Detection of Malicious Code

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

Will Klieber

April 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.

DM23-0445

Outline

- Problem
- Our solution, at a high level
- Motivation for the new technique that we are developing
- Details of the core information our tool will provide to analysts
- Potential next steps to make the tool more useful
  - We want your input on what would be most valuable to you.
  - We want our tool to be able to easily integrate with your existing workflows.

Problem

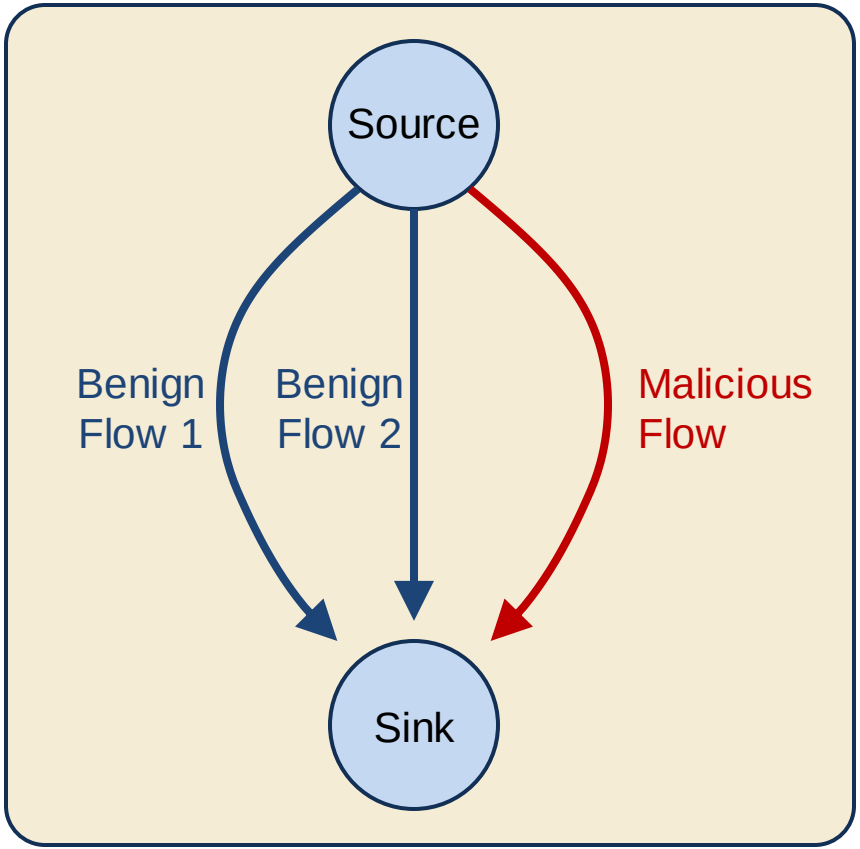
- DoD uses much software produced by various supply chains.
- These supply chains can be compromised by an adversary:
  - Network intrusion
  - Insider threat
- Failing to detect malicious code can be very costly.
- Detection is currently impractical.
- Specifically, we aim to detect two types of malicious code:
  - Exfiltration of potentially sensitive information (e.g., keyloggers)
  - Timebombs / logic bombs, Remote-Access Trojans, etc:
    Calling a potentially sensitive system API call (e.g., writing to a file, starting a new process, etc.) in response to a potentially questionable trigger (e.g., on a specific date, in response to incoming network packets, etc.)

Our solution

- We will use *information flow* techniques, as well as other static analysis.
  - We are building on Phasar, an LLVM-based static-analysis framework: https://github.com/secure-software-engineering/phasar
- 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) that arise from violation of secure-coding rules are outside the main focus of this project.

