Information Flow & Side-Channels

Klaus v. Gleissenthall

VRIJE UNIVERSITEIT AMSTERDAM

Where are we?

- We've moved from verification to security
- We want to prove that a program doesn't leak sensitive information
- The notion of non-interference formally defines this property
- We've seen that we can prove non-interference via verification
- Since non-interference requires reasoning about two program executions, we make two copies of the program and prove non-interference using their composition

Where are we?

- We’ve then looked at an <u>alternative approach</u>
- Language-based security
- Here, we use a type-system to rule out leaky programs
- We’ve seen how to type expressions as either attacker

  observable **obs** or secret **sec**

- Let’s now type statements!
- We’ll start with an approach that doesn’t work and then fix it in a second step!
- For the last three lectures, I’ll talk about some of my research, and Robin will

  run a hands-on session on verification for functional programs

Typing Statements

- Next, we need to type statements

Statement: $s, s_1, s_2 \ni \text{Stmt} ::= \text{skip}$ (no-op)

$\quad | \; x := e$ (assignment)

$\quad | \; s_1 \,;\, s_2$ (sequential composition)

$\quad | \; \text{if } b \text{ then } s_1 \text{ else } s_2$ (if)

$\quad | \; \text{while } b \text{ do } s$ (while)

- We define a typing statement $\Gamma \vdash s$
- Read: statement s is well typed (and therefore doesn't leak!)

Quiz:

- Why don't we assign a type to s?

Typing Statements

- Let's start with skip!

$$\frac{}{\Gamma \vdash \text{skip}}$$

- Skip is always well-typed
- Next, assignments

$$\frac{\Gamma \vdash x : \ell_1 \quad \Gamma \vdash e : \ell_2 \qquad \ell_2 \sqsubseteq \ell_1}{\Gamma \vdash x := e}$$

- Reminder: a $\sqsubseteq$ b, iff b <u>is at least as confidential</u> as a. **obs** $\sqsubseteq$ **sec** **sec** $\not\sqsubseteq$ **obs**
- Makes sure that we can't assign a confidential value to a non-confidential variable
- In general: values can't flow <u>downwards</u> in the lattice

Typing Statements

$$\frac{\Gamma \vdash x : \ell_1 \quad \Gamma \vdash e : \ell_2 \qquad \ell_2 \sqsubseteq \ell_1}{\Gamma \vdash x := e}$$

obs $\sqsubseteq$ **sec** **sec** $\not\sqsubseteq$ **obs**

Quiz:

- Let's say $\Gamma(x) =$ **obs** and $\Gamma(y) =$ **sec**
- Does $\Gamma \vdash y := x$ hold ?
- Does $\Gamma \vdash x := y$ hold ?
- Does $\Gamma \vdash x := x + 1$ hold ?
- Does $\Gamma \vdash y := y + 1$ hold ?

Typing Statements

- Next, let's do sequential composition $s_1 ; s_2$

$$\frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash s_1 ; s_2}$$

- For a composition $s_1 ; s_2$ to be well-typed, each individual statement needs to be well-typed

- For an if-statement, we get:

$$\frac{\Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\text{if b then } s_1 \text{ else } s_2}$$

Typing Statements

- Is this type system correct? Let's try it out!
- Remember, for statement s, if $\Gamma \vdash$ s, then s should be non-inteferent

Quiz:

```
x := secret;
y := x;
obs := y
```

- Let's look again at our example from before
- Since the program is insecure, the check should fail
- Let's say Γ(x)= **sec** and Γ(y)= **obs**
- Does the example type-check?
- Which check fails?

Typing Statements

- Our type system rejects the program due to the direct flow from x to y

Quiz:

```
x := secret;
obs := 0;
if (x=1) { obs := 1;}
```

- Next, let's look at our second example
- Let's set $\Gamma(x) = \mathbf{sec}$, $\Gamma(\mathbf{obs}) = \mathbf{obs}$
- The program is insecure, the check should fail
- Does the example type-check?

Typing Statements

- Our type system rejects the program due to the direct flow from x to y

Quiz:

```
x := secret;
obs := 0;
if (x=1) { obs := 1;}
```

- Next, let's look at our second example
- Let's set $\Gamma(x)$= **sec**, Γ(**obs**) = **obs**
- The program is insecure, the check should fail
- Does the example type-check?
- Out type system <u>fails</u> to detect the indirect flow
- We need to fix it!

Program Counter Label

- To fix the problem, we extend our typing judgement to $\Gamma, pc \vdash s$
- Here pc tracks the security label of the program counter
- In particular, if we're in an if-statement or loop that depends on a **sec** variable the pc label is **sec**
- If we assign to a variable, we need to check it is at least as confidential as the program counter
- Let's fix our type system! As before, we have

$$\frac{}{\Gamma, pc \vdash \text{skip}}$$

Fixing the Type System

- Our new rule for assignments is

$$\frac{\Gamma \vdash x : \ell_1 \quad \Gamma \vdash e : \ell_2 \qquad \ell_2 \sqsubseteq \ell_1 \qquad pc \sqsubseteq \ell_1}{\Gamma, pc \vdash x := e}$$

- We check that the variable we assign to is at least as confidential as the program counter

$$\frac{\Gamma, pc \vdash s_1 \qquad \Gamma, pc \vdash s_2}{\Gamma, pc \vdash s_1 \ ; s_2}$$

- Our rule for sequential composition requires both s_1 and s_2 to have the same pc label

Fixing the Type System

Quiz:

$$\frac{\Gamma \vdash x : \ell_1 \quad \Gamma \vdash e : \ell_2 \qquad \ell_2 \sqsubseteq \ell_1 \qquad pc \sqsubseteq \ell_1}{\Gamma, pc \vdash x := e}$$

$\mathbf{obs} \sqsubseteq \mathbf{sec}$

$\mathbf{sec} \not\sqsubseteq \mathbf{obs}$

- Let's say $\Gamma(x) = \mathbf{obs}$ and $\Gamma(y) = \mathbf{sec}$
- Does $\Gamma, \mathbf{obs} \vdash x := y$ hold ?
- Does $\Gamma, \mathbf{obs} \vdash y := x$ hold ?
- Does $\Gamma, \mathbf{sec} \vdash x := 3$ hold ?
- Does $\Gamma, \mathbf{obs} \vdash x := 3$ hold ?
- Does $\Gamma, \mathbf{sec} \vdash y := x$ hold ?

