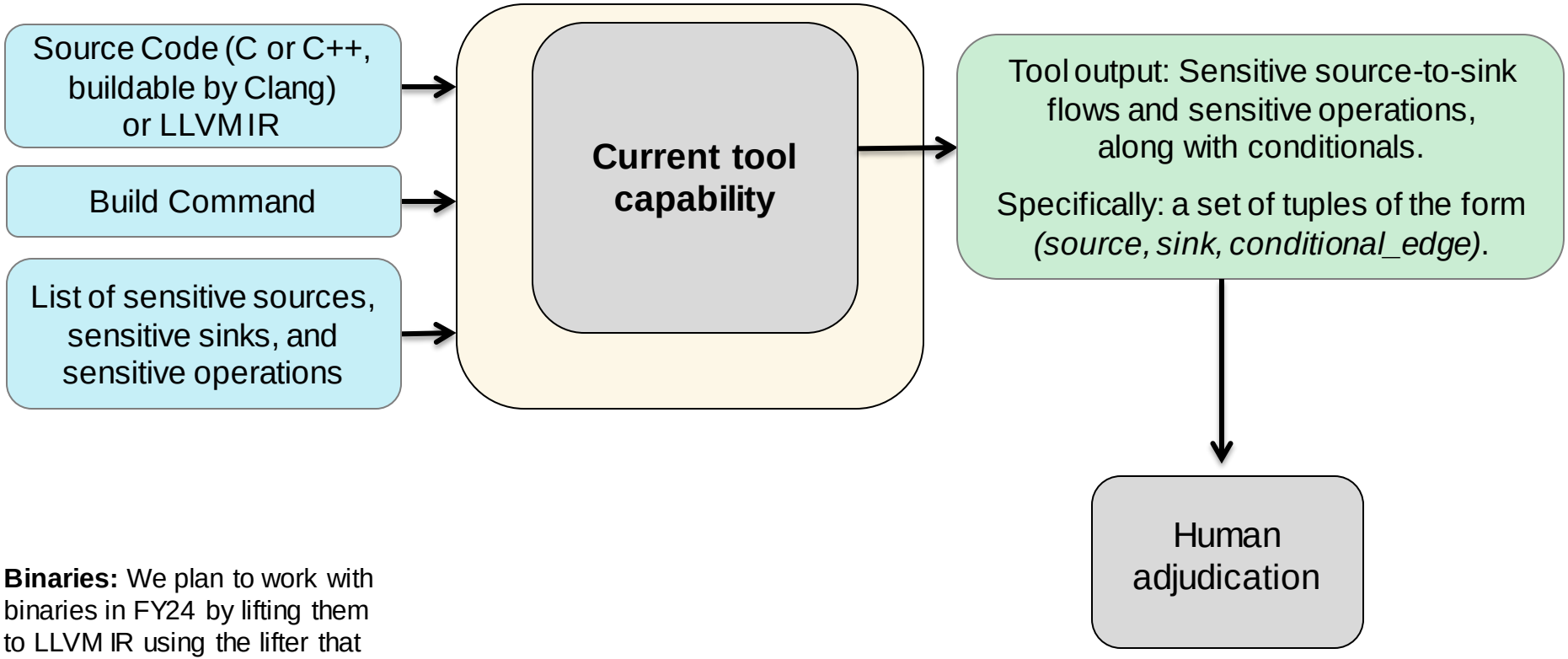
Example E1

```
1.  function Flow_1() {
2.    cmd = read_from_keyboard();
3.    if (is_upload_cmd(cmd)) {
4.      name = get_file_name(cmd);
5.      x = read_from_file(name);
6.      send_to_network(x);
7.    }
8.  }
9.
10. function Flow_2() {
11.   data = read_from_network();
12.   if (is_special_cmd(data)) {
13.     x = read_from_file("…");
14.     send_to_network(x);
15.   }
16. }
```

