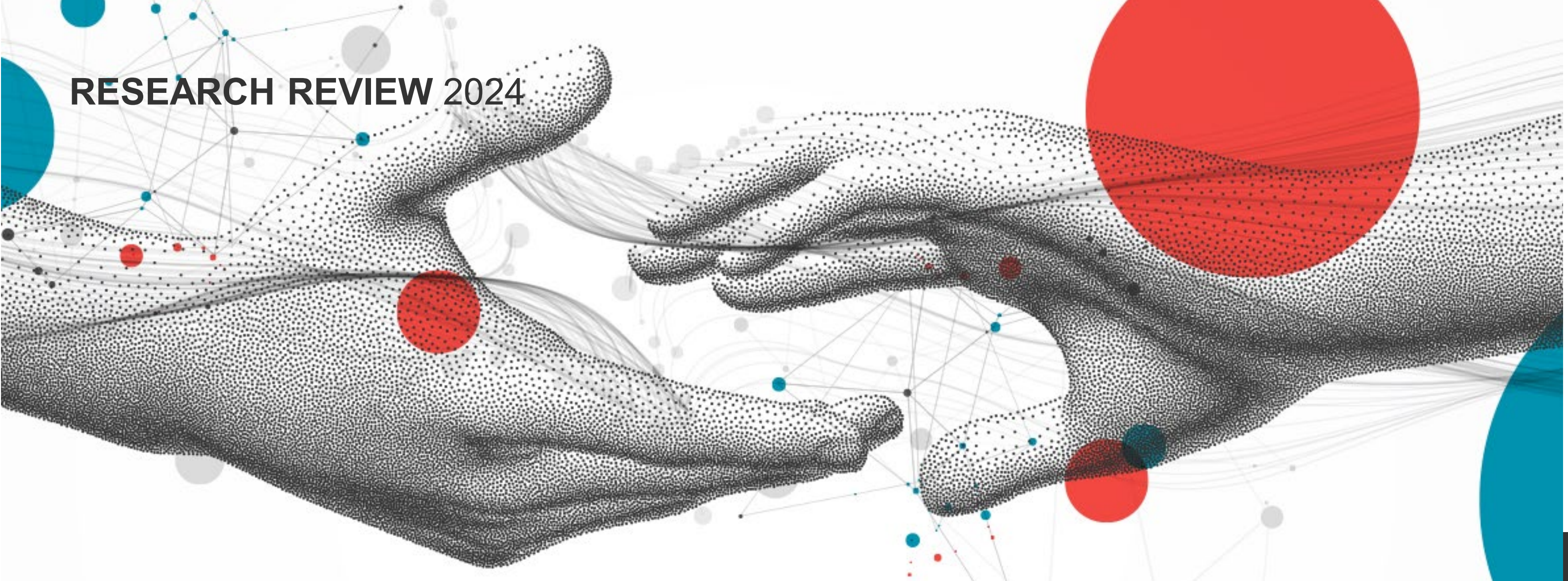
RESEARCH REVIEW 2024



Carnegie Mellon University
Software Engineering Institute

Detection of Malicious Code Using Information Flow Analysis

NOVEMBER 13, 2024

Will Klieber
Software Security Researcher

Document Markings

Copyright 2024 Carnegie Mellon University.

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.

The view, opinions, and/or findings contained in this material are those of the author(s) and should not be construed as an official Government position, policy, or decision, unless designated by other documentation.

References herein to any specific entity, product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute nor of Carnegie Mellon University - Software Engineering Institute by any such named or represented entity.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution.

This material may be reproduced in its entirety, without modification, and freely distributed in written or electronic form without requesting formal permission. Permission is required for any other use. Requests for permission should be directed to the Software Engineering Institute at permission@sei.cmu.edu.

CERT® and Carnegie Mellon® are registered in the U.S. Patent and Trademark Office by Carnegie Mellon University.