Fixing the Type System

- Let's look at the rule for if-statements

$$\frac{\Gamma \vdash b : \ell \qquad \Gamma, pc \sqcup \ell \vdash s_1 \qquad \Gamma, pc \sqcup \ell \vdash s_2}{\Gamma, pc \vdash \text{if } b \text{ then } s_1 \text{ else } s_2}$$

- Our rules make sure that the program counter is at least as confidential as the label of branch condition b

- Notice, we didn't need to change the rules for $\Gamma \vdash b : \ell$ and $\Gamma \vdash e : \ell$

Fixing the Type System

Quiz:

- Let's look again at our example:

```
x := secret;
obs := 0;
if (x=1) { obs := 1;}
```

- Let's look again at our example:
- Let's set Γ(x)= **sec**, Γ(**obs**) = **obs**
- Does the program type-check, if our initial pc label is **obs** ?

Fixing the Type System

- Last, rule for while loops

$$\frac{\Gamma \vdash b : \ell \qquad \Gamma, pc \sqcup \ell \vdash s}{\Gamma, pc \vdash \text{while } b \text{ do } s}$$

Quiz:

- Does the rule guarantee termination sensitive non-interference?
- Can we fix it so it does?

Type System Guarantees

- For a statement s and environment Γ, if $\Gamma, \textbf{obs} \vdash s$ then s is non-interferent

Quiz:

- The statement of our theorem is not quite precise
- Non-interference requires us to say which variables are **observable** to the attacker
- Which variables should those be?

New Problems: Timing and Cache

- Let's look at another example and see if we've fixed all sources of leakage.

Quiz:

```
x := secret;
if (x≥1) {
    b := a;
    d := b + c; ...
}
```

- Say, we let $\Gamma(x)$= **sec**, $\Gamma(a)$ = **sec**, $\Gamma(b)$ = **sec**, $\Gamma(c)$ = **sec**, $\Gamma(d)$ = **sec**, ...
- Does the program type-check?
- Is it non-interferent?
- Could there still be a problem?

New Problems: Timing and Cache

- Let's look at another example and see if we've fixed all sources of leakage.

Quiz:

```
x := secret;
if (x≥1) {
    b := a;
    d := b + c; ...
}
```

- Could there still be a problem?
- Yes! The **timing** of the computation depends on whether or not $x \geq 1$ holds
- If there are many operations in the conditional branch, an attacker can glean information about the secret from observing the computation's timing

New Problems: Timing and Cache

- Let's add back arrays to our language: $a[e_1] := e_2 \mid x := a[e]$
- Let's assume our typing environment Γ also provides a type for arrays

Quiz:

  - Say, we let $\Gamma(a) = \mathbf{obs}$, $\Gamma(x) = \mathbf{obs}$, $\Gamma(y) = \mathbf{sec}$ and $\Gamma(b) = \mathbf{sec}$,

- Should the following program type check? $y := a[x]$
- What about this program? $y := b[x]$
- What about this program? $x := a[y]$
- What about this program? $y := b[y]$

New Problems: Timing and Cache

- Let's add back arrays to our language: $a[e_1] := e_2 \mid x := a[e]$
- Let's assume our typing environment Γ also provides a type for arrays

Quiz:

- Say, we let $\Gamma(a)=$ **obs**, $\Gamma(x)=$ **obs**, $\Gamma(y)=$ **sec** and $\Gamma(b)=$ **sec**,

- What about this program? y := **sec**; y := b[y]
- Even though this program may seem secure at first, we still might run into trouble
- Using y as an array index places y's value in <u>cache</u>
- Even though an attacker cannot get direct access to it, they can retrieve it indirectly, via a <u>cache timing attack</u>

New Problems: Timing and Cache

- Now we're in a bit of a pickle
- Let's take another look at our new programs

```
x := secret;
if (x≥1) { b := a; d := b + c; …}
```

```
y := sec; y := b[y]
```

- Now, are these programs secure or not?
- They <u>do</u> satisfy non-interference
- Yet, we found that, in a certain sense they leak
- They can't be both secure and insecure at the same time, so what gives?

Attacker Models

- Now we're in a bit of a pickle
- Let's take another look at our new programs

x := **secret**;

if (x≥1) { b := a; d := b + c; …}

y := **sec**; y := b[y]

- Now, are these programs secure or not?
- Security is relative and always depends on our attacker model
- Before, our attacker was only able to observe a subset of program variables upon termination
- Now, we're assuming a more powerful attacker that can observe timing & cache

Attacker Models: Cache and Timing

- We can make our attacker model more formal, by specifying what an attacker can observe
- For a conditional statement: if b then s_1 else s_2
- Our attacker can observe the value of branch condition b under current state σ
- Same for while loops
- For an array access: x := a[e] or a[e] := e_1
- Our attacker can observe the value of e under current state σ

Quiz:

- Are these realistic assumptions on an attacker?