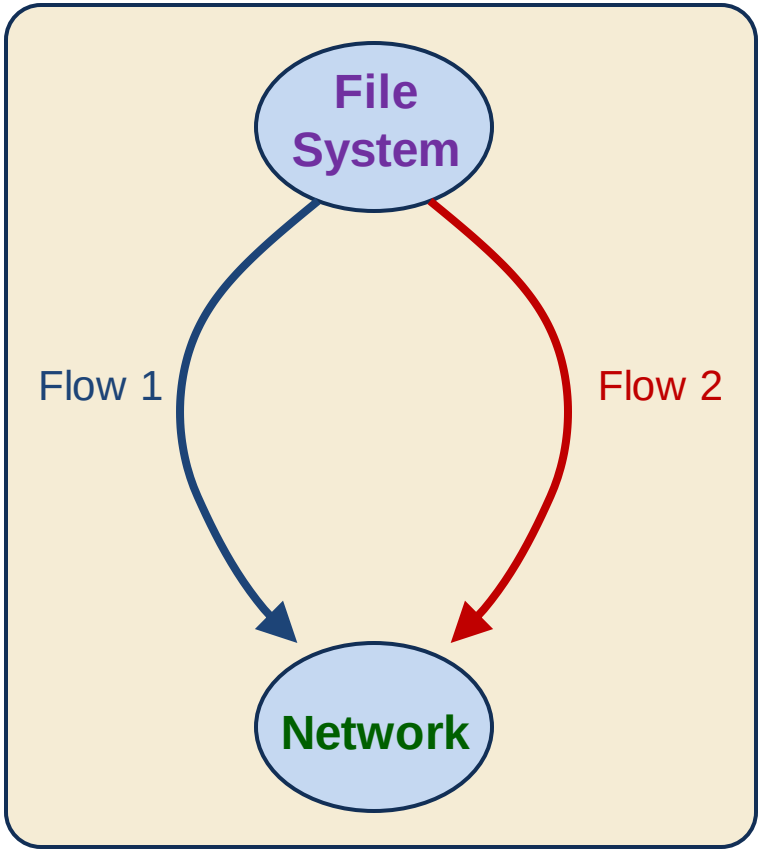
Information Flow Analysis

- 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: designated system API calls that return potentially sensitive information.
  - Sinks: 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:** Flows that depend on different conditionals in the code should be kept separate.



Motivating example E1 (pseudocode)

