Nontransitive Policies Transpiled

Mohammad M. Ahmadpanah / Chalmers University of Technology / mohammad.ahmadpanah@chalmers.se
Aslan Askarov / Aarhus University / aslan@cs.au.dk
Andrei Sabelfeld / Chalmers University of Technology / andrei@chalmers.se

Euro SP 2021

Information Flow

Security classes

Public → **Secret**

Flow relation

Flow Relation

Public $\rightarrow$ Secret $\rightarrow$ Top Secret

The argument for transitivity of the flow relation

"Since $A \rightarrow B$ implies permission to move a value $\mathbf{x}$ from [...] class A to [...] class B, and $B \rightarrow C$ implies it is in turn permissible to move $\mathbf{x}$ to [...] class C, an inconsistency arises if $A \nrightarrow C$"

[D. Denning, *A lattice model for secure information flow,* 1976]

The argument assumes that when $\mathbf{x}$ is moved its original classification is lost

Yet, transitivity of the flow relation is not always desirable, especially in coarse-grained settings!

A case for nontransitivity

[Lu & Zhang, CSF 2020]

- Consider a system w/ three components: Alice, Bob, Charlie
  - Alice permits Bob to read her data, but not Charlie
  - Bob permits his data to be read by Charlie
- We have $A \rightarrow B$, $B \rightarrow C$, but $A \nrightarrow C$
  - If Bob's component sends anything to Charlie, it must be *only* Bob's information, and not Alice's

```
Alice {
    data;
    main() {
        Bob.receive(data);
        Bob.good();
        Bob.bad();
    }
}
Bob {
    data1;
    data2;
    receive(x) { data1 = x; }
    good() { Charlie.receive(data2); }
    bad() { Charlie.receive(data1); }
}
Charlie {
    data;
    receive(x) { data = x; }
}
```

OK

A case for nontransitivity

[Lu & Zhang, CSF 2020]

- Consider a system w/ three components: Alice, Bob, Charlie
  - Alice permits Bob to read her data, but not Charlie
  - Bob permits his data to be read by Charlie
- We have $A \rightarrow B$, $B \rightarrow C$, but $A \not\rightarrow C$
  - If Bob's component sends anything to Charlie, it must be *only* Bob's information, and not Alice's

```
Alice {
    data;
    main() {
        Bob.receive(data);
        Bob.good();
        Bob.bad();
    }
}
Bob {
    data1;
    data2;
    receive(x) { data1 = x; }
    good() { Charlie.receive(data2); }
    bad() { Charlie.receive(data1); }
}
Charlie {
    data;
    receive(x) { data = x; }
}
```

A case for nontransitivity

[Lu & Zhang, CSF 2020]

- Consider a system w/ three components: Alice, Bob, Charlie
  - Alice permits Bob to read her data, but not Charlie
  - Bob permits his data to be read by Charlie
- We have $A \rightarrow B$, $B \rightarrow C$, but $A \not\rightarrow C$
  - If Bob's component sends anything to Charlie, it must be *only* Bob's information, and not Alice's
- Lu & Zhang's approach
  - nontransitive flow relation $\rightarrow$
  - new definition of nontransitive noninterference (NTNI) that generalizes standard NI
  - specialized type system for enforcement
  - new proof of soundness

```
Alice {
    data;
    main() {
        Bob.receive(data);
        Bob.good();
        Bob.bad();
    }
}
Bob {
    data1;
    data2;
    receive(x) { data1 = x; }
    good() { Charlie.receive(data2); }
    bad() { Charlie.receive(data1); }
}
Charlie {
    data;
    receive(x) { data = x; }
}
```

OK

BAD

This paper

Standard (transitive) information flow machinery can enforce nontransitive noninterference

Two steps

Program transformation

Lattice encoding

The answer to “dropping the lattice assumption” is … power lattices :-)

- Based on the insight from complex label models such as DLM [Myers & Liskov, 1998] and DC [Stefan et al., 2011]

From Nontransitive to Transitive

```
Alice {
    data;
    main() {
        Bob.receive(data);
        Bob.good();
        Bob.bad();
    }
}
Bob {
    data1;
    data2;
    receive(x) { data1 = x; }
    good() { Charlie.receive(data2); }
    bad() { Charlie.receive(data1); }
}
Charlie {
    data;
    receive(x) { data = x; }
}
```

Observation: parts of the component state such as Bob.data1 are used as both sources (inputs to the system) and sinks (outputs)

Step 1: rewrite the program so that sink and source usage is separated

- source vars (inputs) are never modified (read-only)
- sink vars (outputs) are never read (write-only)
- all other updates are done in temp variables