Fixing the Type-System

- How do we have to change the type-system to account for this new attacker model?
- First, we have to fix the rule for if-statements

$$\frac{\Gamma \vdash b : \textbf{obs} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \text{if } b \text{ then } s_1 \text{ else } s_2}$$

- We no longer need the pc label, and our loop condition needs to be observable

$$\frac{\Gamma \vdash e : \textbf{obs} \qquad \Gamma \vdash a : \ell}{\Gamma \vdash a[e] : \ell}$$

- For arrays, we require the index expression to be **observable**

Fixing the Type-System

$$\frac{\Gamma \vdash b : \textbf{obs} \qquad \Gamma \vdash s_1 \qquad \Gamma \vdash s_2}{\Gamma \vdash \text{if } b \text{ then } s_1 \text{ else } s_2}$$

$$\frac{\Gamma \vdash e : \textbf{obs} \qquad \Gamma \vdash a : \ell}{\Gamma \vdash a[e] : \ell}$$

Quiz:

- Say, we let $\Gamma(a)=$ **obs**, $\Gamma(x)=$ **obs**, $\Gamma(y)=$ **sec** and $\Gamma(b)=$ **sec**,

- Does the following program type-check? if (y≥1) { y := y+1;}

- What about this program? y := b[y]

Speculative Execution

- Have ~~we done enough now? Well, for this attacker model, but there's of course~~ more

```
void SHA2_update_last_512(int *input_len, ...)
    if (! valid(input_len)) {/*error*/ ... }
    int len = *input_len;
    int *dst3 = base + len;
    ...
    *dst3 = pad;
```

- This code doesn't leak during normal execution, as an out of bounds read triggers an error
- But during speculative execution, input_len can be out of bounds
- This means len might now contain a **secret**, which can then be leaked through the **memory access**

Speculative Execution

- Let's look at a minimal example of the problem
- Let's assume: $\Gamma(a)=$ **obs**, $\Gamma(b)=$ **obs**, $\Gamma(i)=$ **obs**, $\Gamma(x)=$ **obs**,

```
x := a[i];
_ := b[x];
```

- Let's assume i is within bounds in regular execution
- During speculation, the read may go out of bounds, and **transiently** access a **secret**
- Then, that secret value ends up in the cache through the access in array b
- Even though the computation gets rolled back, the cache modification persists
- and an attacker can retrieve the secret value via a cache timing attack

Speculative Execution

- Let's look at a minimal example of the problem
- Let's assume: $\Gamma(a)=$ **obs**, $\Gamma(b)=$ **obs**, $\Gamma(i)=$ **obs**, $\Gamma(x)=$ **obs**,

```
x := a[i];
_ := b[x];
```

- How can we fix the program?
- We can add a speculation **fence** to stop the **secret** from reaching the second array access

```
x := a[i];
y := fence(x);
_ := b[y];
```

- The speculation fence will ensure that y never contains a transient secret

Speculative Execution: Types

- We now want to prove that a program is speculation safe via a type system
- We first add another level in our lattice, **tsec** for transient secrets such that

$$\mathbf{obs} \sqsubseteq \mathbf{tsec} \qquad \mathbf{tsec} \sqsubseteq \mathbf{sec}$$

$$\mathbf{tsec} \not\sqsubseteq \mathbf{obs} \qquad \mathbf{sec} \not\sqsubseteq \mathbf{tsec}$$

- Then, we can adjust our rules for arrays:

$$\frac{\Gamma \vdash e : \mathbf{obs}}{\Gamma \vdash a[e] : \mathbf{tsec}}$$

Quiz:

- Do we have to adjust the rule for assignments?

Speculative Execution: Types

- We now want to prove that a program is speculation safe via a type system
- We first add another level in our lattice, **tsec** for transient secrets such that

$$\mathbf{obs} \sqsubseteq \mathbf{tsec} \qquad \mathbf{tsec} \sqsubseteq \mathbf{sec}$$

$$\mathbf{tsec} \not\sqsubseteq \mathbf{obs} \qquad \mathbf{sec} \not\sqsubseteq \mathbf{tsec}$$

Quiz:

- What's the rule for x := **fence**(**e**)?

Speculative Execution: Types

- We now want to prove that a program is speculation safe via a type system
- We first add another level in our lattice, **tsec** for transient secrets such that

$$\mathbf{obs} \sqsubseteq \mathbf{tsec} \qquad \mathbf{tsec} \sqsubseteq \mathbf{sec}$$

$$\mathbf{tsec} \not\sqsubseteq \mathbf{obs} \qquad \mathbf{sec} \not\sqsubseteq \mathbf{tsec}$$

Quiz:

- What's the rule for x := **fence**(**e**)?

- Now: <u>Repairing</u> broken programs by inferring where to place **fence** expressions

Read from array a

```
x = a[i₁];
y = a[i₂];
z = x + y;
_ = b[z];
```

Read from array a

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

... compute new index z

Read from array a

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

… compute new index z

… and use z to index into array b

Not safe:

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

Not safe:

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

x and y can transiently contain **secrets**

Not safe:

