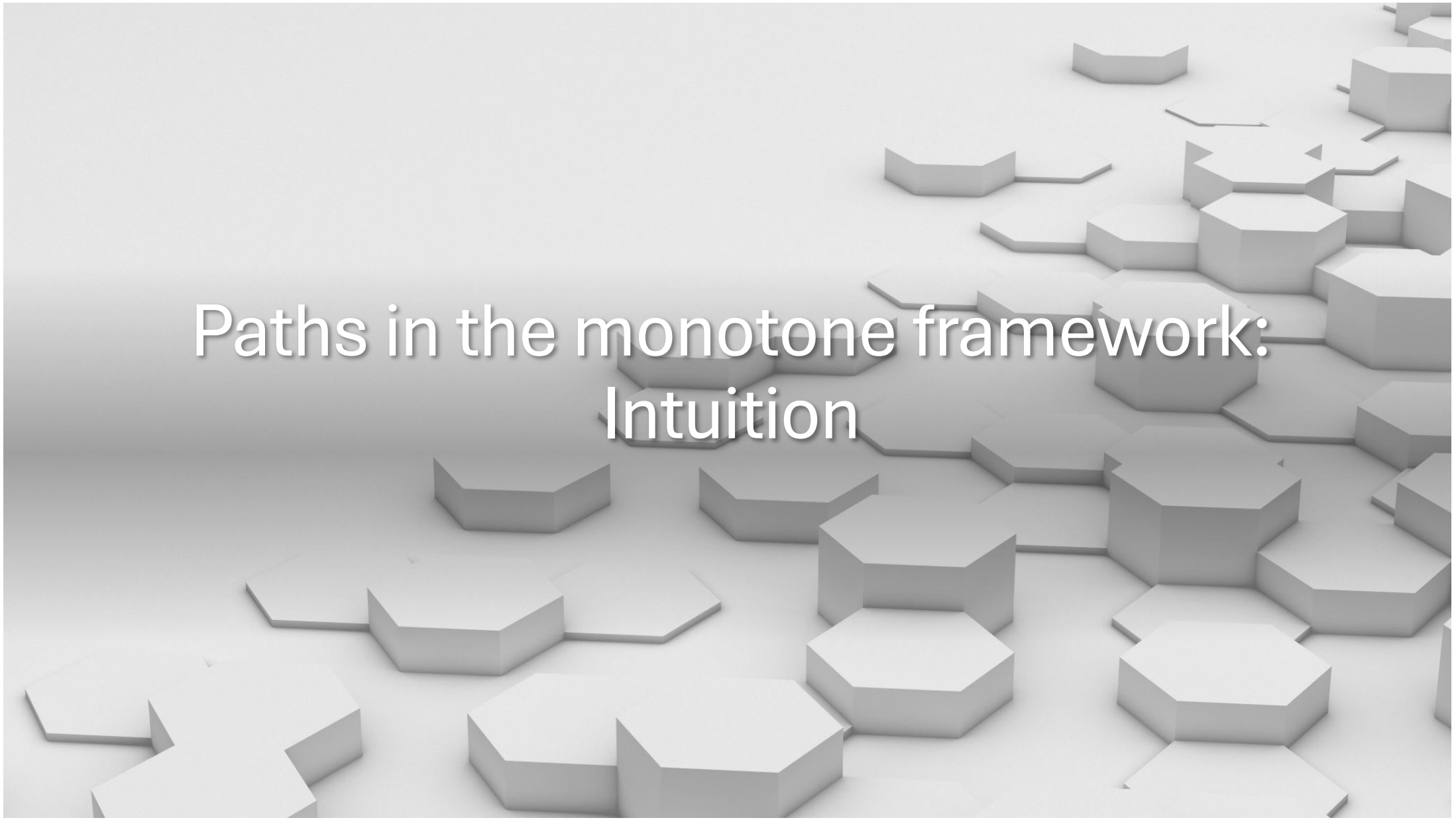

Paths in the monotone framework:
Intuition

The steps

We outline **how abstract states evolve** across **branches** in a **path-sensitive taint analysis**: **path predicates** and transfer functions are exploited to trace how taint information propagates and merges across control flow branches.