DM24-1172

Problem

- Department of Defense (DoD) software supply chains
- Example incidents:
  - `xz` backdoor incident of 2024
  - SolarWinds incident of 2020: infected 18,000 organizations, 100 of which were then targeted
- Our tool detects two types of malicious code:
  1. Exfiltration of sensitive information
  2. Timebombs/logic bombs, remote-access Trojans (RATs), etc.
- We call our tool “DMC” (short for “Detection of Malicious Code”).
  - `https://github.com/cmu-sei/dmc`
- Project status: Began October 2022, ending November 2024

Our Approach (1)

- Our tool flags code as **potentially** malicious.
- It detects "business logic" vulnerabilities (such as Log4Shell in Log4j) too.
- Out of scope: Undefined behavior (e.g., buffer overflows)
- **Goal for our tool:** concise and precise output → quick and accurate human adjudication

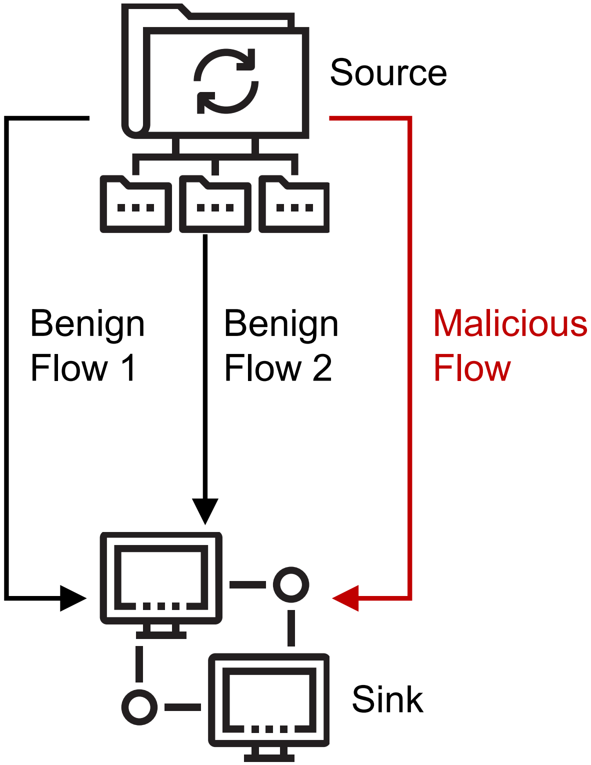
Our Approach (2)

- We are using only static analysis, not dynamic analysis.
- So far, we have focused on C/C++ codebases.
- Our tool works natively on LLVM intermediate representation (IR).
  - LLVM is a compiler infrastructure project.
  - The name “LLVM” originally stood for “Low Level Virtual Machine.”
- We have some support for binaries by lifting to LLVM IR.
- We can also fairly easily support other languages that compile to LLVM IR.

PhASAR

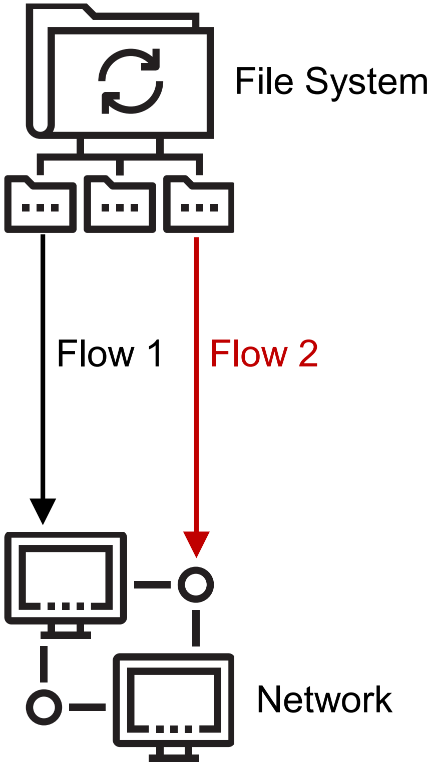
- Initially, we built on PhASAR, which is a static-analysis framework based on LLVM.
- Unfortunately, PhASAR ended up having trouble as we scaled to real-world codebases:
  - Took 15 minutes to analyze `dos2unix` (a very small program, approx. 4,000 lines of code)
  - Ran out of memory (with 24 GB of RAM) on `git`
  - Attempts to simplify the analysis (to speed it up and reduce memory usage) were unfruitful
  - Global variables were always aliased with function parameters, producing many false positives
- Abandoned PhASAR, reimplemented taint analysis from scratch, building only on LLVM
  - We improved scalability by avoiding construction of the supergraph used in PhASAR’s Interprocedural Finite Distributive Subset (IFDS) analysis, at the cost of less context sensitivity.
  - Much faster and less memory-intensive; can analyze `git` (approx. 275,000 lines of code) in just a few minutes with memory usage under 15% on a virtual machine (VM) with 8 GB of RAM
  - Current limitations: Only handles C (with incomplete support for C++), limited alias analysis, limited analysis on function pointers, etc.