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

x and y can transiently contain **secrets**

...that are leaked through the **cache**

Problem: **transient secrets** flow to **observable site**

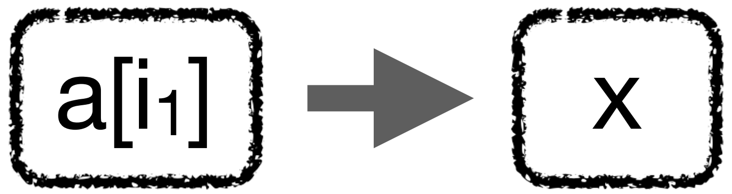
```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

Idea: cut data flow through a **fence**

Build **Data-Flow** Graph

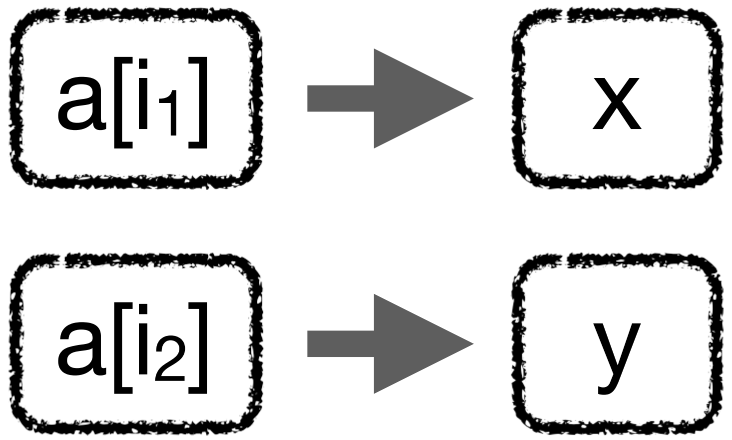
Build **Data-Flow** Graph

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



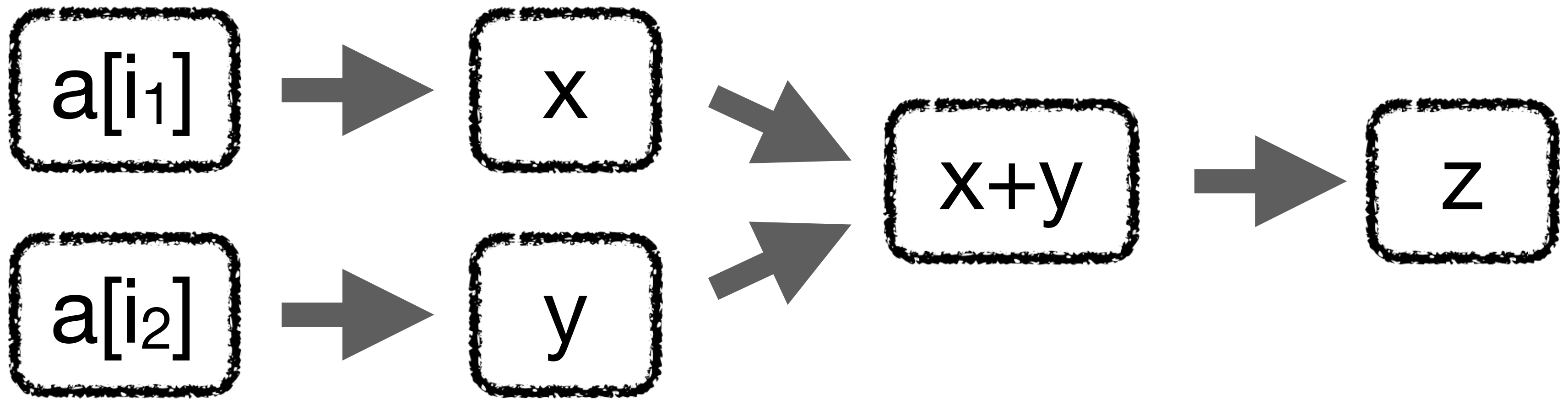
Build Data-Flow Graph