Three Steps:

1. **The Lattice Model Setup**: Formal definitions
2. **Branch Evolution**: State splits, transfer, and joins
3. **Worked Example**: Formal trace of a branching program

The Model

Variables: $PVar = \{x, y, z \ldots.\}$ is the set of program variables.

Abstract Taint Domain

A taint state $t = PVar \rightarrow \{untaint, taint\}$ ($untaint \leq taint$)

Path Condition Each abstract state is annotated with a **path predicate** φ (a Boolean formula) You can call them ABSTRACT PATH STATE

Path-sensitive state $s = (\varphi, t)$

The overall abstract state $S \in AS$ is a **set of such path states**:

$$S = \{s_i = (\varphi_1, t_i) for\ i \in I\}$$

ABSTRACT STATES

Map lattice of functions that go from path to pvar to taint states

$$AS = PATH \rightarrow PVar \rightarrow \{untaint \leq taint\}$$

?

Can be seen as a function taking a path, taking a variable and producing as a result a taint / untaint states

AS is a mapping from path to Prog.variable to taint statuses

TRANSFER FUNCTION

For each command (aka basic block) **cmd** we introduce the transfer function

$$f_{cmd} : AS \rightarrow AS$$

Take AS and provide AS

Since S is a set of path sensitive states

$$f_{cmd}(S) = \bigcup \{ f_{cmd}((\varphi, t) \mid (\varphi, t) \in S \}$$ ①

We need to define for each block a transfer func: how ab. state is modified by the command/block.

① How do we define it? We apply transf. fct. to each of the compoents of AS

AS is a set of path sensitive states. We apply fcmd to each path sensitive state.

Conditional

```
if (cond) {
    cmd1;
} else {
    cmd2;
}
```

$\forall(\varphi, t) \in S$ where S is the input abstract state

- For each pair (φ, t) apply transfer function: 1st actually, split paths because you have to apply branch.

Splitting paths → cond is true

Then Branch: $(\varphi \wedge cond, t)$

Else Branch: $(\varphi \wedge \neg\, cond, t)$

Conditional Merge Point

$S' = \{(\varphi \wedge cond, t_1), (\varphi \wedge \neg cmd, t_2)\}$ ①

↑ For example, if $S = \{(\varphi, t)\}$

Transfer Funtions

$f_{cmd1}(\varphi \wedge cond, t) = (\varphi \wedge cond, t_1)$

$f_{cmd2}(\varphi \wedge \neg cmd, t) = (\varphi \wedge \neg cmd, t_2)$

→ Then we apply the transfer functions for the then and else branch.

① To put together result of two paths is the union of the two different branches. We keep statuses different here because of path sensitivity.

Conditional

```
if (cond) {
    cmd1;
} else {
    cmd2;
}
```

$\forall(\varphi, t) \in S$ where S is the input abstract state

Splitting paths

Then Branch: $(\boldsymbol{\varphi \wedge cond, t})$

Else Branch: $(\varphi \wedge \neg\, cond, t)$

Conditional Merge Point

$\boldsymbol{S' = \{(\varphi \wedge cond, t_1), (\varphi \wedge \neg cmd, t_2)\}}$

Transfer Funtions

$\boldsymbol{f_{cmd1}(\varphi \wedge cond, t) = (\varphi \wedge cond, t_1)}$

$\boldsymbol{f_{cmd2}(\varphi \wedge \neg cmd, t) = (\varphi \wedge \neg cmd, t_2)}$