Information Flow Analysis



- Static taint analysis to track flow of sensitive data
  - Successful track record (e.g., finding malicious flows of information in Android apps)
  - *Sources* are designated system application programming interface (API) calls that return potentially sensitive information.
  - *Sinks* are designated system API calls that can be used to exfiltrate information to outside the program.
- **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:** Separate the flows by features relevant to detection of malicious code

Motivating Example E1 (Pseudocode)



```
function Flow_1() {
  cmd = read_from_keyboard();
  if (is_upload_cmd(cmd)) {
    name = get_file_name(cmd);
    x = read_from_file(name);
    send_to_network(x);
  }
}

function Flow_2() {
  data = read_from_network();
  if (is_special_cmd(data)) {
    x = read_from_file("secrets.txt");
    send_to_network(x);
  }
}
```

Idea for Ideal Output

Example of Ideal Output

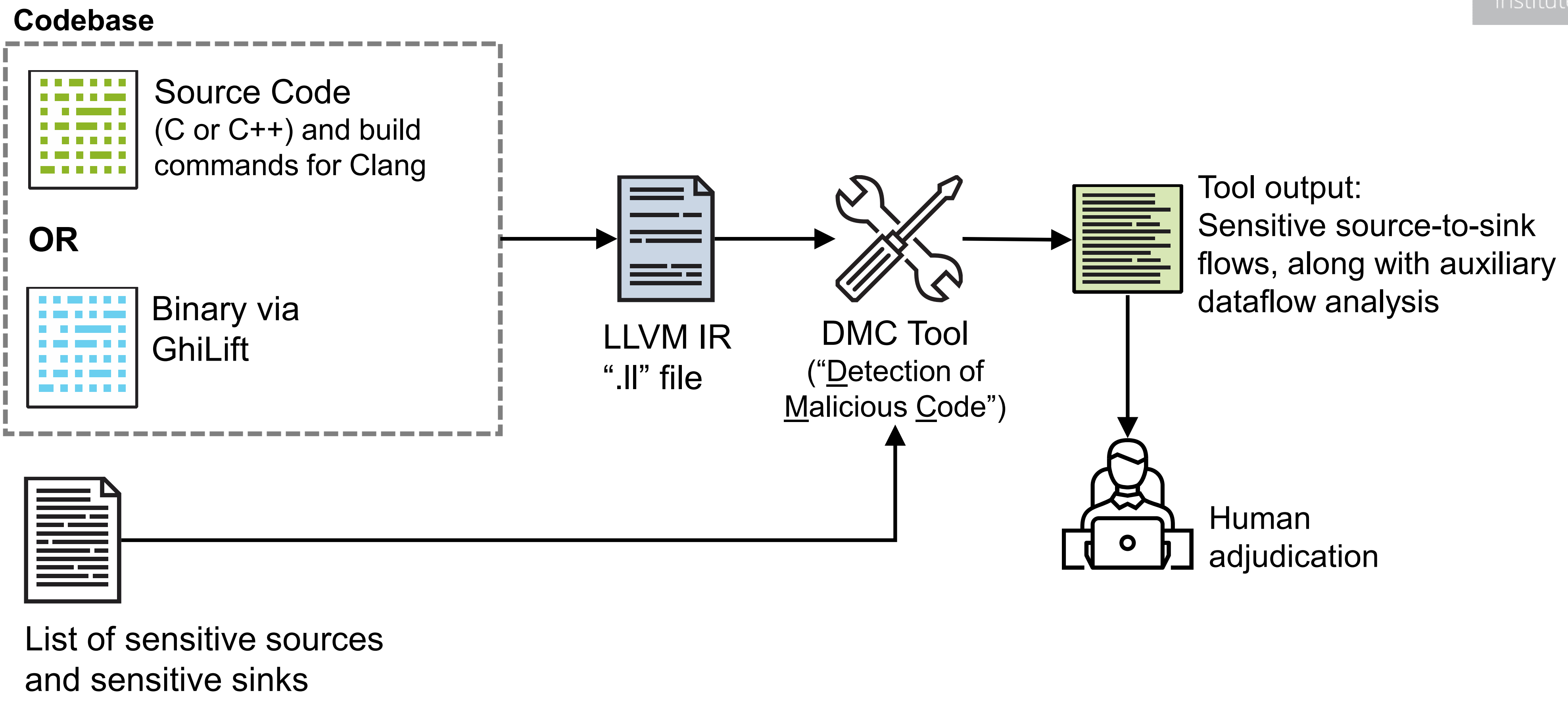
- Flow 1:
  - Source: File system
    - Filename is specified by user.
  - Sink: Network
    - IP: 127.0.0.1
    - Port: 12345