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



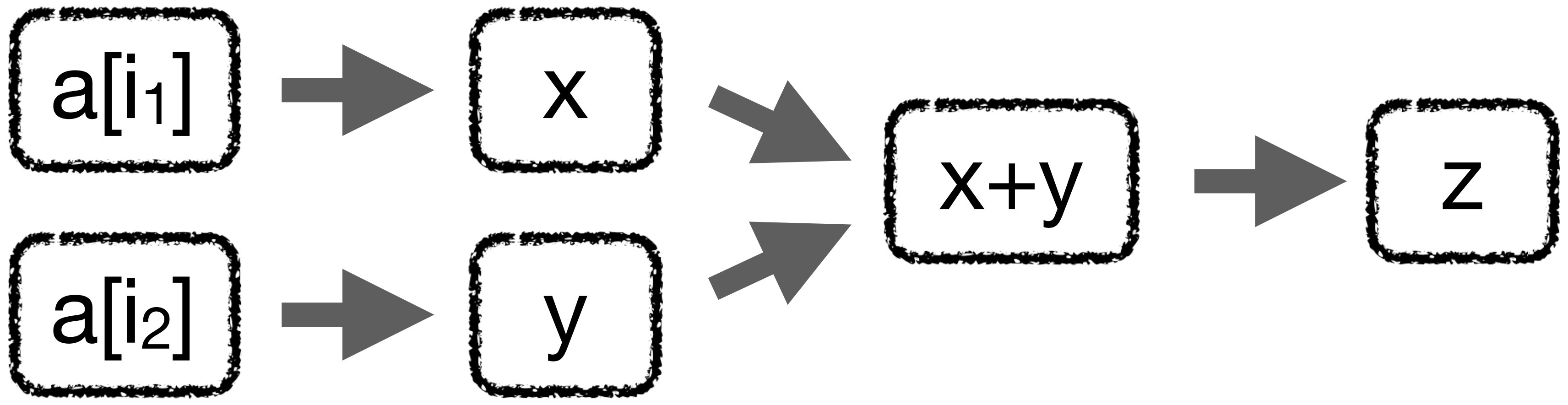
Build Data-Flow Graph

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



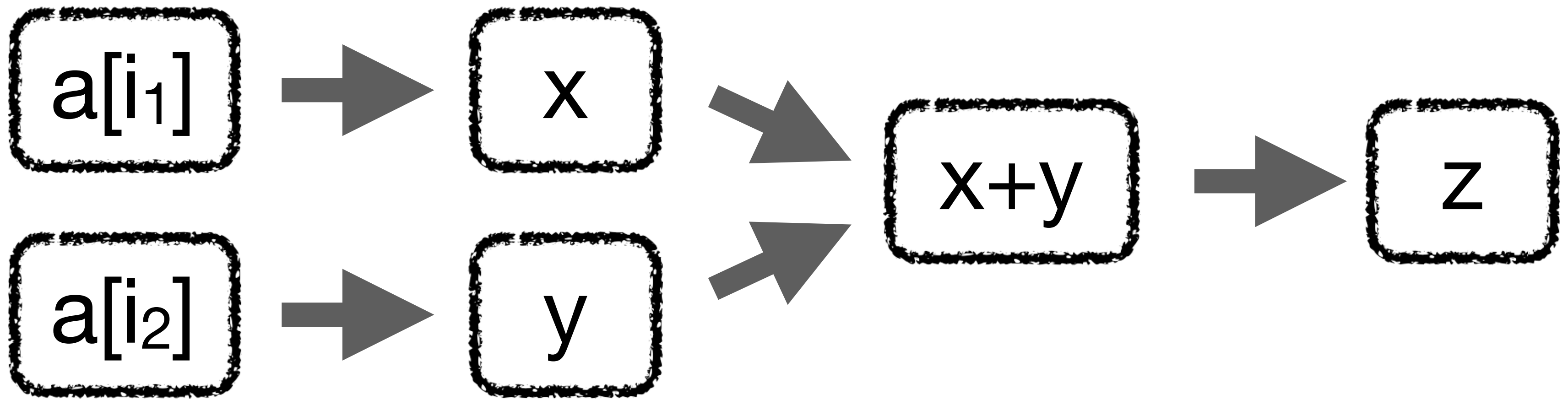
Build Data-Flow Graph

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



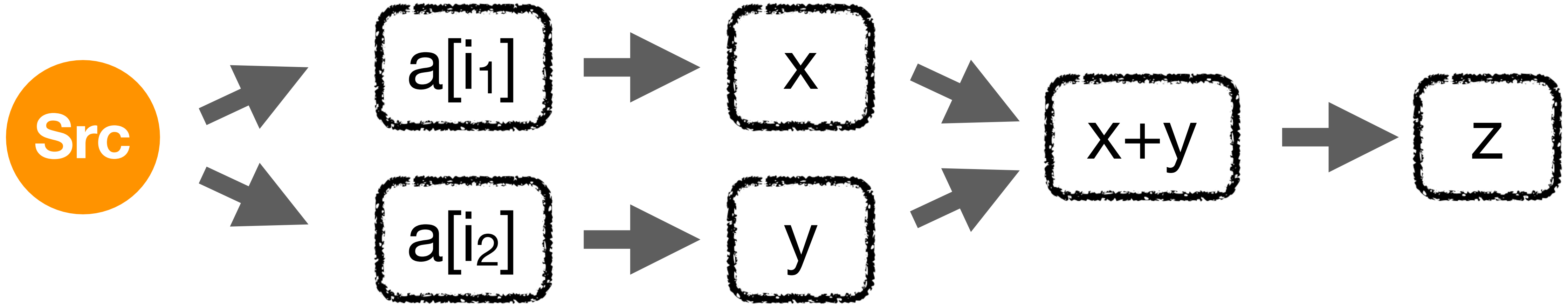
Add Source and Sink nodes

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



Add Source and Sink nodes

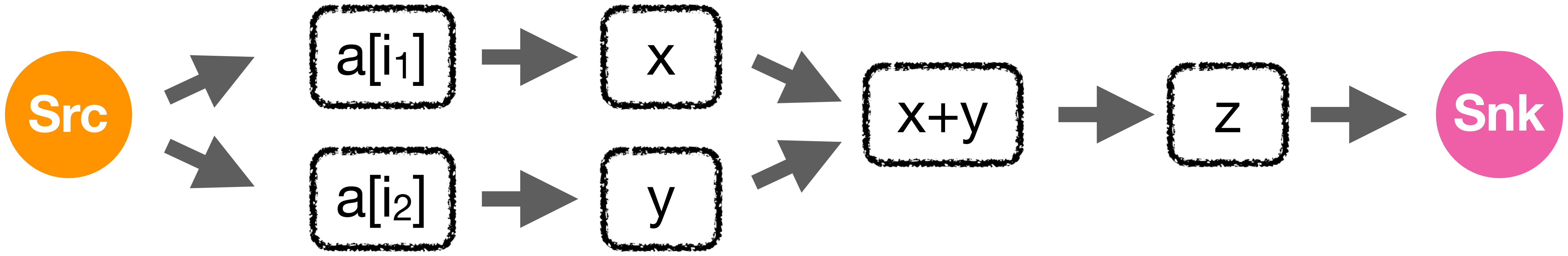
```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