Output of initial tool (continued)

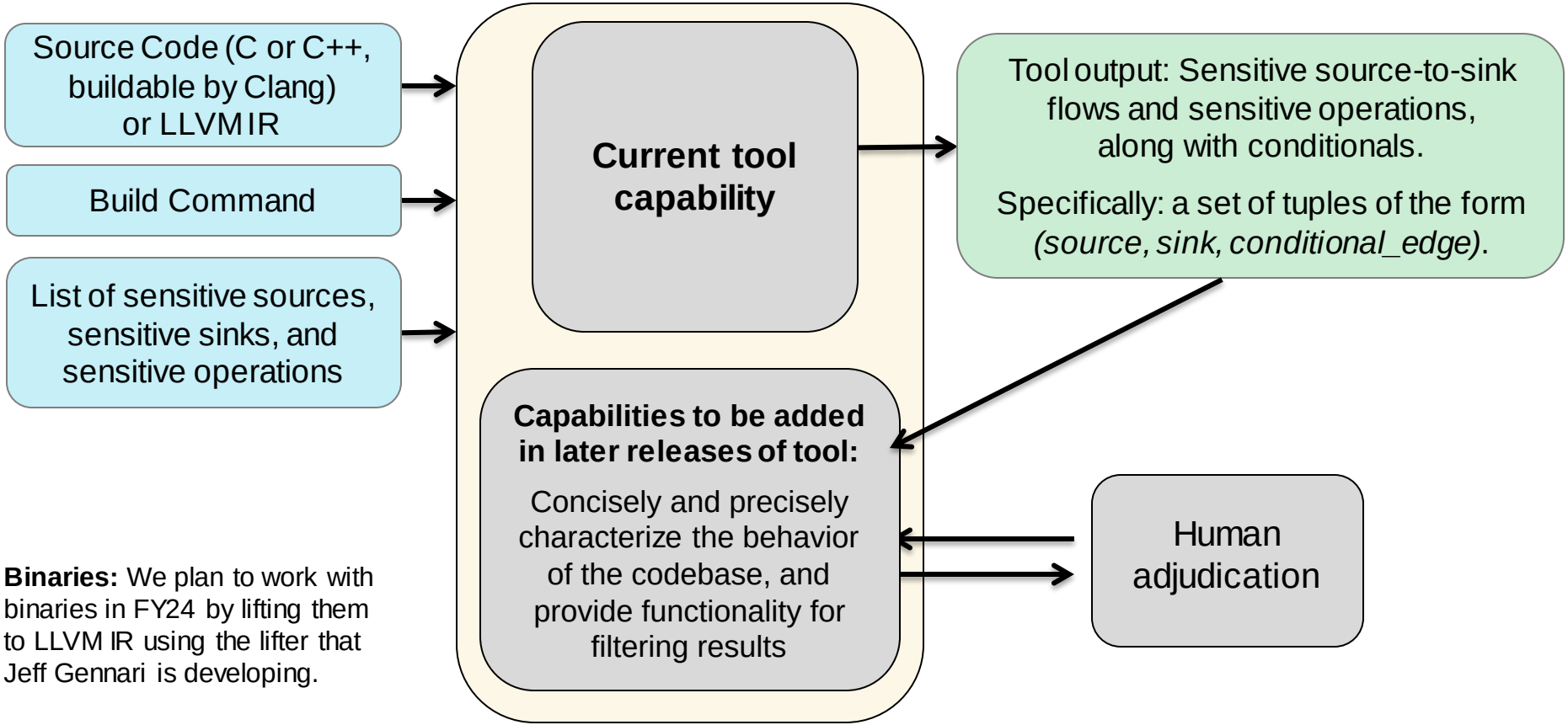
- If a source-to-sink flow happens unconditionally, a dummy value NULL is used in the *conditional_edge* field.
- If a flow involves multiple conditionals, then the output includes a tuple for each conditional.
  - So, an upper bound on the number of entries in the output list is: num_sources * num_sinks * (num_cond_edges + 1).
- For sensitive operations without a source-to-sink flow:
  - The *source* field is NULL, and
  - the *sink* field is the sensitive API function.