- Flow 2:
  - Source: File system
    - Filename is hardcoded **`secrets.txt`**.
  - Sink: Network
    - IP: 127.0.0.1
    - Port: 12345

Example E1:

```
function Flow_1() {
  cmd = read_from_keyboard();
  if (is_upload_cmd(cmd)) {
    name = get_file_name(cmd);
    x = read_from_file(name);
    send_to_network(x);
  }
}

function Flow_2() {
  data = read_from_network();
  if (is_special_cmd(data)) {
    x = read_from_file("secrets.txt");
    send_to_network(x);
  }
}
```

Diagram of Our Tool with Its Input and Output



Codebase

Source Code
(C or C++) and build
commands for Clang

OR

Binary via
GhiLift

LLVM IR
".ll" file

DMC Tool
("Detection of
Malicious Code")

Tool output:
Sensitive source-to-sink
flows, along with auxiliary
dataflow analysis

Human
adjudication

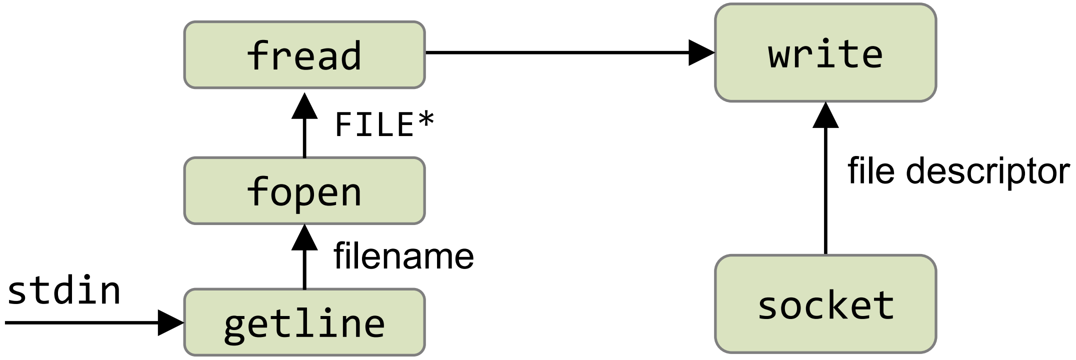
List of sensitive sources
and sensitive sinks