`Bob.data1` is substituted by 3 vars:

```
- Bob.data1         (* contains initial value of Bob.data1  *)
- Bob.data1_sink    (* contains final value of Bob.data1    *)
- Bob.data1_temp    (* for intermediate values of Bob.data1 *)
```

Add initialization/finalization that copy to/from the temp vars

The rewritten program is semantically equivalent to the original (modulo renaming and having 3x more variables in the state)

Example of the Rewriting

```
Alice {
    data;
    main() {
        Bob.receive(data);
        Bob.good();
        Bob.bad();
    }
}
Bob {
    data1;
    data2;
    receive(x) { data1 = x; }
    good() { Charlie.receive(data2); }
    bad() { Charlie.receive(data1); }
}
Charlie {
    data;
    receive(x) { data = x; }
}
```

Before

```
// init
Alice.data_temp := Alice.data;
Bob.data1_temp := Bob.data1;
Bob.data2_temp := Bob.data2;
Charlie.data_temp := Charlie.data;

Bob.data1_temp := Alice.data_temp;
Charlie.data_temp := Bob.data2_temp;
Charlie.data_temp := Bob.data1_temp;

// final
Alice.data_sink := Alice.data_temp;
Bob.data1_sink := Bob.data1_temp;
Bob.data2_sink := Bob.data2_temp;
Charlie.data_sink := Charlie.data_temp;
```

After (with inlining of main for reader’s convenience)

