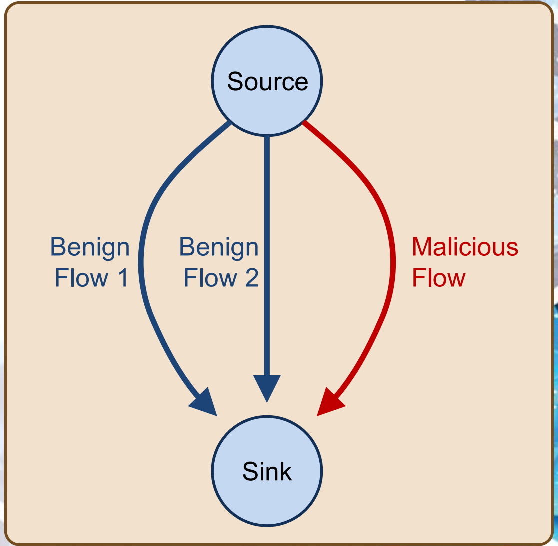
Department of Defense

Weapon Systems Software Summit

Detection of Malicious Code:
Taint Flow Analysis for Weapons Systems Software



Lori Flynn and Will Klieber

December 2024

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.

DM24-1172

Introduction

- 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`

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

Applicability

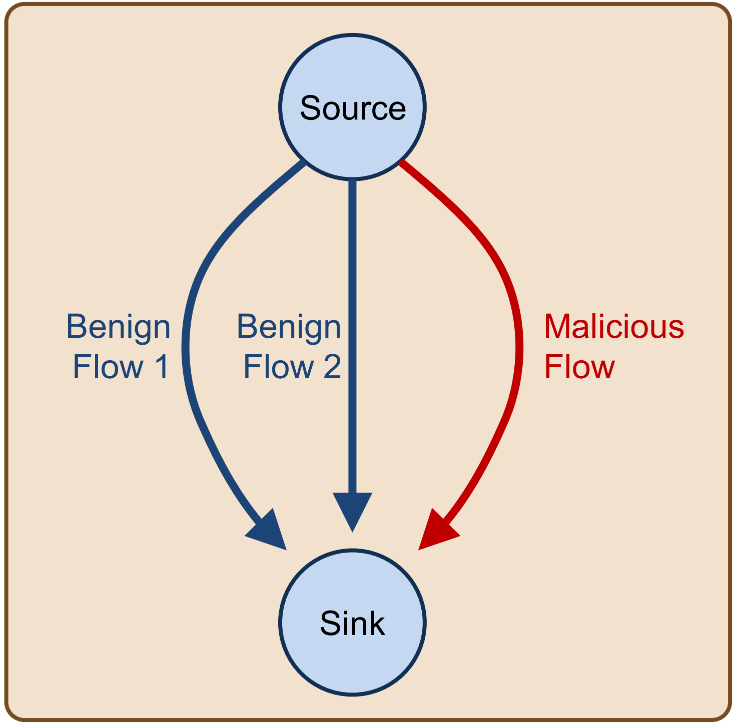
Method is applicable to applications on:

- embedded systems such as weapons systems, medical systems, etc.
- Linux OS
- Windows OS
- other systems

PhASAR

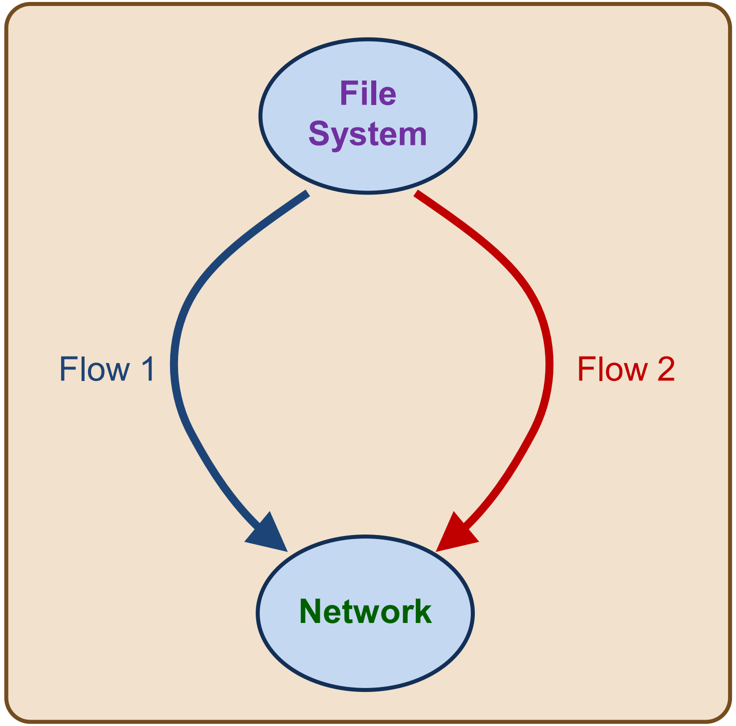
- Initially, we tried building on PhASAR (a static-analysis framework based on LLVM)
- Unfortunately, PhASAR ended up having trouble as we scaled to real-world codebases:
  - Took 15 minutes to analyze `dos2unix` (a very small program, approx. 4,000 lines of code)
  - Ran out of memory (with 24 GB of 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 IFDS analysis, at the cost of less context sensitivity.
  - Much faster and less memory-intensive. Can analyze `git` (approx 275,000 LoC) in just a few minutes with memory usage under 15% on a 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 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