Creating the List of Sources and Sinks

- For well-known functions exported by the operating system and libraries, we used a large language model (LLM), specifically GPT-4, to identify:
  - whether the return value is a source
  - which parameters are sinks
  - which parameters are sources
    - A parameter is a source if it is a pointer to a buffer that the API call fills with potentially sensitive data.
- GPT-4 knows common Windows and Linux API functions. For lesser-known operating systems, the LLM may need a description of the function (e.g., the `man` page).
- Two methods of generating the list of sources and sinks:
  1. Do up-front analysis of system API functions and/or
  2. Run our tool on the program:
     a. The tool’s output will indicate which external functions it doesn’t recognize.
     b. Feed those function names to the LLM.

Simple Example (mal-client-3.c)

```
[
{"sink": {"func":"write", "callsite":["mal-client-3.c","main",152,21],
    "aux file": [{"func":"socket", "callsite":["mal-client-3.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-3.c","main",139,29],
    "aux file": [{"func":"fopen", "callsite":["mal-client-3.c","main",132,26],
      "aux file": [{"func":"getline", "callsite":["mal-client-3.c","main",106,26], "FILE*":"stdin"},
                   {"func":"getline", "callsite":["mal-client-3.c","main",117,30], "FILE*":"stdin"}]}]}]},
...
]
```

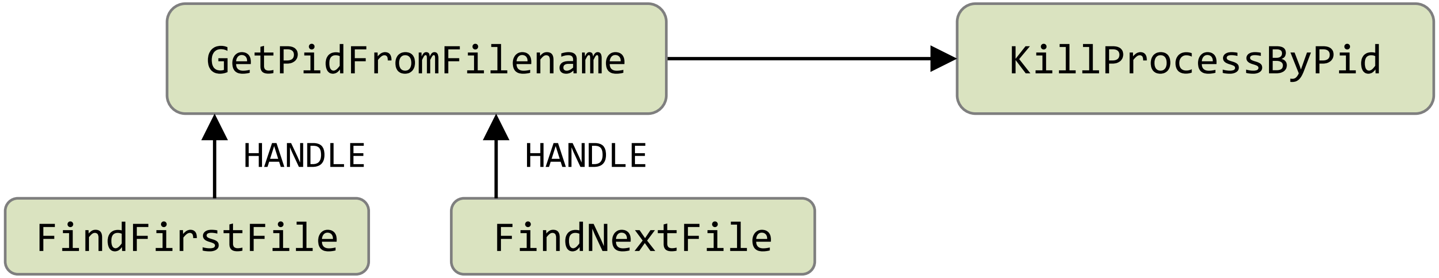


Simple Example (mal-client-3.c)