Diagram of our tool, with its input and output



Binaries: We plan to work with binaries in FY24 by lifting them to LLVM IR using the lifter that Jeff Gennari is developing.

Diagram of our tool, with its input and output



Binaries: We plan to work with binaries in FY24 by lifting them to LLVM IR using the lifter that Jeff Gennari is developing.

Demo of tool on toy example

```
$ docker run --rm -v $PWD:/data phasar -m /data/mal-client-2.ll -D ifds-taint
--analysis-config /data/file-to-net.config.json --call-graph-analysis=cha
```

```
----- Found the following leaks -----
Source: function 'fread', callsite: /data/toybench/mal-client-2.c, line  44, col 13
Sink:   function 'write', callsite: /data/toybench/mal-client-2.c, line 131, col 21
Involved conditional edges:
  [Line38:c9] -> [Line39:c9]          'if (fp) {'
  [Line43:c13] -> [Line44:c13]        'if (ret) {'
  [Line60:c9] -> [Line65:c57]         'if (protoent == NULL) {'
  [Line66:c9] -> [Line71:c31]         'if (sockfd == -1) {'
  [Line72:c9] -> [Line77:c76]         'if (hostent == NULL) {'
  [Line78:c9] -> [Line85:c5]          'if (in_addr == (in_addr_t) - 1) {'
  [Line94:c9] -> [Line99:c5]          'if (connect_error) {'
  [Line107:c13] -> [Line115:c32]      'if (user_input_len == -1) {'
  [Line115:c13] -> [Line116:c21]      'if (strcmp("upload\n", user_input) == 0) {'
  [Line118:c17] -> [Line126:c13]      'if (user_input_len == -1) {'
  [Line127:c17] -> [Line129:c47]      'if (access(user_input, F_OK) == 0) {'
  [Line130:c21] -> [Line131:c49]      'if (fileContents) {'
```

Parameterized sinks

- In the `mal-client-2.c` example, the system API function **`write`** is listed as a sink.
- However, this function can write to both network sockets and to regular files, depending on its first argument (the file descriptor).
- To distinguish between sending data to the network and writing data to a local file, we will do an auxiliary information-flow analysis to trace the origin of the file descriptor used in the call to `write`.

Next steps

- Write auxiliary dataflow analyses to provide more information about sources and sinks, e.g.:
  - Whether a file descriptor corresponds to a regular file, a network socket, or something else.
  - For file-system sources/sinks: information about the file path (e.g., hard-coded filenames or directories, whether the filename came from another sensitive source, etc.).
- Expand our default list of sensitive sources, sensitive sinks, and sensitive operations.
- Identify features of conditionals that are indicative of timebombs / logic bombs, including for operations that don't involve flows of sensitive information.
  - Yanick et al. "Triggerscope: Towards detecting logic bombs in Android applications." IEEE S&P, 2016.
- Iteratively refine the tool to ease the manual burden of reviewing the results.
  1. Run the tool on a codebase and manually adjudicate whether the behaviors are malicious or benign.
  2. Record difficulties encountered and how to improve the tool to make the manual review more efficient.
  3. Implement those improvements and repeat.
- Improve support for decompiled/lifted binaries.