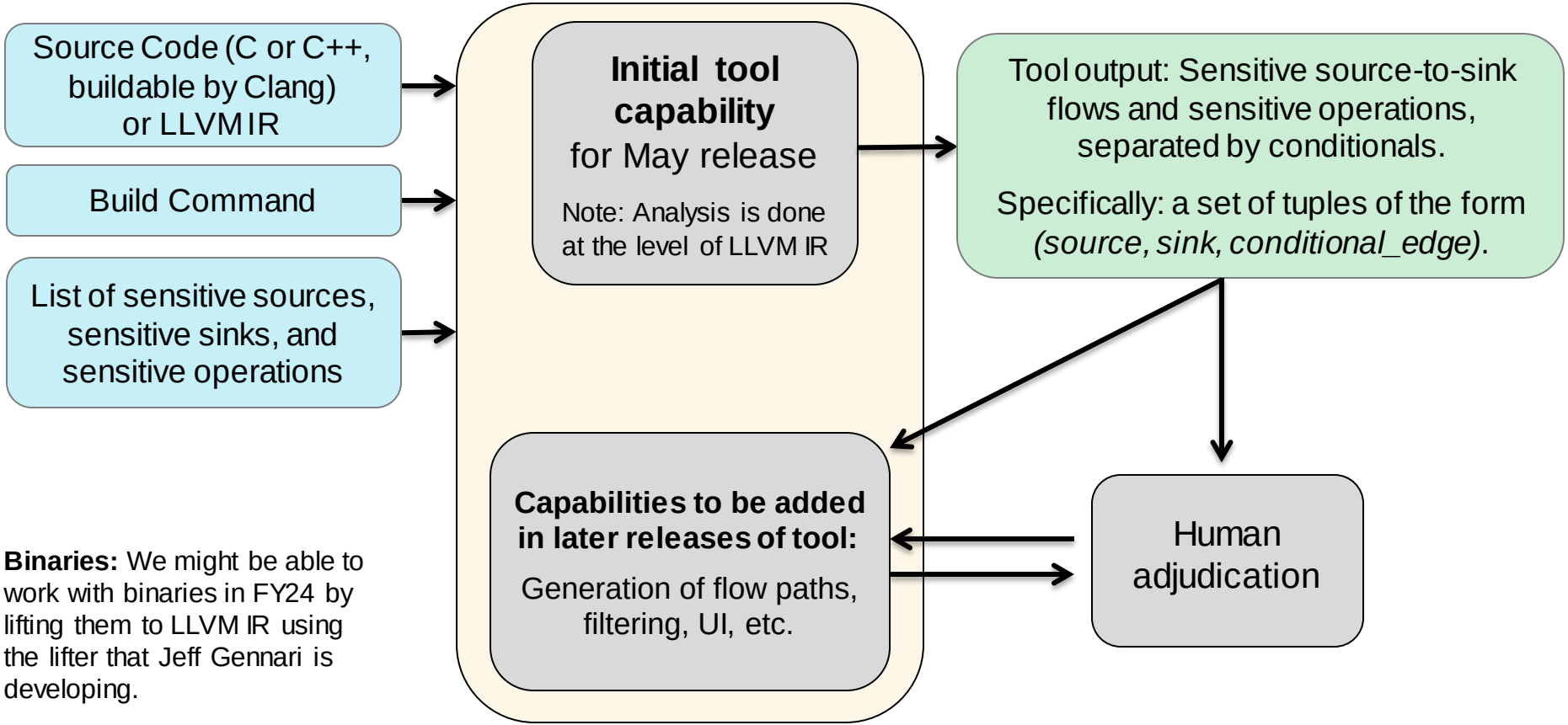
```
1.   function Flow_1() {
2.     cmd = read_from_keyboard();
3.     if (is_upload_cmd(cmd)) {
4.       name = get_file_name(cmd);
5.       x = read_from_file(name);
6.       send_to_network(x);
7.     }
8.   }
9.
10.  function Flow_2() {
11.    data = read_from_network();
12.    if (is_special_cmd(data))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



Motivating example (continued)

- In example E1 (on the previous slide):
  - Flow 1 happens if the `true` branch of one conditional is taken, and
  - Flow 2 happens if the `true` branch of another conditional is taken.
- Standard taint analysis conflates these two flows together.
- **Our idea:** Separate the flows by which branches of which conditionals need to be taken for the flow to happen.

Diagram of our tool, with its input and output



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

Output of initial tool

- The output of our initial tool will be a 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)*.
  - At the user's choice, the fields *source* and *sink* may simply hold the names of system API functions, or they may also include the source-code location.
- Example output for E1 (reproduced at the right):

```
[
 {"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}]},
 ...
]
```

```
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)

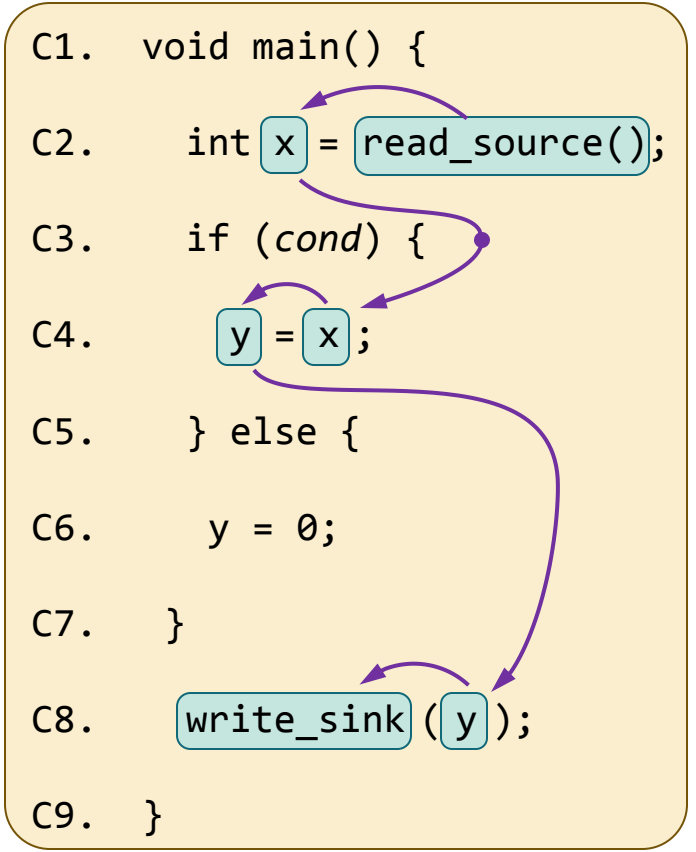
- If a source-to-sink flow happens unconditionally, a dummy value NULL is used in the *conditional_edge* field.
- If a flow involves multiple conditionals, then the output includes a tuple for each conditional.
  - So, an upper bound on the number of entries in the output list is: num_sources * num_sinks * (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.
- In addition to the set of (*source*, *sink*, *conditional_edge*) tuples, we plan to provide a *flow path* (described on next slide) for each tuple.