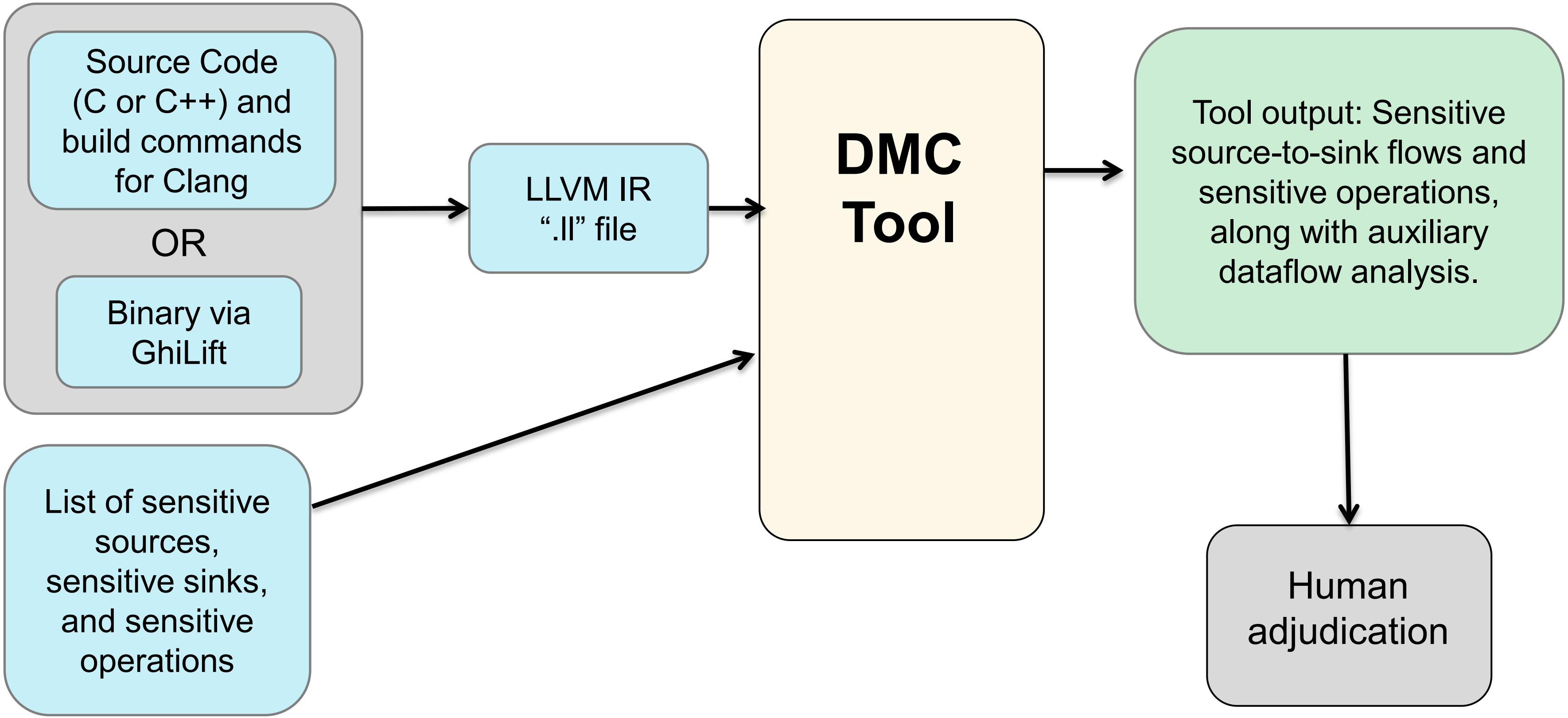
- Flow 1:
  - Source: File system
    - Filename is specified by user.
  - Sink: Network
    - IP: 127.0.0.1
    - Port: 12345
- Flow 2:
  - Source: File system
    - Filename is hardcoded "secrets.txt".
  - Sink: Network
    - IP: 127.0.0.1
    - Port: 12345

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



Creating the List of Sources and Sinks

DMC needs a list of sources and sinks provided by the operating system (OS) and libraries.