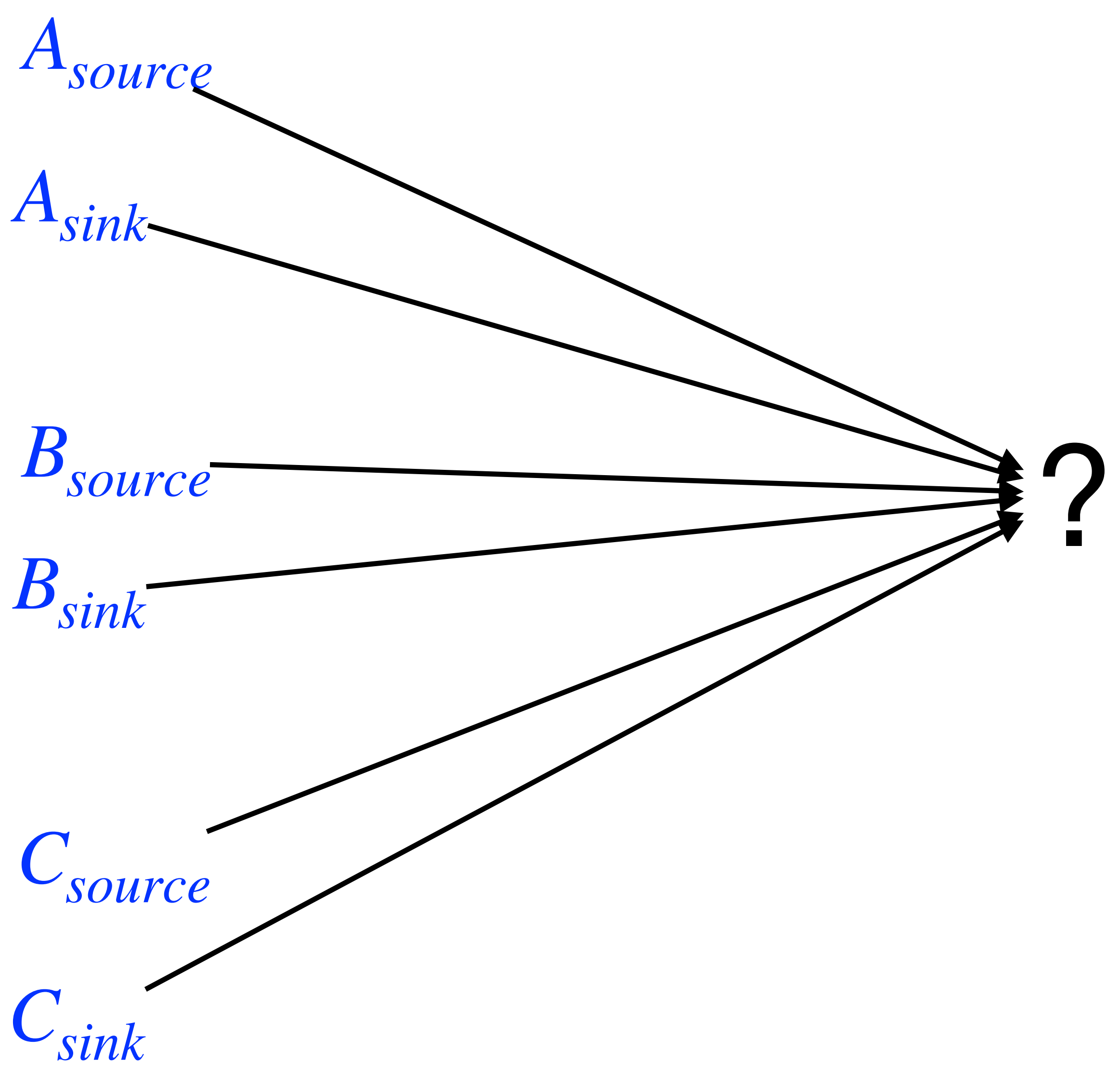
From Nontransitive to Transitive

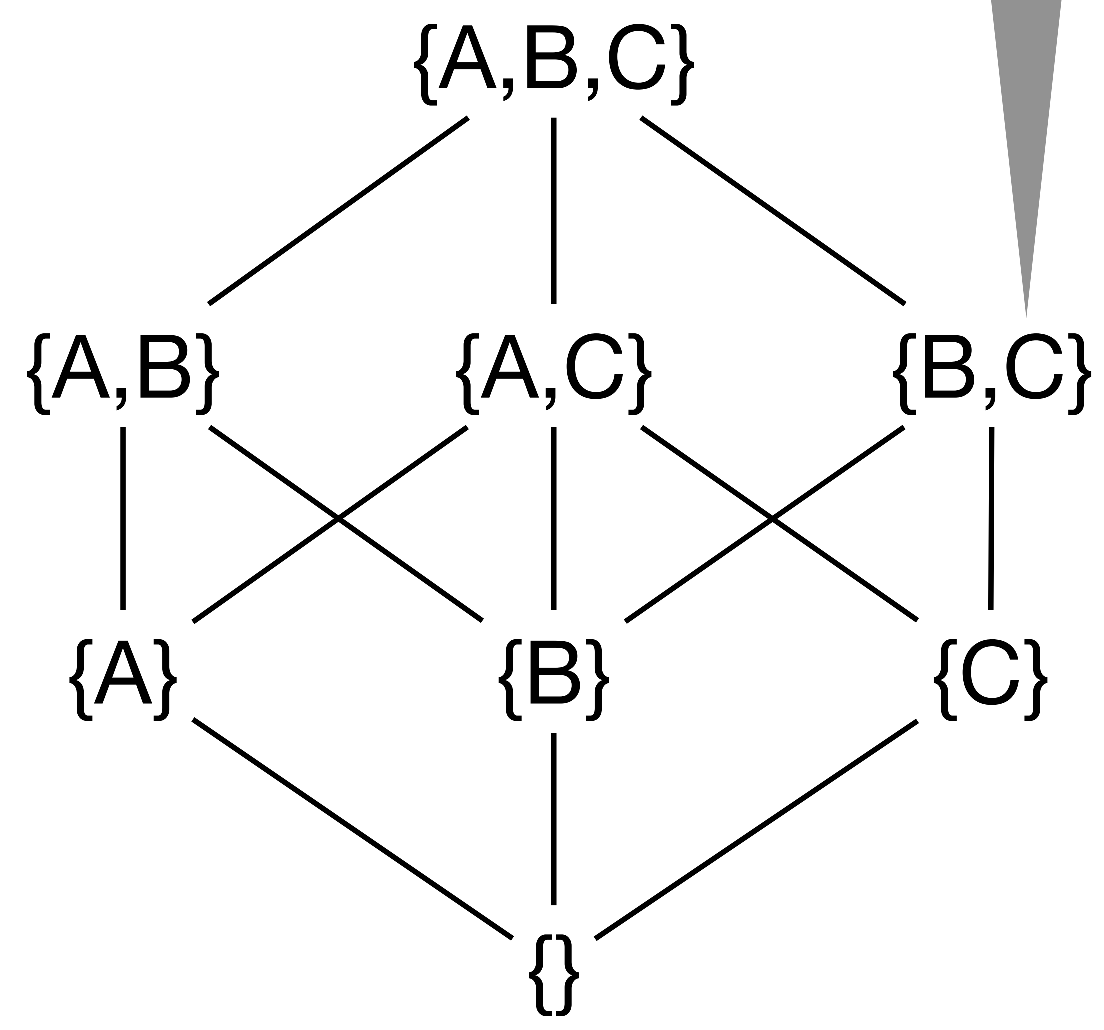
Observation: nontransitive $A \to B$ is really about permitting flows from A's source to B's sink