Source introduces **transient secrets**

Add Source and Sink nodes

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

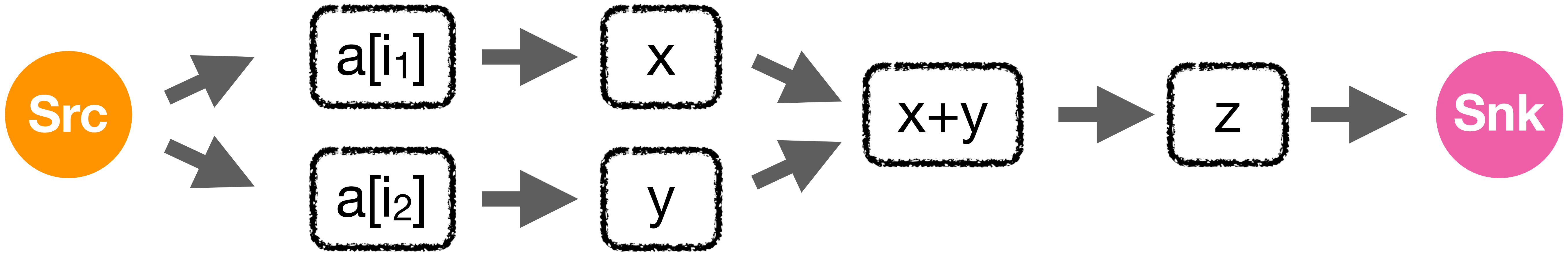


Source introduces **transient secrets**

Sink leaks values through the cache

Idea: Cut data-flow between Source and Sink

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

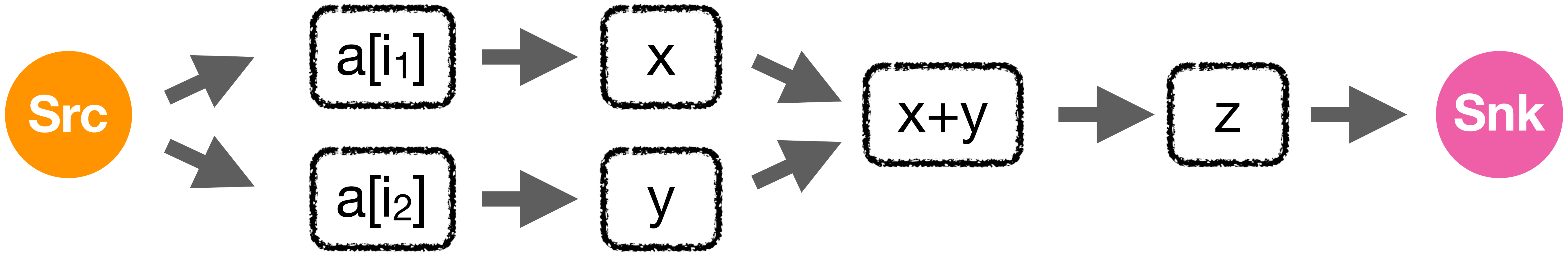


Source introduces **transient secrets**

Sink leaks values through the cache

Idea: Cut data-flow between **Source** and **Sink**

```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```

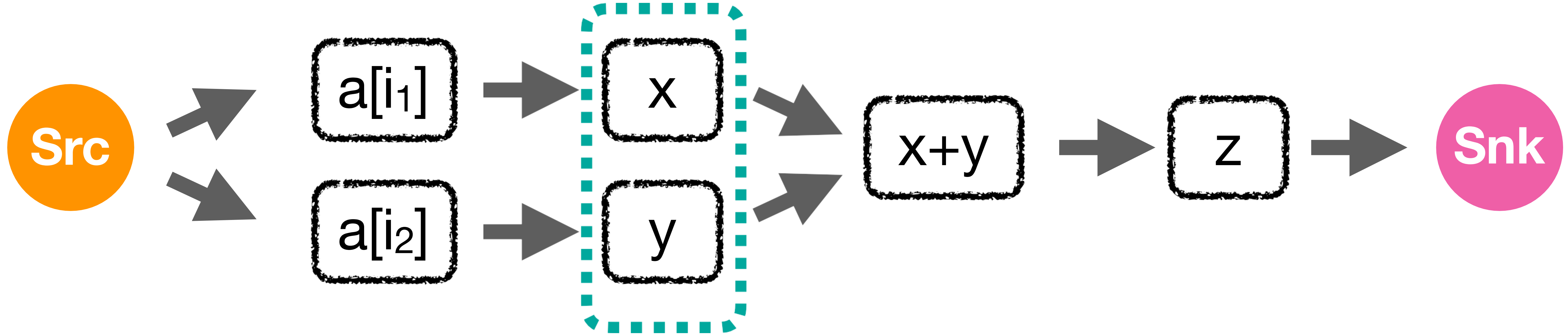


… by removing variables from the graph

… equivalently inserting **fences** into the program

Idea: **Cut** data-flow between **Source** and **Sink**

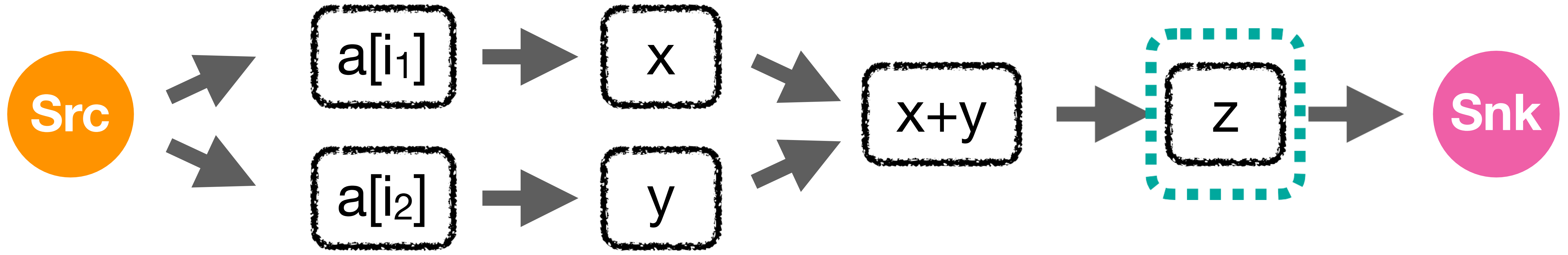
```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



… removing x and y requires two **fences**

Idea: Cut data-flow between **Source** and **Sink**

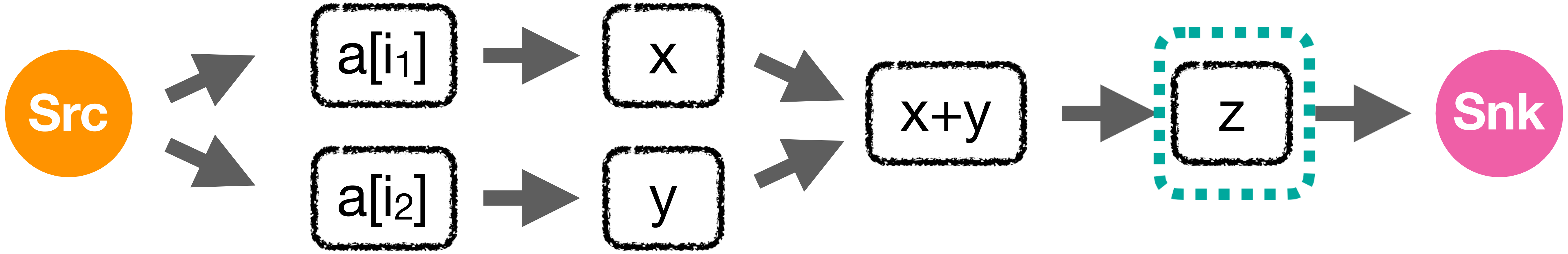
```
x = a[i1];
y = a[i2];
z = x + y;
_ = b[z];
```