We do not join

$\boldsymbol{t_1, t_2}$

preserving path sensitivity

```
1: x = input();                  // Tainted source
2: if (x > 0) {
3:     y = sanitize(x);          // y = untaint
4: } else {
5:     y = x;                    // y = taint
6: }
7: print(y);                     // Sink
```

```
1: x = input();                 // Tainted source
2: if (x > 0) {
3:     y = sanitize(x);         // y = untaint
4: } else {
5:     y = x;                   // y = taint
6: }
7: print(y);                    // Sink
```

S_7 = {
  (x > 0, {x= taint, y = untaint}),
  (¬(x > 0), {x = taint, y = taint})
}

The analysis issues a **conditional warning** only on the path where **x ≤ 0.**

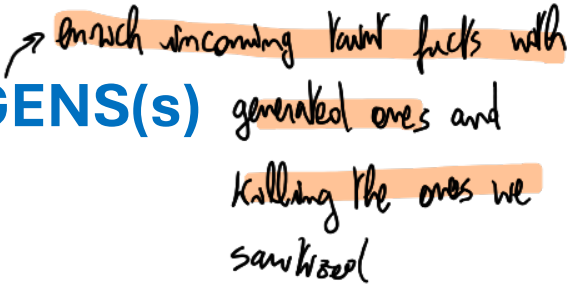
Line	Path Predicate	Taint Status
1	true	{x = taint}
2	x > 0	{x = taint}
3	x > 0	{ x = taint, y = untaint}
4	¬(x > 0)	{x = taint}
5	¬(x > 0)	{ x = taint, y = taint}
6	—	Path merge
7	Both predicates	{x = taint, y = untaint or taint}

What we have learned

Taint analysis tracks the flow of potentially unsafe (tainted) data through a program to identify integrity risks like injection attacks.

Key Concepts:

- **Taint sources:** e.g., user input, files, network
- **Taint sinks:** e.g., database queries, authenrtication routines
- **Sanitizers:** operations that remove taint
- **Dataflow Equations: OUT(s) = (IN(s) – KILL(s) U GENS(s)**
- **PATH CONDITIONS:** $(\varphi, TaintVar)$



Wht's next?

- **An analysis that models only a single function at a time is intra-procedural**
- **An analysis that takes multiple functions into account is inter-procedural**

Why interprocedural is important

```
fn read_input() -> String { get_user_input() }
fn process(x: String) { display(x); }

fn main() {
    let s = read_input();  // tainted source
    process(s);            // sink
}
```

The taint flows **across function calls**, so we need **interprocedural analysis** to detect it!

Interprocedural Analysis: Possible Steps

Build the CFG of the overall program

- Introducing function calls and return points

} We can easily build it

Summarize Functions

- Taint-in → Taint-out behavior
- Use **GEN/KILL sets**

} We summarize taint behavior for different functions individually

Propagate Taint

- Track flow from sources to sinks
- Across function boundaries

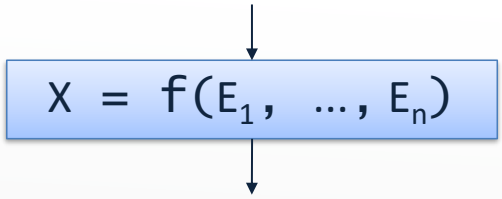
} And then we propagate facts across different calls

Report Violations

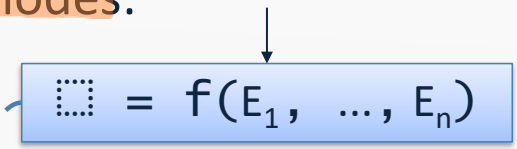
- If taint reaches sink then raise the alarm

The overall CFG

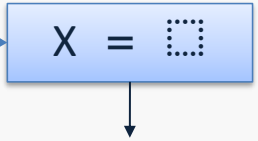
Split each original call node



into two nodes:



the "call node"

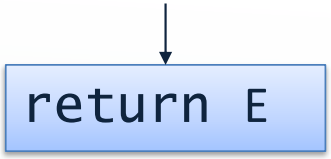


the "after-call node"