Step 2: given nontransitive $\to$ relation on components, represent each component by two levels in a powerset-lattice: one level for source and one for sinks

Nontransitive policy: $A \to B, B \to C$

Security class that can read source data of B and C





Standard (transitive) power-lattice

Lattice element $\{x_1, \ldots, x_n\}$corresponds to a security class that can read source data of components $x_1, \ldots, x_n$

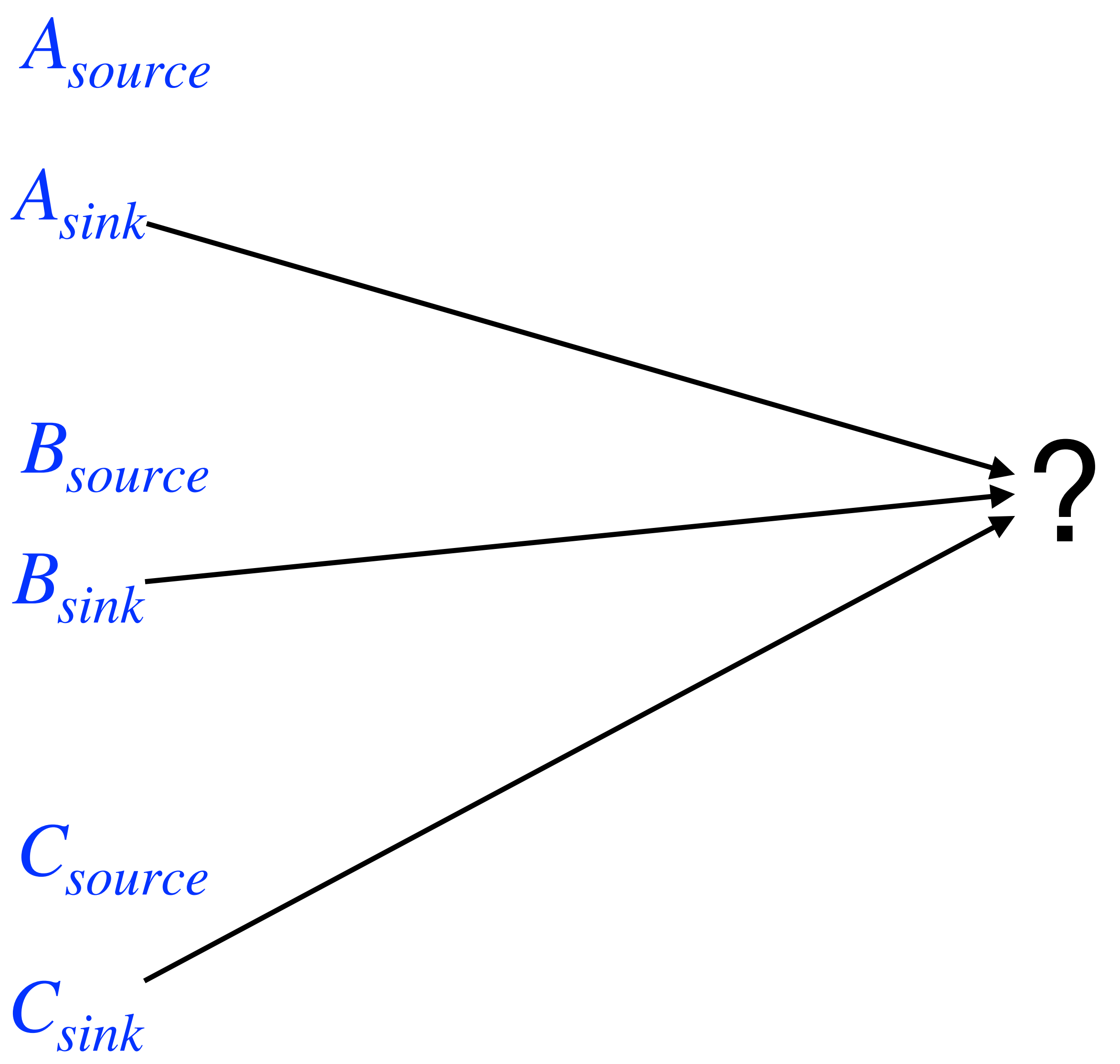
From Nontransitive to Transitive

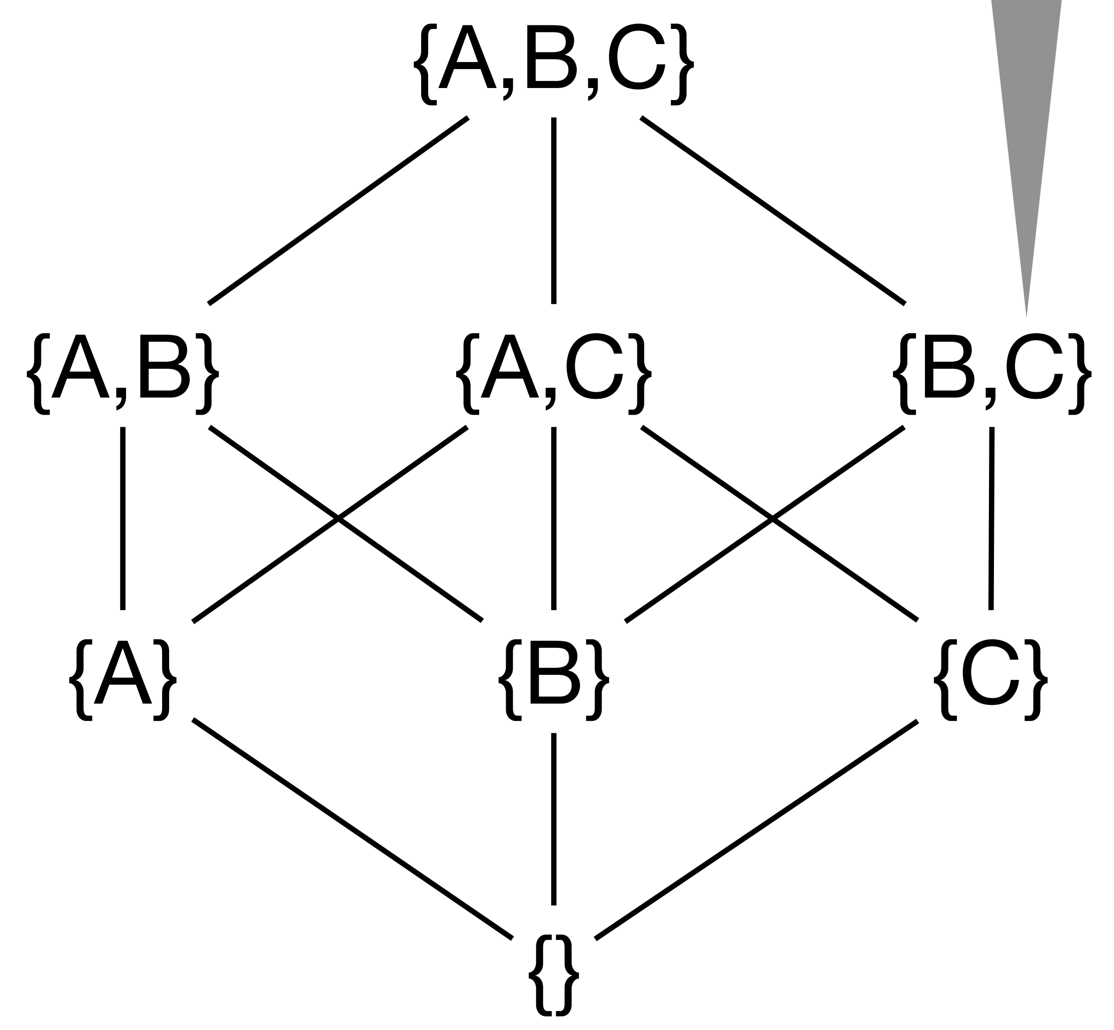
Observation: nontransitive $A \rightarrow B$ is really about permitting flows from A's source to B's sink

Step 2: given nontransitive $\rightarrow$ relation on components, represent each component by two levels in a powerset-lattice: one level for source and one for sinks

Nontransitive policy: $A \rightarrow B, B \rightarrow C$

Security class that can read source data of B and C





Standard (transitive) power-lattice

Lattice element $\{x_1, \ldots, x_n\}$ corresponds to security class that can read source data of components $x_1, \ldots, x_n$

From Nontransitive to Transitive

Observation: nontransitive $A \rightarrow B$ is really about permitting flows from A's source to B's sink

Step 2: given nontransitive $\rightarrow$ relation on components, represent each component by two levels in a powerset-lattice: one level for source and one for sinks