... removing z requires only one **fence**

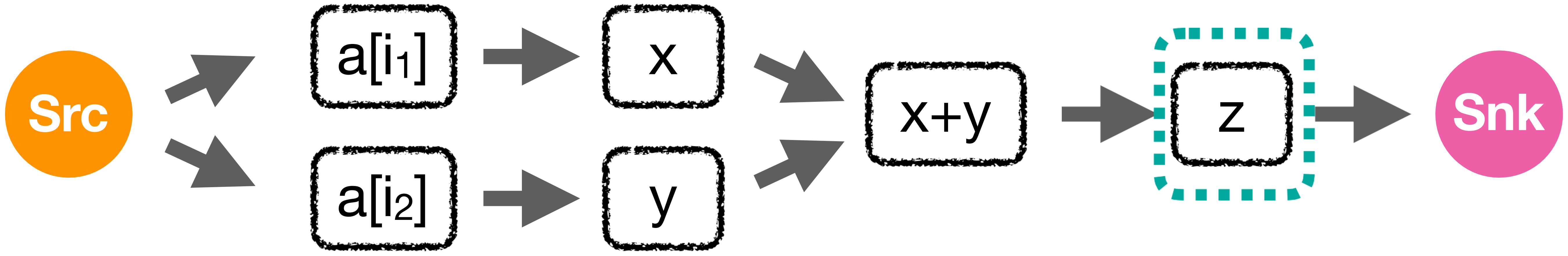
Idea: Cut data-flow between **Source** and **Sink**

```
x = a[i1];
y = a[i2];
z = fence( x + y);
_ = b[z];
```

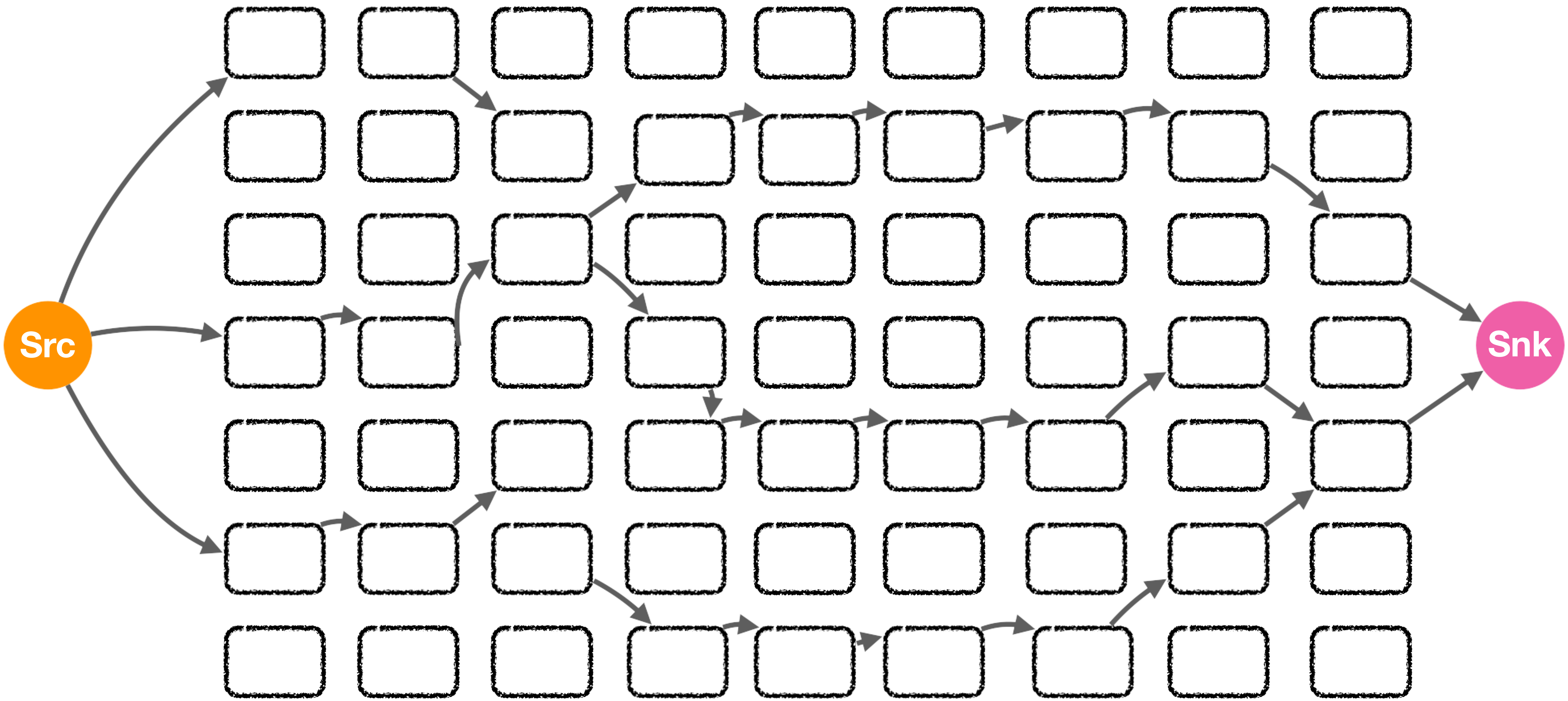


... removing z requires only one **fence**

Problem: Graph too big to do by hand!