- For each function, the LLM* determines:
  - whether the return value is a source
  - which (if any) parameters are sinks
  - which (if any) parameters are sources
    - A parameter is a source if it points to buffer that the API call fills with potentially sensitive data.
- Two methods of generating the list of sources and sinks:
  1. Do upfront analysis of system API functions using DMC scripts and LLM, and/or
  2. Run DMC on the program:
     a. DMC output will indicate which external functions it doesn't recognize.
     b. Feed those function names to the LLM.

* We used GPT-4 https://api.openai.com

Using an LLM to help create list of sources and sinks

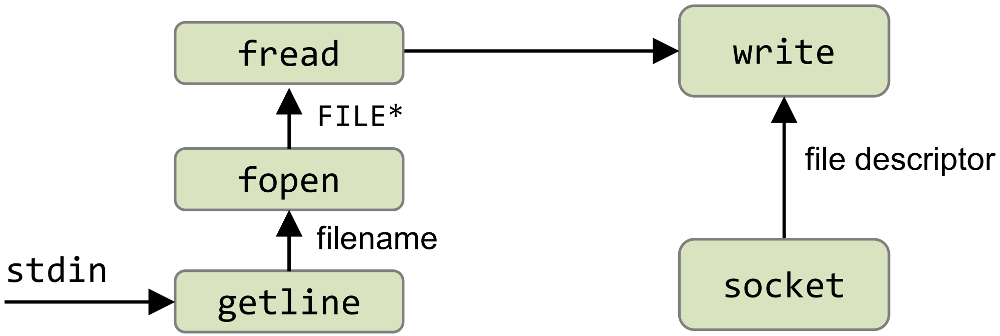
- We used GPT-4, but DMC scripts can be edited to switch (e.g., to on-premise Llama 3 LLM)
- One script interacts with the LLM, one conversation per function
  - Input: a list of system API calls
  - We used just names, since GPT-4 knows common Windows and POSIX API functions
  - For lesser-known OSes, the LLM may need a natural-language description of the function (e.g., the man page for the function)
- Other scripts process the LLM-supplied per-function source/sink data into a policy file listing each API function

Sources and Sinks File (excerpt)

```
dup2        FileSink FileSink -> FileSrc
dup         FileSink -> FileSrc
erfc        -
erf         -
execle      Sink Sink Sink Sink -> none
execlp      FileSink Sink Sink -> none
fgetc       FileSink -> Src
```

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)

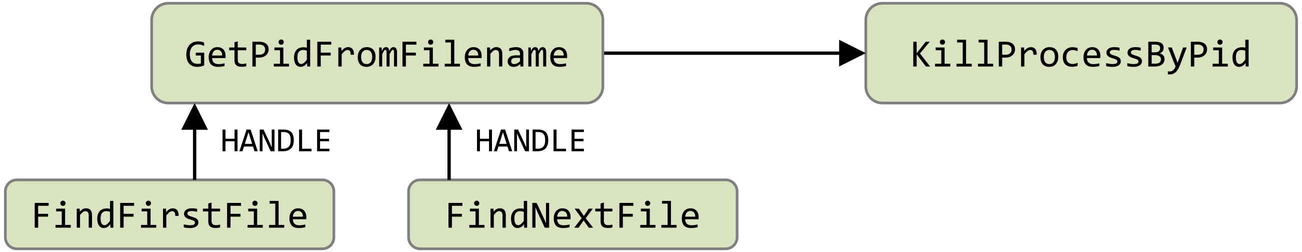

```
[
{"sink": {"func":"write", "callsite":["mal-client-3.c","main",152,21],
    "aux file": [{"func":"socket", "callsite":["mal-client-3.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-3.c","main",139,29],
    "aux file": [{"func":"fopen", "callsite":["mal-client-3.c","main",132,26],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



Wrapper functions

- In the previous slides (mal-client-3.c), there were two separate callsites of the source `fread`, and they were analyzed as distinct sources.
- What if we change the code to add a wrapper function `read_from_file`, so that there is only callsite of `fread` (inside `read_from_file`), and two callsites of `read_from_file`?
- To retain precision, our tool currently requires that `read_from_file` be manually designated as a wrapper function; otherwise, the two flows will be conflated together.

Initial output for mal-client-2.c, without wrappers specified