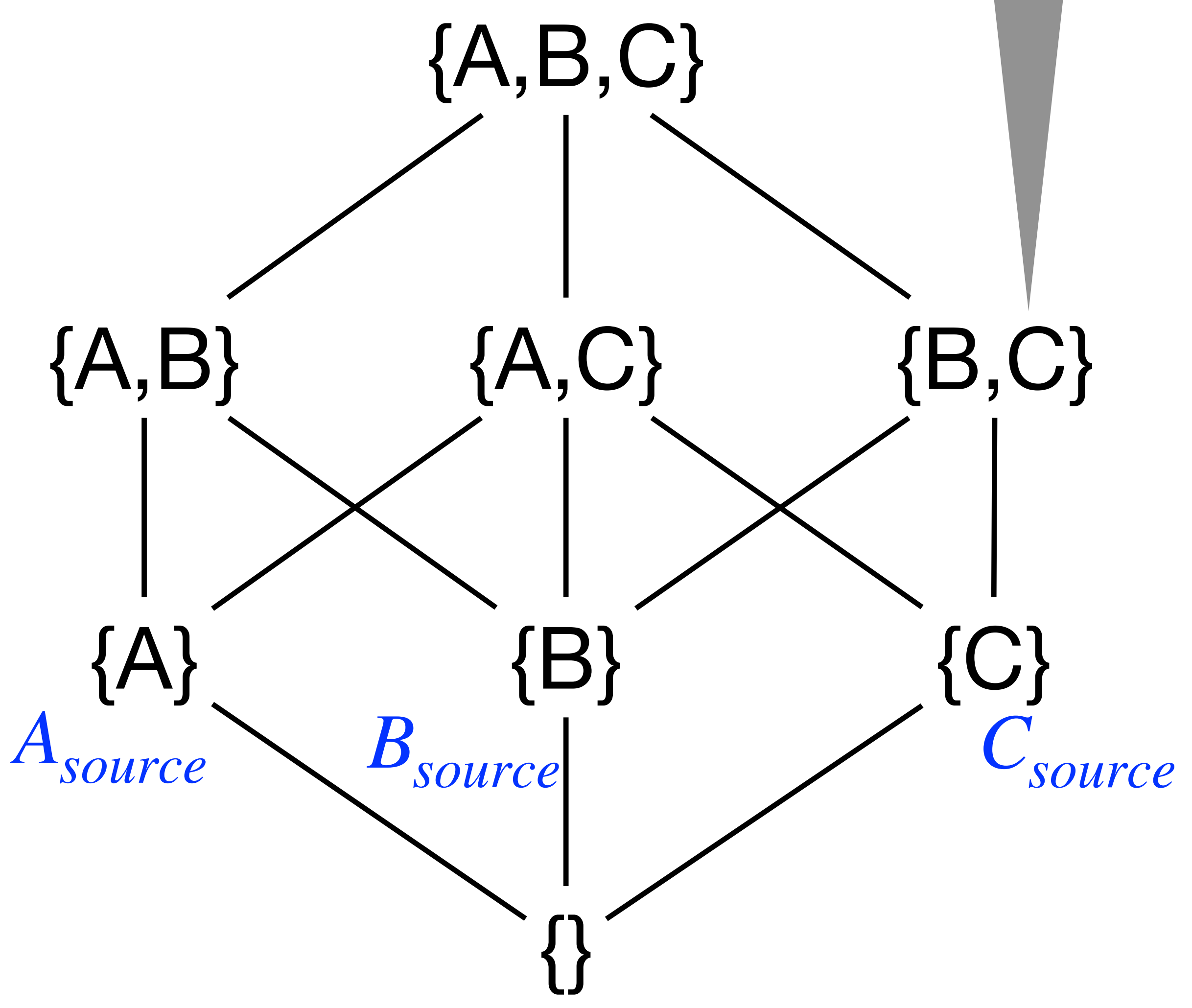
Nontransitive policy: $A \rightarrow B, B \rightarrow C$

Security class that can read source data of B and C

A_{sink}

B_{sink}

C_{sink}



Standard (transitive) power-lattice

Lattice element $\{x_1, \ldots, x_n\}$ corresponds to security class that can read source data of components $x_1, \ldots, x_n$

Theorem

Given a program c and a nontransitive flow relation $\rightarrow$, there is a program c' that is semantically equivalent to c (modulo temp-var rewriting) and a transitive flow relation $\rightarrow'$ such that

$$\text{NTNI}\ (c, \rightarrow) \Leftrightarrow \text{TNI}\ (c', \rightarrow')$$

What's the Theorem good for?

No need to use special type systems for NTNI – just use what's out there!