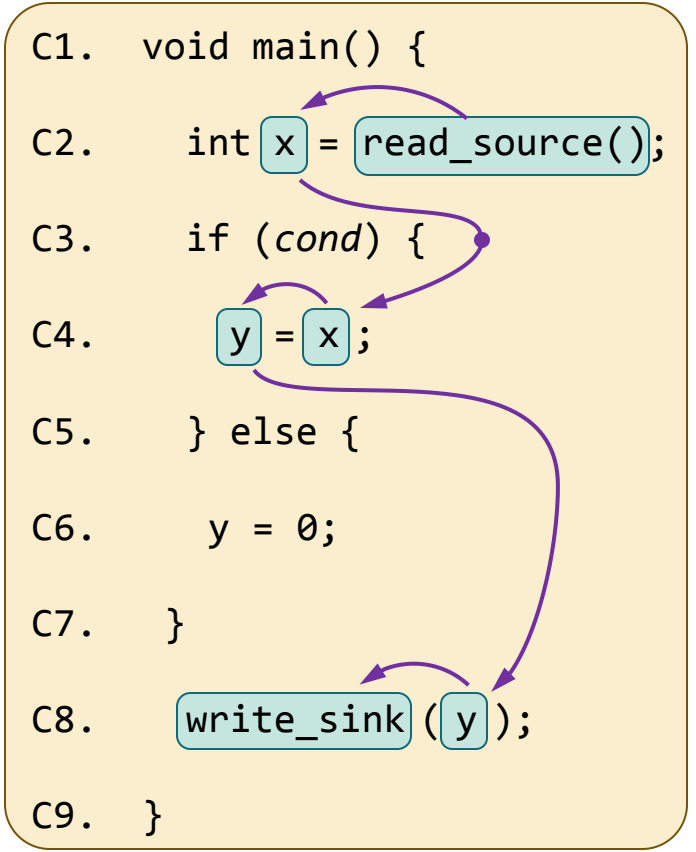
Additional Details

Flow paths

- A *flow path* describes a flow of information in a single run of the program.
- Example: The arrows in the diagram at the right illustrate a flow path from `read_source` to `write_sink`.
  - Symbolically, we write this flow path as:

```
[(C2, x, read_source),
 (C3, x, x),
 (C4, y, x),
 (C8, write_sink, y)]
```

- In general, we represent a flow path as a sequence of (*command*, *new*, *old*) tuples such that:
  1. The *old* field of each tuple is the same as the *new* field of the previous tuple.
  2. There is a direct flow from *old* to *new* during *command*. (This includes the case where *old* is untouched and *old* = *new*.)
  3. The sequence of commands is a *trace* (i.e., the sequence of instructions executed in a run of the program) or part of a trace.



Implicit flows

We say there's an *implicit flow* from a source to a sink iff: data written to the sink depends on which branch of a conditional jump is taken, which in turn depends on data from the source.

Implicit flow:

```
x = read_bit_from_source();
if (x) {y=1;} else {y=0;}
write_bit_to_sink(y);
```

Explicit flow:

```
x = read_bit_from_source();
if (rand_bool()) {y=x;} else {y=0;}
write_bit_to_sink(y);
```

Implicit flows are evident only when examining multiple traces, in contrast to explicit flows, which can be shown (via a flow path) on a single trace.

We currently don't plan to consider implicit flows in this project.

- Techniques for implicit flows generally introduce an excessive amount of false alarms.
- However, there are heuristics that we can try to identify laundering of data thru an implicit flow.

Two ways that an explicit flow can depend on a conditional

Way 1: The tainted data is written to a memory location (or sink) inside a branch:

```
void main() {
  int x = read_source();
  if (condition) {
    y = x;  // true branch
  } else {
    y = 0;  // false branch
  }
  write_sink(y);
}
```

Way 2: The tainted data is overwritten with untainted data inside one branch but not the other:

```
void main() {
  int x = read_source();
  if (condition) {
    // empty true branch
  } else {
    x = 0;  // false branch
  }
  write_sink(x);
 }
```

Memory abstraction

- To run the analysis in a reasonable amount of time, we must abstract the memory.
- Examples:
  - All the elements in an array may be conflated together.
  - All memory allocated at a single `malloc` callsite may be conflated together.
- The IFDS taint analysis is orthogonal to the type of memory abstraction used.