```
[
{"sink": {"func":"write", "callsite":["mal-client-2.c","main",131,21],
    "aux file": [{"func":"socket", "callsite":["mal-client-2.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-2.c","read_from_file",44,13],
    "aux file": [{"func":"fopen", "callsite":["mal-client-2.c","read_from_file",37,10],
      "aux file": [{"filename":"secrets.txt"} ,
      {"func":"getline", "callsite":["mal-client-2.c","main",106,26], "FILE*":"stdin"},
      {"func":"getline", "callsite":["mal-client-2.c","main",117,30], "FILE*":"stdin"}]}]}]},

...,

{"sink": {"func":"write", "callsite":["mal-client-2.c","main",152,29],
    "aux file": [{"func":"socket", "callsite":["mal-client-2.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-2.c","read_from_file",44,13],
    "aux file": [{"func":"fopen", "callsite":["mal-client-2.c","read_from_file",37,10],
      "aux file": [{"filename":"secrets.txt"} ,
      {"func":"getline", "callsite":["mal-client-2.c","main",106,26], "FILE*":"stdin"},
      {"func":"getline", "callsite":["mal-client-2.c","main",117,30], "FILE*":"stdin"}]}]}]}
]
```

Initial output for mal-client-2.c, with wrappers specified


```
[
{"sink": {"func":"write", "callsite":["mal-client-2.c","main",131,21],
    "aux file": [{"func":"socket", "callsite":["mal-client-2.c","main",65,18]}]},
 "srcs": [{"func":"read_from_file", "callsite":["mal-client-2.c","main",129,32],
    "wrapped":{"func":"fread", "callsite":["mal-client-2.c","read_from_file",44,13],
      "aux file": [{"func":"fopen", "callsite":["mal-client-2.c","read_from_file",37,10],
        "aux file": [{"func":"getline", "callsite":["mal-client-2.c","main",106,26], "FILE*":"stdin"},
                     {"func":"getline", "callsite":["mal-client-2.c","main",117,30], "FILE*":"stdin"}]}]}}]},

...,

{"sink": {"func":"write", "callsite":["mal-client-2.c","main",152,29],
    "aux file": [{"func":"socket", "callsite":["mal-client-2.c","main",65,18]}]},
 "srcs": [{"func":"read_from_file", "callsite":["mal-client-2.c","main",149,40],
    "wrapped":{"func":"fread", "callsite":["mal-client-2.c","read_from_file",44,13],
      "aux file": [{"func":"fopen", "callsite":["mal-client-2.c","read_from_file",37,10],
        "aux file": [{"filename":"secrets.txt"} ]}]}}]}
]
```

Flow paths

- 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).