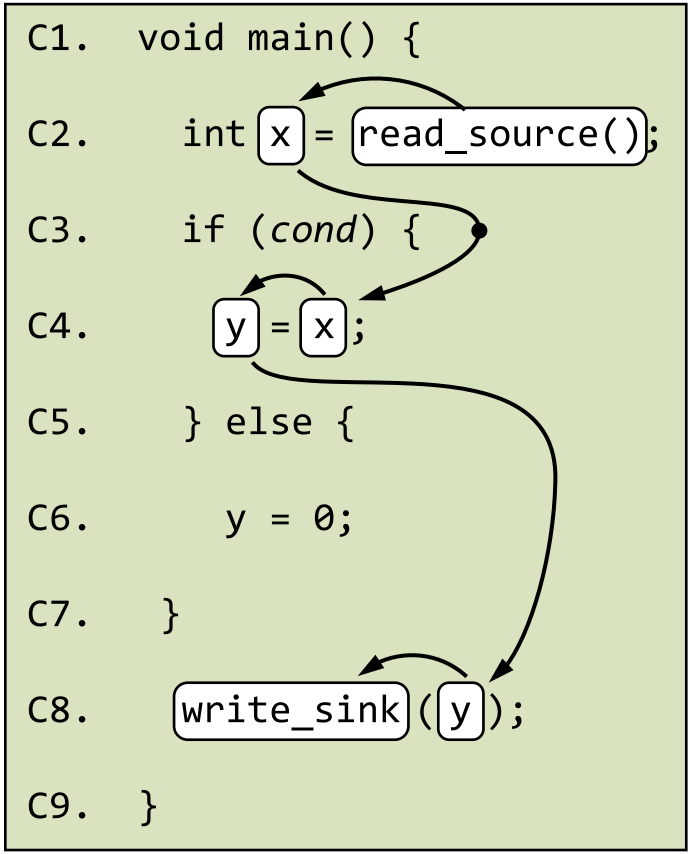
```
[
{"sink": {"func":"write", "callsite":["mal-client-3.c","main",152,21],
    "aux file": [{"func":"socket", "callsite":["mal-client-3.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-3.c","main",139,29],
    "aux file": [{"func":"fopen", "callsite":["mal-client-3.c","main",132,26],
      "aux file": [{"func":"getline", "callsite":["mal-client-3.c","main",106,26], "FILE*":"stdin"},
                   {"func":"getline", "callsite":["mal-client-3.c","main",117,30], "FILE*":"stdin"}]}]}]},

{"sink": {"func":"write", "callsite":["mal-client-3.c","main",158,17],
    "aux file": [{"func":"socket", "callsite":["mal-client-3.c","main",65,18]}]},
 "srcs": [{"func":"getline", "callsite":["mal-client-3.c","main",106,26], "FILE*":"stdin"},
          {"func":"getline", "callsite":["mal-client-3.c","main",117,30], "FILE*":"stdin"}]},

{"sink": {"func":"write", "callsite":["mal-client-3.c","main",194,29],
    "aux file": [{"func":"socket", "callsite":["mal-client-3.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-3.c","main",180,37],
    "aux file": [{"func":"fopen", "callsite":["mal-client-3.c","main",173,34],
      "aux file": [{"filename":"secrets.txt"} ]}]}]}
]
```

Real-World Example: Athena Malware

- Available at: `https://github.com/ytisf/theZoo`
- One malicious action we can detect is finding and terminating other bots.
- `Botkiller.cpp`, function `ScanDirectoryForBots`
- Detected flow:



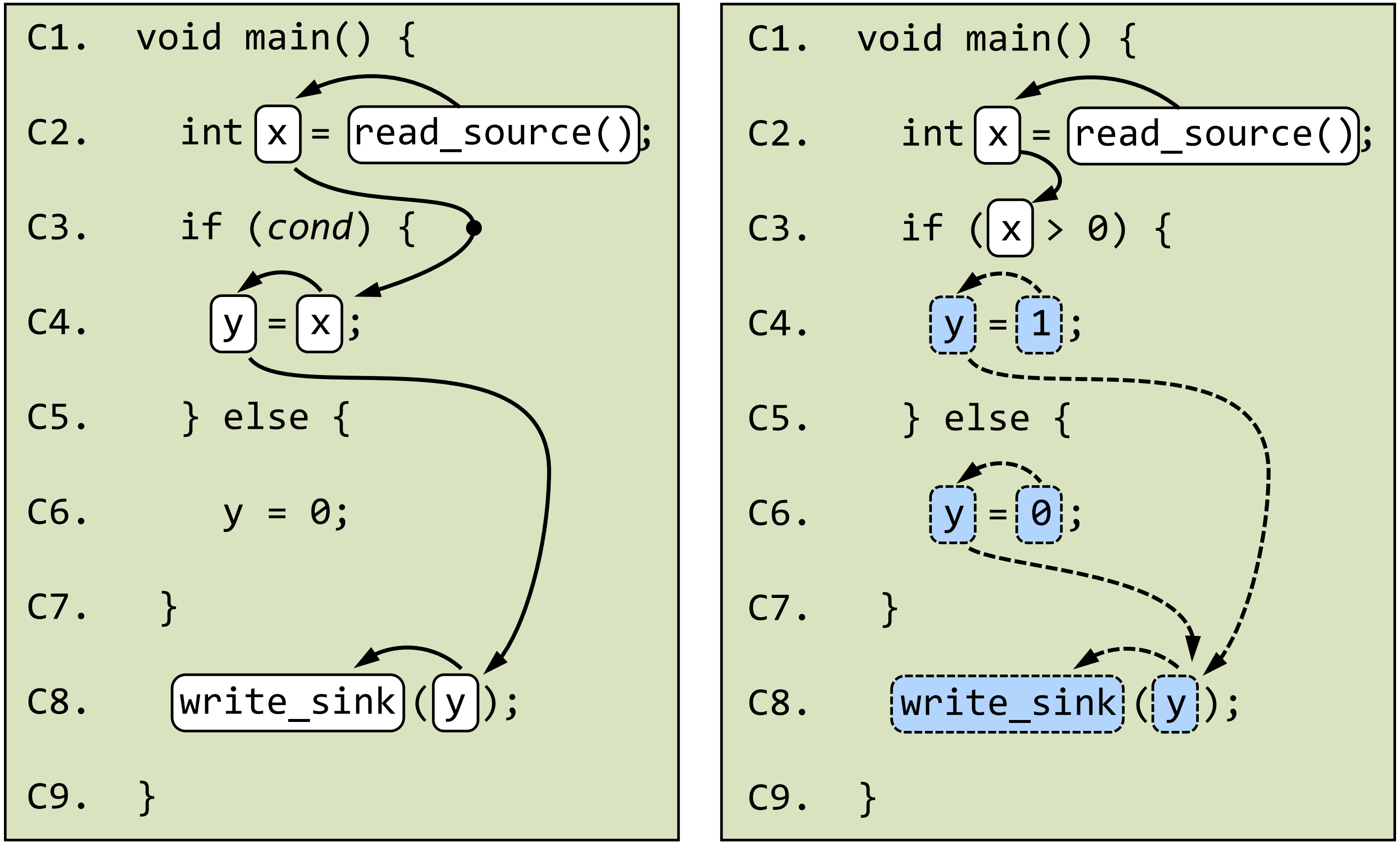
Flow Paths



```
C1.  void main() {
C2.    int x = read_source();
C3.    if (cond) {
C4.      y = x;
C5.    } else {
C6.      y = 0;
C7.   }
C8.    write_sink(y);
C9.  }
```

- A *flow path* describes a flow of information in a single run of the program.
- The arrows in the diagram at the right illustrate a flow path from `read_source` to `write_sink`.
  - For each arrow, there is a direct flow from the origin of the arrow to the target of the arrow.
  - The arrows follow along a *trace* (i.e., the sequence of instructions executed in a run of the program).

Explicit Flow vs. Implicit Flow



```
void main() {
   int x = read_source();
   if (cond) {
     y = x;
   } else {
     y = 0;
   }
  write_sink(y);
}
```