Backup Slides

Conditional paths in unstructured LLVM IR

- Earlier, we used the terminology "inside a conditional branch". This works for "`if`" statements in structured code, but we also need to handle loops and GOTOs.
- Recall: We say a ***conditional edge*** is an outgoing edge (in the control-flow graph) of a conditional jump.
- For each conditional edge, we define one or more ***merge edges***.
  - For an "`if`" statement in structured code, a merge edge is where the branch ends, coming back together with the other branch. A more general definition is given on the next slide.
- We say that a ***conditional path*** is a path in the control-flow graph that:
  1. starts with a conditional edge *e*,
  2. ends with a merge edge of *e*, and
  3. doesn't repeat any edges.
- Now we will speak of "inside a conditional path" instead of "inside a conditional branch".

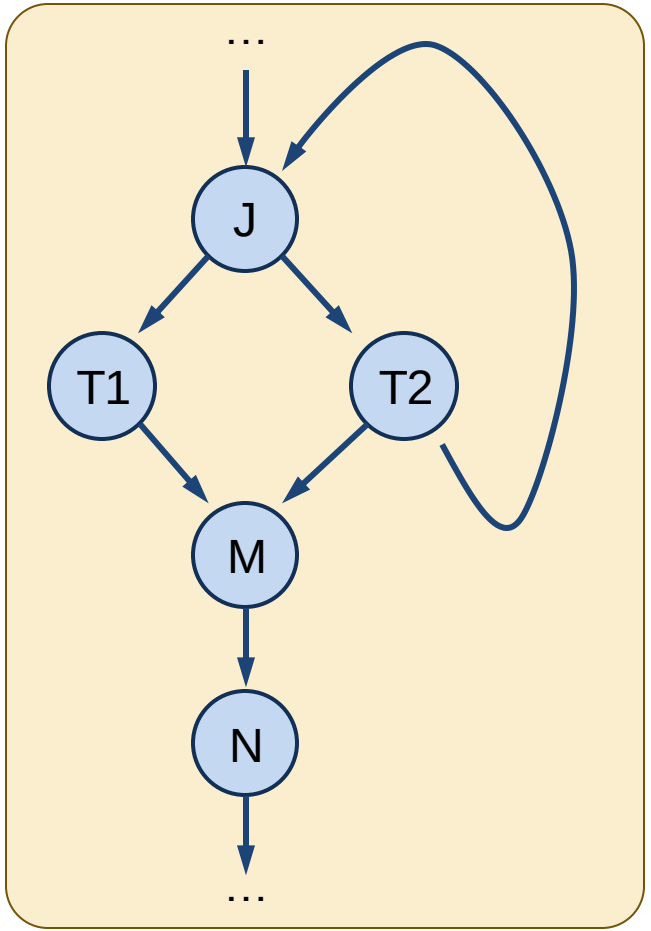
Merge edges

- Consider a conditional edge (J → T).
  - J is a conditional-jump instruction.
  - T is an instruction that J can jump to.

- An edge (X → Y) is a merge edge for (J → T) iff:
  1. Y postdominates J or dominates J, and
  2. X is reachable from J without passing thru another merge edge.

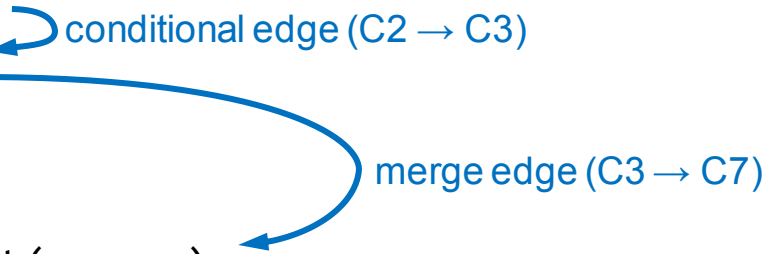
  - Y ***dominates*** J iff Y appears in every path from the function entry node to J.
  - Y ***postdominates*** J iff Y appears in every path from J to the function exit node.

