Auto-Labeling for Fun and Profit

Matt Elder Bill Harris
University of Wisconsin - Madison
{elder, wrharris}@cs.wisc.edu

CS 736: Advanced Operating Systems Final Presentation

Outline

DIFC

DIFC: Decentralized Information Flow Control

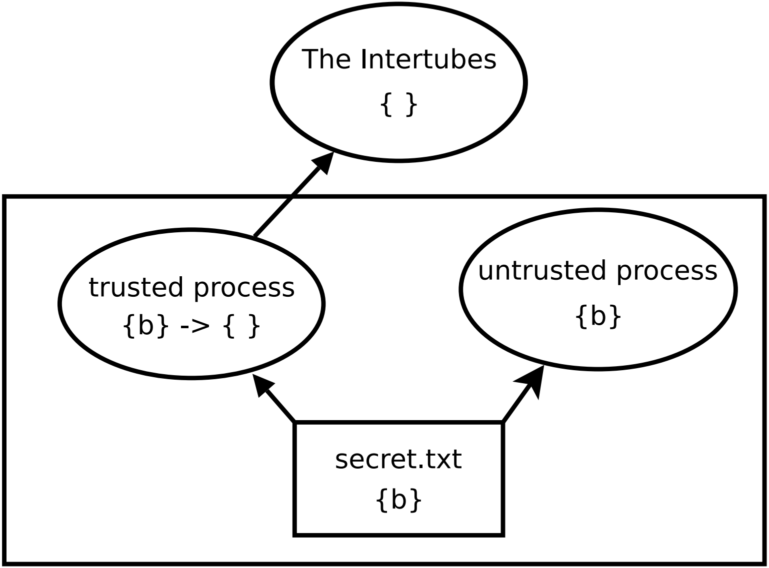
Approaches, granularities of DIFC:

- The variable level, enforced by static analysis and type systems (JIF, JFlow), taint analysis
- The process, system object level, enforced by OS mechanisms (Asbestos, HiStar, Flume)

DIFC Mechanisms

- System maintains a security label for every process, file.
- Label is a set of tags: randomly generated integers.
- An IPC instance from p to q depends on the labels of p and q.
- Processes can:
  - Create tags
  - Add or subtract tags from labels
  - Add or subtract privileges from processes

A toy example


The Intertubes
{ }
trusted process
{b} -> { }
untrusted process
{b}
secret.txt
{b}

A DIFiCult problem

- DIFC programmers must be conscious of the current label sets and privilege sets of all processes and files.
- Guarantees that that code satisfies security and functionality requirements are difficult to think about.

Our solution

1. Read a high-level specification describing a security policy.
2. Represent a specification as constraints over label variables.
3. Solve the constraints.
4. Rewrite source code, incorporating solution.

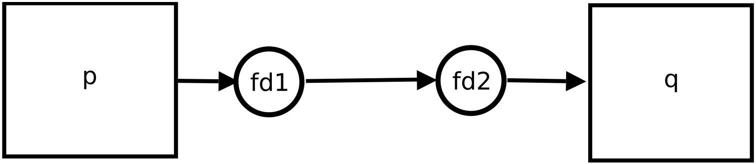
Outline

1. Introduction
2. Flume
3. Boat
   - Dataflow Constraints
   - Specification Constraints
4. Evaluation
5. Conclusion

Flume background

- We’ve implemented our programming system on top of Flume
- Flume is a complete DIFC system; today, we focus on sockets

Sockets illustrated



```
flume_socketpair(&fd,
    &their_tok)
...
spawn(&their_tok, ...)
...
write(fd)
```

```
fd = flume_claim_socket(tok)
...
x = read(fd)
```

Flume Basics

- Flume always ensures that the file descriptor label equals the process label.
- If P writes to Q, it ensures that $S_P \subseteq S_Q$.
- Processes with privileges may add or drop security tags.

Outline

1. Introduction
2. Flume
3. Boat
   - Dataflow Constraints
   - Specification Constraints
4. Evaluation
5. Conclusion

Solution: A (slightly) deeper look

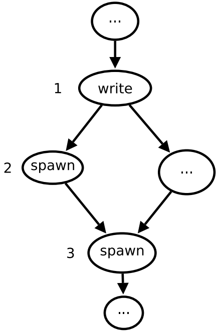
A multi-phase solution:

1. Generate constraints relating program points
2. Generate constraints representing the user specification
3. Munge those constraints together
4. Solve the constraints
5. Rewrite code reflecting the solution

Intraprocess analysis

```
pid = spawn()
if (...)
  write(...)
else
  ...

spawn(...)
```



1. For every relevant program point, generate a variable.
2. Use standard dataflow (compile-time) analysis to determine relevant pairs (u, v).
3. In above example: $(1,2),(1,3),(2,3)$.

Intraprocess constraints

- S^+ is the set of positive privileges
- S^- is the set of negative privileges
- For each pair, generate constraints:

$$v \subseteq u \cup S^+$$
$$u \subseteq v \cup S^-$$

Program Annotations

Programs must be annotated with extra “advice” for the analysis: a family name for each `spawn` program point and the destination family for each `write`.

- `spawn(arglist)` becomes
  `spawn("familyname", arglist)`
- `write(arglist)` becomes
  `write("familyname", arglist)`

`"familyname"` must be a constant in both instances

Specification Language

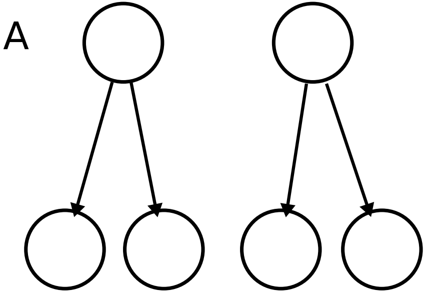
Defines relationships between process families and across child-parent relations.