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.



Generation of flow paths

- We plan to provide one flow path for each (*source*, *sink*, *conditional_edge*) tuple.
- Also, we plan to provide functionality for querying for additional flow paths.
- Query conditions might include:
  - Source and sink
  - Which conditionals must be involved in the flow
  - Which conditionals must not be involved in the flow
  - Which abstract memory locations must (or must not) be involved in the flow

User interface / API for tool

- The raw output of the initial tool probably won't be convenient to use as-is.
- We can add UI / API functionality to make the tool more convenient to work with.
- Example functionality:
  - Marking a conditional as non-suspicious and filtering it out.
  - Marking a source/sink callsite as non-suspicious and filtering it out.
  - Displaying the relevant parts of the source code when investigating a flow.
  - Integration with an existing code editor / IDE / etc.

Additional static-analysis functionality

- Identifying and highlighting/prioritizing suspicious features of conditionals (e.g., features indicatives of timebombs)
- Separating flows that depend on conditional control flow other than conditional `br`:
  - `switch`, `indirectbr`, function pointers (easy to implement)
  - C++ exception handling, setjmp/longjmp (harder to implement)
- Separating flows by pointer-aliasing conditions they depend on, e.g.:

```
int x=0; int* p1 = &x;
int y=0; int* p2 = &y;
if (cond) {p2 = p1;}
*p1 = read_source();
write_sink(*p2);
```

Questions (1) – integration and measurement of our tool

- I want to ensure that the tool being developed can usefully fit into your workflows.
- I was thinking it might be helpful for me to make an in-person visit to learn more about your existing workflows and tools. What do you think of a visit?
  - Perhaps a “ride along” where I see your current practices and tools?
  - Do you use any kind of taint flow analysis today? If so, how do you use them?
  - Anything else you can tell us about your current workflow, to help ease integrating our tool?
- What can we measure w.r.t. your current baseline and how our tool improves on it?
  - E.g.: false-positive rate, false-negative rate, and/or amount of manual effort.
  - What metrics are most important to you when evaluating our tool?
  - In terms of these metrics, where do we need to be for the tool to be useful to you in practice?
  - From our last meeting: something like “get us from 100,000 LoC down to 400 LoC to review, even if 90% of remaining alerts are false positives”

Questions (2)

1. How would you imagine using the flow information provided by the tool?
   - Set of *(source, sink, conditional_edge)* tuples
   - Flow paths
2. What features of flow paths might you want to prescribe in the query and want identified in the output?
3. Currently, we plan to treat system calls (and direct I/O) as sources. We can also provide functionality for treating some of the program's internal data as sensitive – would that be a high-priority capability?
4. What potential capabilities / features are most important to you?

Questions (3)

1. What we need to get our tool accepted into your environment? (E.g., SBOM, etc.)
   - We plan to distribute a self-contained Linux Docker image – does this work for you?
2. What size of codebases are you usually looking at?
3. Set up regular monthly meetings?
4. Is handling *implicit flows* a high priority? (See next slide)

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 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.