- In the example at the right:
  - (T1 → M) is a merge edge for (J → T1).
  - (T2 → M) is a merge edge for (J → T2).
  - (T2 → J) is a merge edge for (J → T2). (Every node dominates itself.)
  - (M → N) is not a merge edge.



Example 1 of merge edges (structured if/else)

```
C1:   x = read_file(...);
C2:   if (cond) {
C3:       y = x;
      } else {
C5:       y = 0;
      }
C7:   write_to_net(y, ...);
```



- The merge edge for conditional edge (C2 → C3) is (C3 → C7).
- The merge edge for conditional edge (C2 → C5) is (C5 → C7).
- The conditional paths are:
  - C2 → C3 → C7
  - C2 → C5 → C7
- Add to the output list: (read_file, write_to_net, (C2 → C3))

Example 2 of merge edges (GOTO version of example 1)

```
C1:   x = read_file(...);
C2:   if (cond) {goto C3;} else {goto C5;}
C3:   y = x;
C4:   goto C6;
C5:   y = 0;
C6:   write_to_net(y, ...);
```

- The merge edge for conditional edge (C2 → C3) is (C4 → C6).
- The merge edge for conditional edge (C2 → C5) is (C5 → C6).
- The conditional paths are:
  - C2 → C3 → C4 → C6
  - C2 → C5 → C6
- Add to the output list: (read_file, write_to_net, (C2 → C3))

Example 3 of merge edges (empty “else” branch)

```
C1:   x = read_file(...); y = 0;
C2:   if (cond) {goto C3;} else {goto C5;}
C3:   y = x;
C4:   goto C5;
C5:   write_to_net(y, ...);
```

- The merge edge for conditional edge (C2 → C3) is (C4 → C5).
- The conditional edge (C2 → C5) is identical to its merge edge.
- The conditional paths are:
  - C2 → C3 → C4 → C5
  - C2 → C5
- Add to the output list: (read_file, write_to_net, (C2 → C3))

Example 4 of merge edges (unstructured loop)

```
C1:   y = 0;
C2:   if (cond) {goto C3;} else {goto C5;}
C3:   y = read_file(...);
C4:   goto C2;
C5:   write_to_net(y, ...);
```

- The merge edge for conditional edge (C2 → C3) is (C4 → C2).
- The conditional edge (C2 → C5) is identical to its merge edge.
- The conditional paths are:
  - C2 → C3 → C4 → C2
  - C2 → C5
- Add to the output list: (read_file, write_to_net, (C2 → C3))

Example 5 of merge edges (structured loop)

```
C1:   y = 0;
C2:   while (cond) {
C3:     y = read_file(...);
      }
C5:   write_to_net(y, ...);
```

- The merge edge for conditional edge (C2 → C3) is (C3 → C2).
- The conditional edge (C2 → C5) is identical to its merge edge.
- The conditional paths are:
  - C2 → C3 → C2
  - C2 → C5
- Add to the output list: (read_file, write_to_net, (C2 → C3))

Definition of “directly depends”

Let L_1 and L_2 be memory locations, and let C be an IR instruction.

The value held in L_2 immediately after executing C ***directly depends*** on the value held in L_1 immediately before executing C iff one of the following holds true:

1. $L_2 = L_1$ and L_1 isn’t written to by C
2. C computes an operation (e.g., an arithmetic or bitwise operation) using the value in L_1 and stores the results in L_2
3. C takes the value in L_1 and writes it to L_2
   - If C is a call instruction, then C is considered to take the actual arguments at the callsite and write them to the memory locations of the formal parameters of the callee.
   - If C is a return instruction, and the calling function (which is being returned to) assigns the return value to a variable x, then the return instruction is considered to write the return value to the memory location of x.