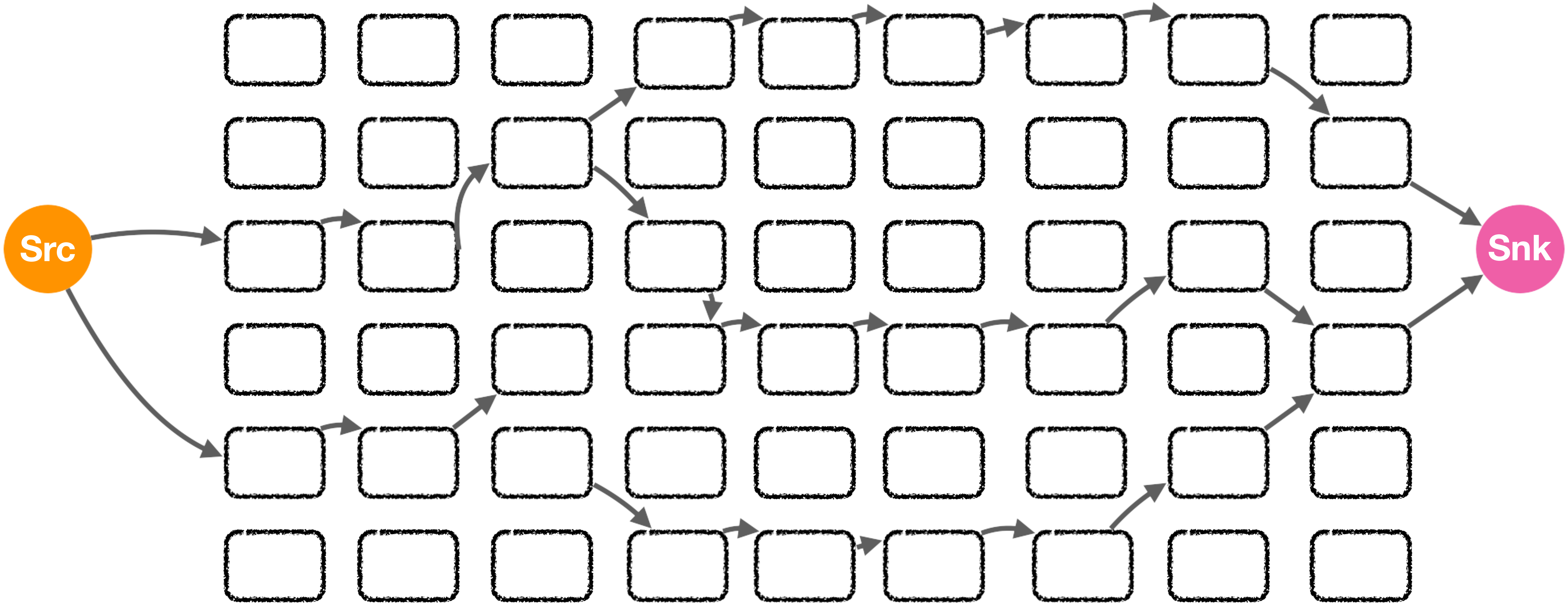


Problem: Graph too big to do by hand!



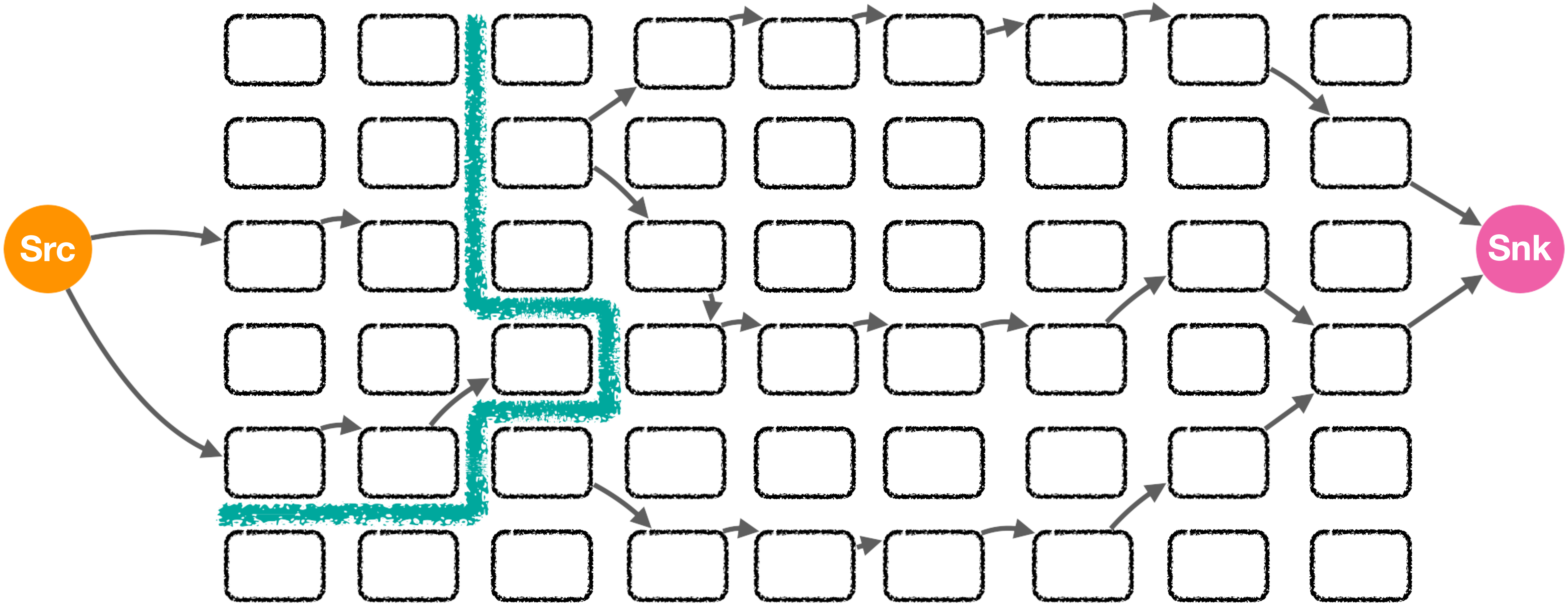
Problem: Graph too big to do by hand!

IDEA BLADE: Reduce to **Min Cut/Max Flow**!



Problem: Graph too big to do by hand!

BLADE: Reduce to Min Cut/Max Flow!



Reading

- Cache and timing attackers:

  https://www.usenix.org/system/files/conference/usenixsecurity16/sec16_paper_almeida.pd

- Speculative Execution Semantics: https://gleissen.github.io/papers/spectre-semantics.pdf

- Speculative Execution Type-System:
- https://blog.sigplan.org/2021/04/21/automatically-eliminating-speculative-leaks-from-cryptographic-code-with-blade/

  https://gleissen.github.io/papers/BLADE.pdf