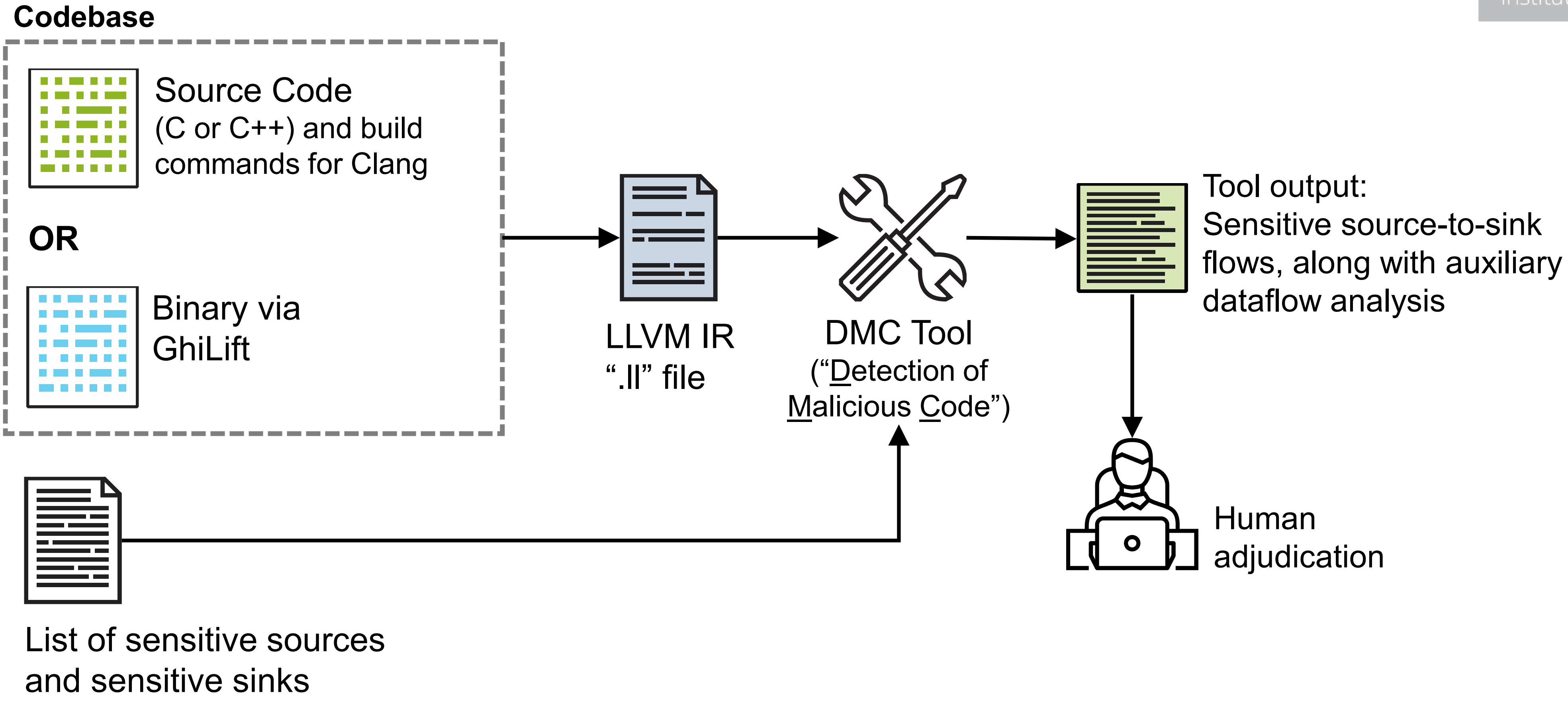
```
void main() {
   int x = read_source();
   if (x > 0) {
     y = 1;
   } else {
     y = 0;
   }
  write_sink(y);
}
```

An *implicit flow* doesn't have a flow path from source to sink; rather, the source influences the sink indirectly via a branch condition.

We do not consider implicit flows in this project.

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

Recap: Diagram of Our Tool with Its Input and Output



Codebase

Source Code
(C or C++) and build
commands for Clang

OR

Binary via
GhiLift

LLVM IR
“.ll” file

DMC Tool
(“Detection of
Malicious Code”)

Tool output:
Sensitive source-to-sink
flows, along with auxiliary
dataflow analysis

Human
adjudication

List of sensitive sources
and sensitive sinks

Conclusion and Team

Will Klieber

Software Security Engineer

Lori Flynn

Senior Software Security Researcher

David Svoboda

Senior Software Security Engineer

Matt Wildermuth

Assistant Security Researcher

Ruben Martins

Assistant Research Professor, CMU - Computer Science Department

Our tool detects potentially malicious code by tracing the flow of sensitive information and auxiliary information.

Contact: info@sei.cmu.edu

Tool available at:
https://github.com/cmu-sei/dmc

This release includes the source code, a Docker file, tests, documentation, and a demo.

We'd appreciate any feedback if you try out the tool!