For the formal calculus

Flow-sensitive type system of [Hunt & Sands, POPL'06] is strictly more permissive than the specialized type system of [Lu & Zhang, CSF'20]

For Java

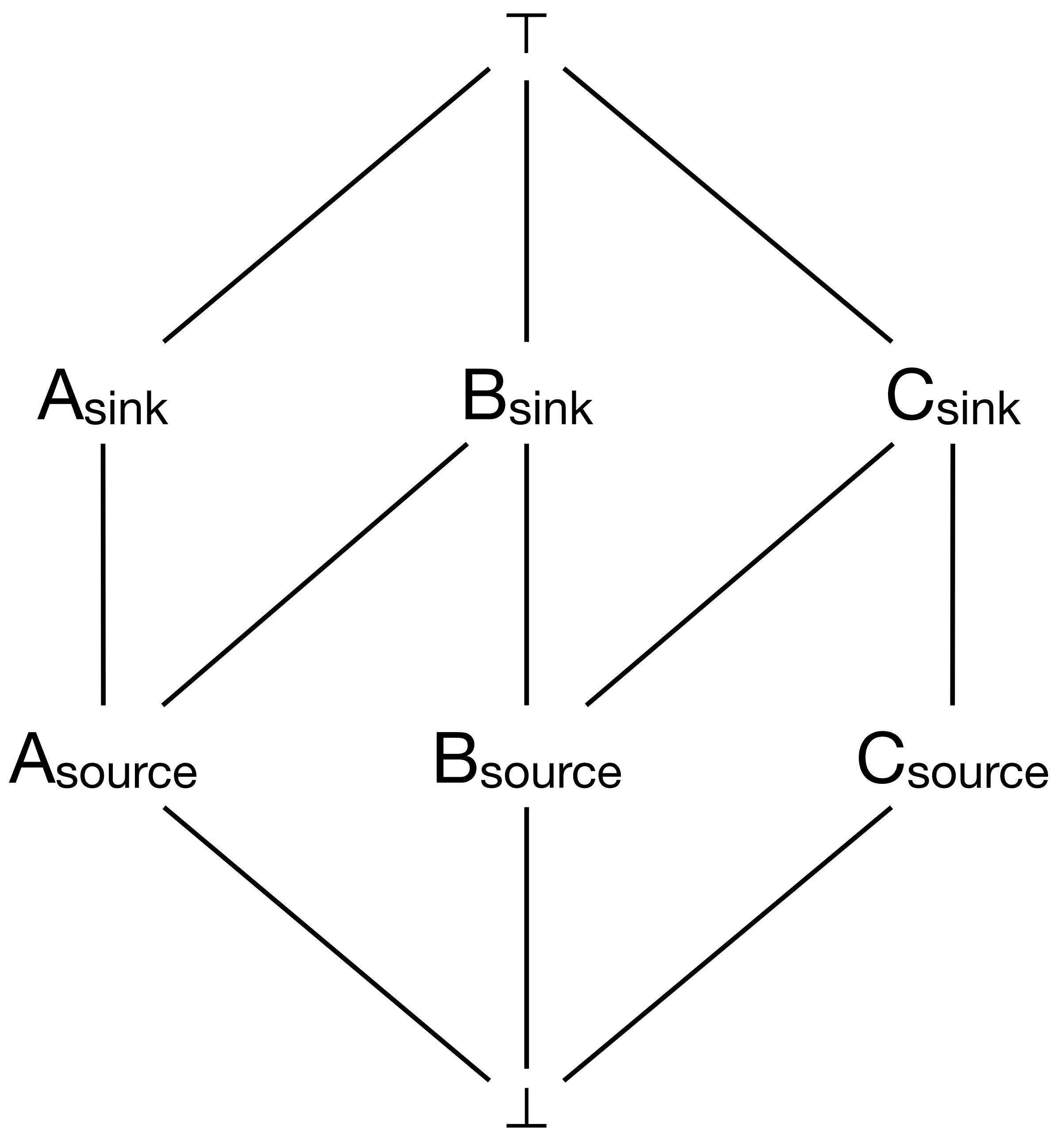
Case studies using JOANA information flow analyzer [Hammer, Snelting, 2020]

```
setLattice e<=A,e<=B,e<=C,A<=AB,A<=AC,B<=AB,
   B<=BC,AB<=ABC,C<=AC,C<=BC,AC<=ABC,BC<=ABC
source Alice.data_source   A
sink   Alice.data_sink     A
source Bob.data1_source    B
sink   Bob.data1_sink      AB
source Bob.data2_source    B
sink   Bob.data2_sink      AB
source Charlie.data_source C
sink   Charlie.data_sink   BC
run    classical-ni
```