Backup slides

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

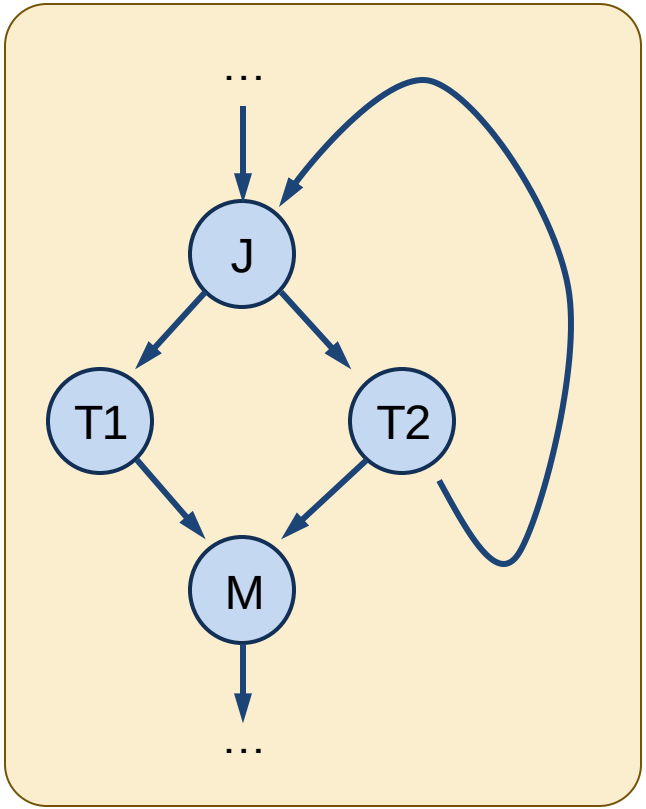
- In order to run the analysis in a reasonable amount of time, we must abstract the memory so that there are a relatively small number of *abstract memory locations*.
- Each abstract memory location conflates together multiple concrete memory locations.
- For example, usually a single abstract memory location is usually used to represent all the elements in an array.
- With *allocation site abstraction*, all memory allocations at a single *allocation site* (e.g., a `malloc` callsite in the codebase) are conflated together.
- The IFDS taint analysis is orthogonal to the type of memory abstraction used.

Separating the flows – unstructured LLVM IR

- Earlier, we used the terminology "inside a conditional branch". This works for structured "if" statements, but LLVM IR can also have loops and unstructured 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*. A ***merge edge*** is, roughly, where the branch ends, coming back together with the other branch. (More on 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.
- For unstructured code, "inside a conditional branch" becomes "inside a conditional path"

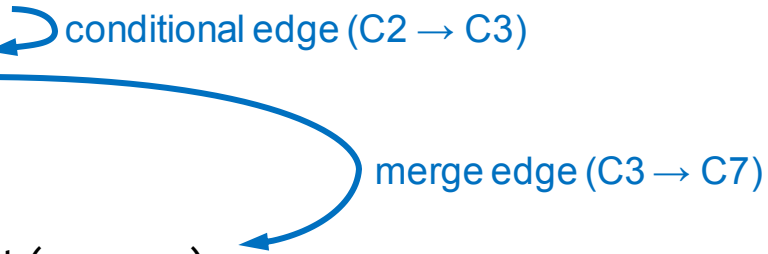
Merge edges

- 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 (pre-)dominates J, and
  2. X is reachable from J without passing thru another merge edge.
- 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).



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.

Demo of Phasar on toy example

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

```
PhASAR v1222
A LLVM-based static analysis framework

----- Found the following leaks -----
At instruction
IR  : %call1 = call i64 @write(i32 noundef %sockfd, i8* noundef %s,
i64 noundef %call) #16, !dbg !376, !psr.id !377 | ID: 111

Leak(s):
IR  : i8* %s | ID: send_to_network.1
```

Phasar example: `file-to-net.config.json` (list of sources and sinks)

```
{ "name": "taint-config-test",
  "version": 1.0,
  "functions": [
    { "name": "fread",
      "params": {
        "source": [ 0 ]
      }
    },
    { "name": "write",
      "params": {
        "sink": [ 1 ]
      }
    }
  ]
}
```

Parameterized sinks

- In the `mal-client.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`.