```
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.  }
```

Implicit Flow vs Explicit Flow

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

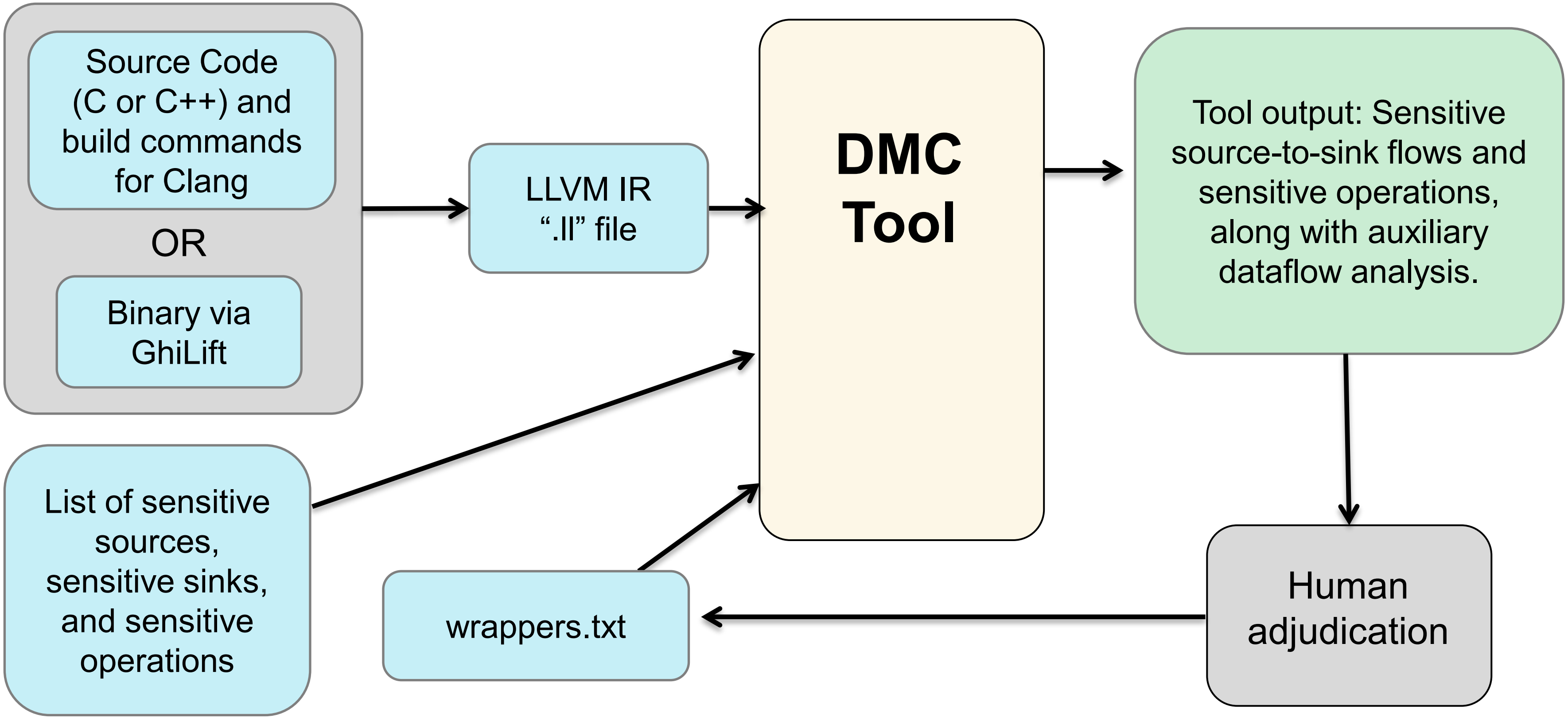
We do not consider implicit flows in this project.

- 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 thru an implicit flow.

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

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

Diagram of our tool, with its input and output



Ideas for building on this work

1. **Incorporate AI to prioritize flows by likelihood of being malicious**
2. **Hard research problem: Develop methods for users to more easily adjudicate categories of flows within a project as benign or not**
3. **Extend the variety of programming languages DMC can handle**
4. **Design for maintenance of previous adjudications of benign flows**, to speed analysis of future versions of the codebase
5. **Extended conditionals analysis**
6. **Test with a variety of embedded systems** (creating reusable sources+sinks lists)
7. **Increase precision**
8. **Integrate with other tools/systems** for malware detection and other app testing
9. **Additional ideas?**

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

Accomplishments, Contact Info, and Next Steps

Accomplishments

DMC prototype tool can detect two types of malicious code:

a. Exfiltration of sensitive information
b. Timebombs/logic bombs, RATs, etc.

It can be run on applications for a variety of systems.

Release includes:

- Code
- Tests
- Dockerfile for containerization
- Demo
- Documentation

Contact

Will Klieber (PI) weklieber@sei.cmu.edu

Lori Flynn lflynn@sei.cmu.edu

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

Ideas for future work include:

1. Incorporate AI to identify flows
2. Develop methods for users to more easily adjudicate categories of flow
3. Extend to more coding languages
4. Extend conditionals analysis

Additional Details

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);
}
```

Wednesday Morning Sessions

8:00 - 9:15 am Panel: PMOs and Organic Software Teams: Developing Software Together	8:00 - 9:15 am Panel: Workforce Innovations to Recruit and Retain Software and Cyber Professionals	8:00 - 9:15 am Panel: Agile Coaches Corner: Ask Anything Agile
Break 9:30-10:00		
Connection Corner 10:00-10:30		
10:30 - 11:15 am Bowlby Armaments Center Software Factory	10:30 - 11:15 Bochenek Models to Code for Sw Assurance	10:30 - 11:15 am Yasar Harmonizing Agile, DSO, and AI for Systems Development

QR Code for App

CONNECTION CORNER

Open to all during exhibit hours!

Putt Putt!

Cornhole!

Jenga!

Golf Simulator!

Scrabble Competition!

Focused subject schedule

Tuesday	
1200	Field Maintainers
1300	Depot Maintainers
1700	Software by Design
1800	AI/ML
Wednesday	
0700	Software Assurance
1000	Agile & Software Workforce
1130	Women in DoD
1230	Logisticians
1700	DIB/Supply Chain
Thursday	
0700	Software Modernization
1115	RDML Grace Hopper Award Celebration w/SWEG
1200	Sustainment Data
1300	OIB

Enhance your experience by networking with other DoD Maintenance Symposium attendees!