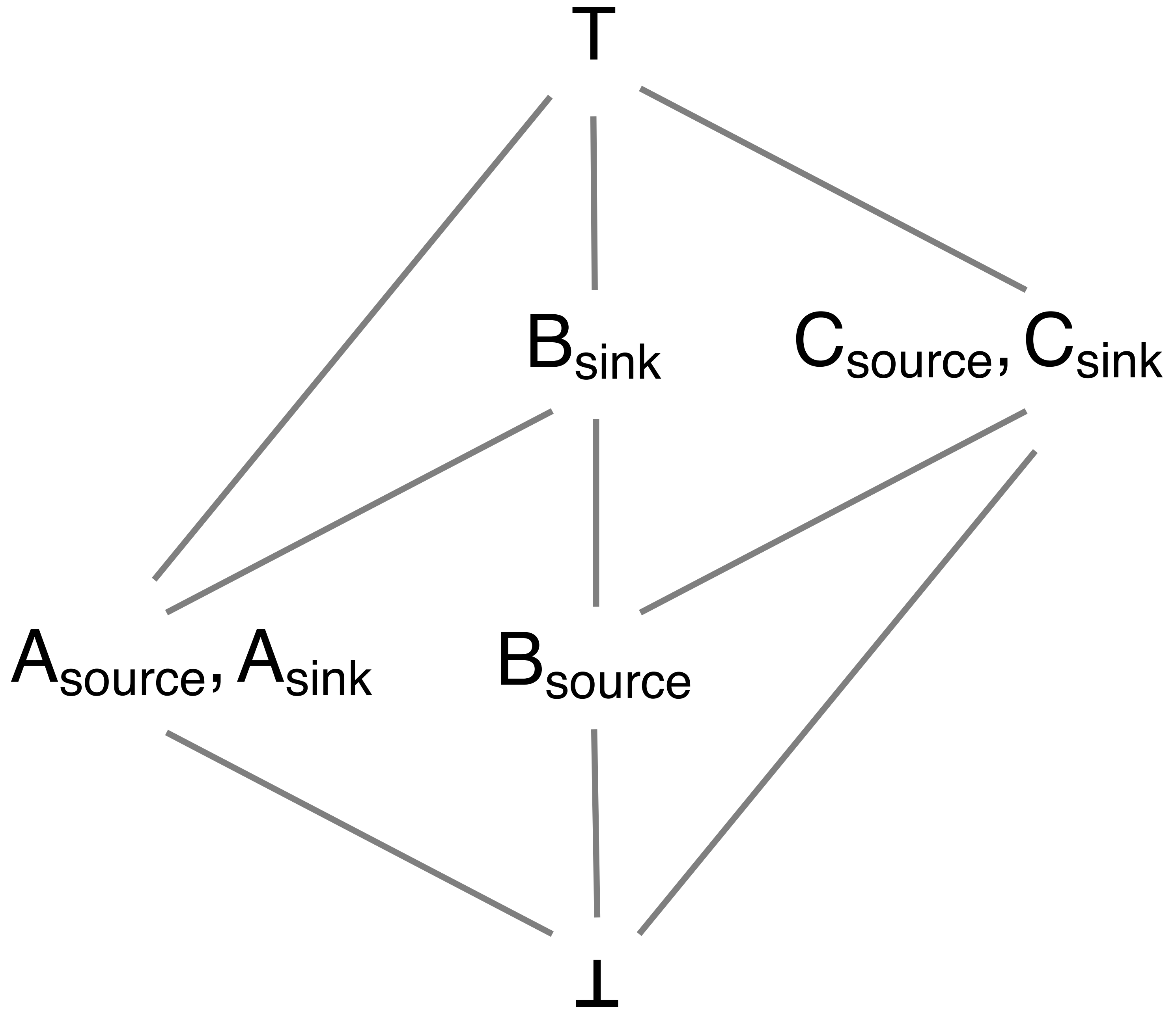
Lattice model input to JOANA for the running example

Alternatives to power-lattice

Nontransitive policy: $A \rightarrow B, B \rightarrow C$



Source-sink lattice via Dedekind-MacNeille completion algorithm



Minimal lattice

Takeaways

- We got inspired by Lu & Zhang work on nontransitive noninterference
- Nontransitive policies are interesting and we expect other applications (e.g., social network restrictions on who can view user's post)
- Our paper shows that we can reuse much of the existing info flow machinery to enforce nontransitive policies
- Minimal lattice encoding remains tantalizing
- Paper details:
  - https://www.cse.chalmers.se/research/group/security/ntni/