Examples:

- `A : B`: `A` processes may spawn `B` processes.
- `A -> B`: An `A` process can reach `B` processes.
- `A !-> B`: An `A` process can never reach `B` processes.
- `B !-> B`: Two `B` processes can never reach each other.

Specification Language (more examples)

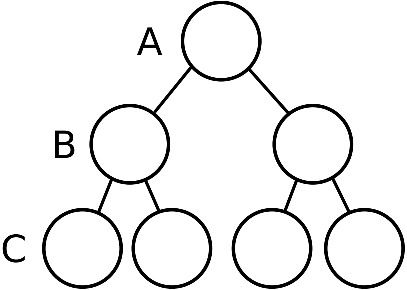
- `A -> *B`: An `A` process can reach its `B` children.
- `A !-> \B`: An `A` process can never reach `B` processes that aren't its children.



Specification Constraint Generation

First, generate a forest of representative processes:

1. Every family is represented by at least one process.
2. Every process that might spawn a `B` process has two `B` children.



Specification Constraint Generation

- If the spec file says that process P in this forest can reach process Q, then generate the constraint $(u \subseteq v)$ for every pair of program points u in P and v in Q where P might write to Q.
- If the spec file says that process P cannot reach Q, then generate the constraint $(u \not\subseteq v)$ for every pair of program points u in P and v in Q.

Solving Constraints

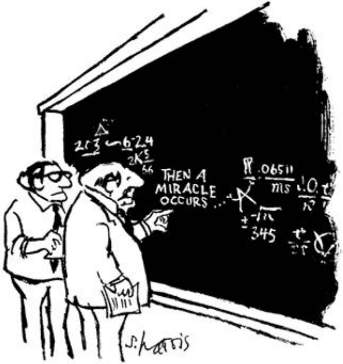
- Now have a set of constraints all of the form $A \subseteq B \cup C$ or $A \subseteq B$ or $A \not\subseteq B$
- Work by Rehof and Mogenson directly implies that a solution is *NP*-hard to find in general
- We "cheat" using extra knowledge from the problem domain

Solving Constraints

1. The constraints from analysis of the control flow graph and the analysis of the specification are combined, and the resultant constraint system is solved:

Solving Constraints

1. The constraints from analysis of the control flow graph and the analysis of the specification are combined, and the resultant constraint system is solved:



"I think you should be more explicit here in step two."

Outline

1. Introduction
2. Flume
3. Boat
   - Dataflow Constraints
   - Specification Constraints
4. Evaluation
5. Conclusion

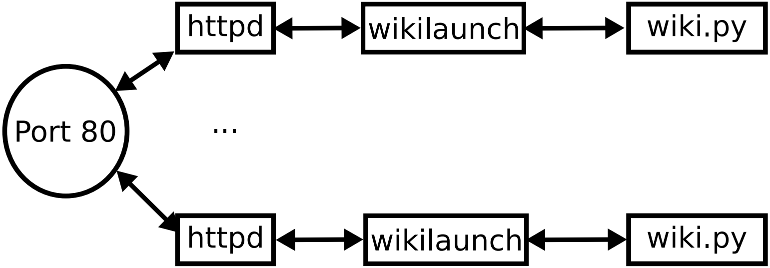
Implementation

Implemented as Boat, a CIL extension. Consists of modules for:

- Analyzing an annotated program
- Parsing a specification
- Solving constraints
- Rewriting code (incomplete... for now)

FlumeWiki

FlumeWiki: a security conscious version of MoinMoin:



Port 80

httpd

wikilaunch

wiki.py

...

httpd

wikilaunch

wiki.py

FlumeWiki code

The relevant snippet from the original cgilaunch.c:

```
rc = flume_socketpair (&input, &fdhandles->val[0], ...);

/* setup CGI's labelset */
S_label = label_alloc (1);
rc = label_set (S_label, 0, S_handle);
labs = labelset_alloc ();
rc = labelset_set_S (labs, S_label);

/* spawn the CGI */
rc = flume_spawn_legacy (labs, fdhandles, ...);
/* send form information to cgi */
if (cgl_form_len ()) {
    rc = write (input, ...);
}
```

BoatWiki specification

```
wikilaunch wiki;
wikilaunch:wiki;

wikilaunch -> wiki;
wiki !<-> wiki;
```

BoatWiki code annotations

```
/* setup CGI’s labelset */
rc = flume_socketpair (&input, &fdhandles->val[0], ...);
rc = flume_spawn_legacy ("wiki", fdhandles, ...);

/* send form information to cgi */
if (cgl_form_len ()) {
  rc = write ("wiki", input, ...);
}
```

Outline

1. Introduction
2. Flume
3. Boat
    - Dataflow Constraints
    - Specification Constraints
4. Evaluation
5. Conclusion

Conclusion

- We have designed a simple, high level specification language that can specify many security policies
- Flume code can be generated efficiently and automatically from these policies

EOP

Thank you for playing! Any questions?

(Appendix) BoatWiki generated code

```
int parent[2] = { 0, -1 };
int child[2] = { 0, -1 };
int per_spawn[2] = {0, -1};
int boat_buf2[1]  = {-1};
...
boat_pre_spawn(parent, child, per_spawn);
rc = flume_spawn_legacy(fdhandles, ...);
tmp___3 = cgl_form_len();

/* send form information to cgi */
if (tmp___3) {
  boat_pre_write(boat_buf2);
  rc = write(input, ...);
```