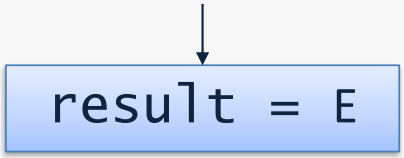
a special edge that connects the call node with its after-call node

The overall CFG

Change each return node



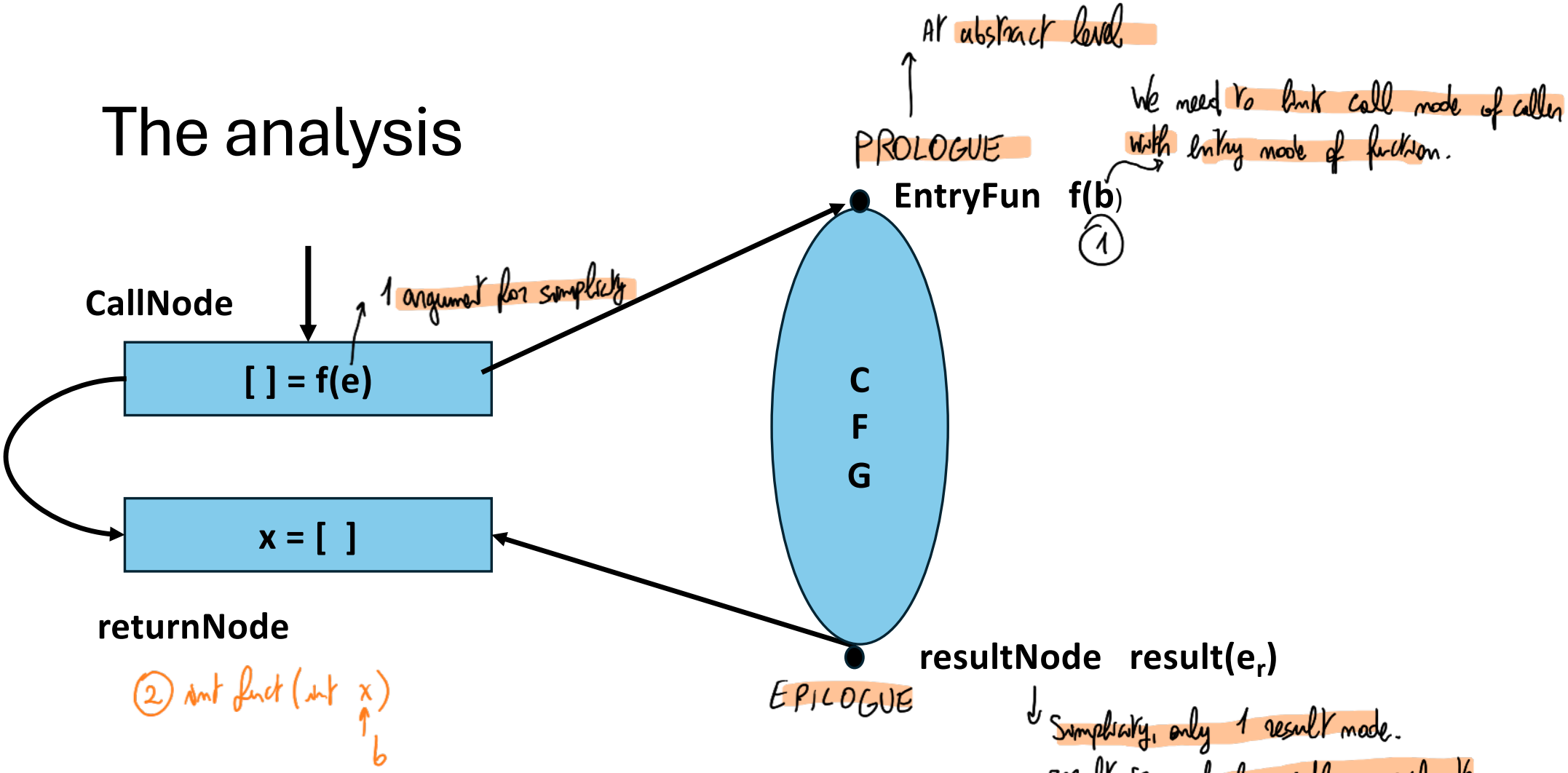
into an assignment:



(where `result` is a fresh variable)

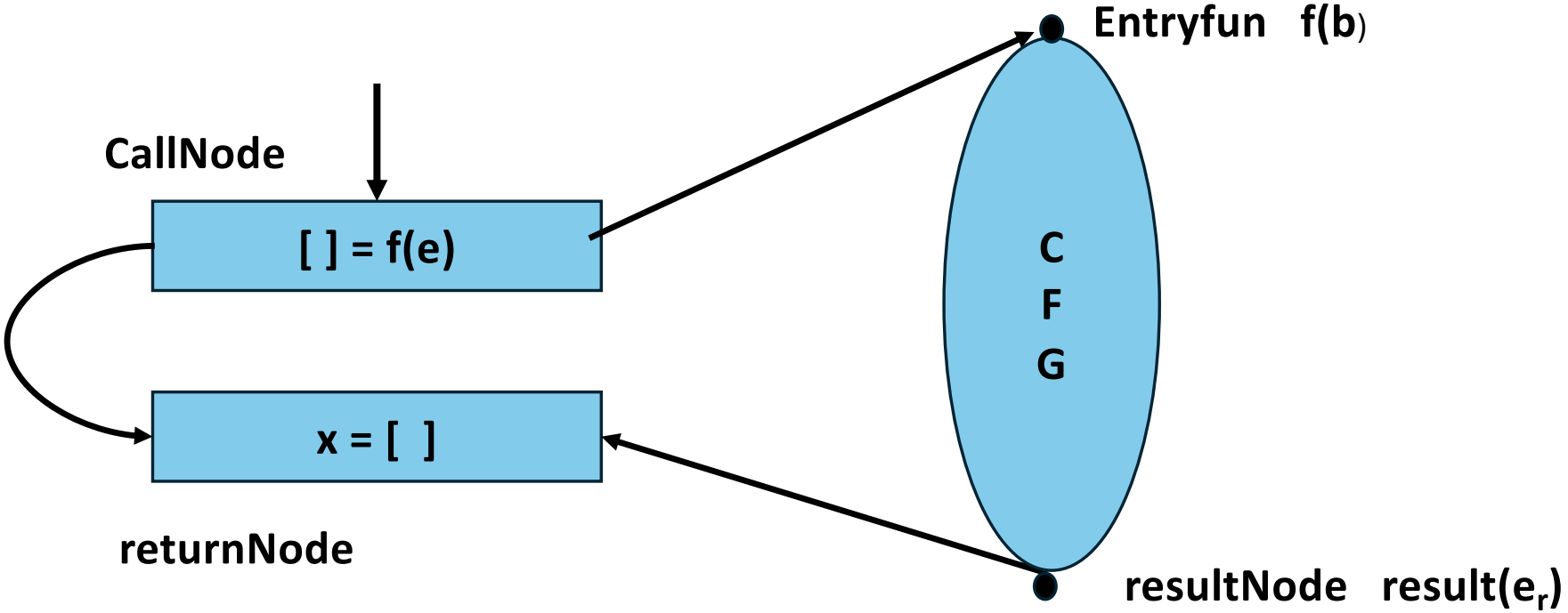
We modify function foo. We introduce a fresh variable result and return is an assignment to fresh variable result. Why fresh? Because we are under the assumption of SSA program. We don't break SSA property this way.

The analysis



1 f(e); call of a fun, and you have actual param.. In the function, b is the formal parameter you use in the function 2. Their correspondence is done when f. is called. PARAMETER PASSING BY VALUE

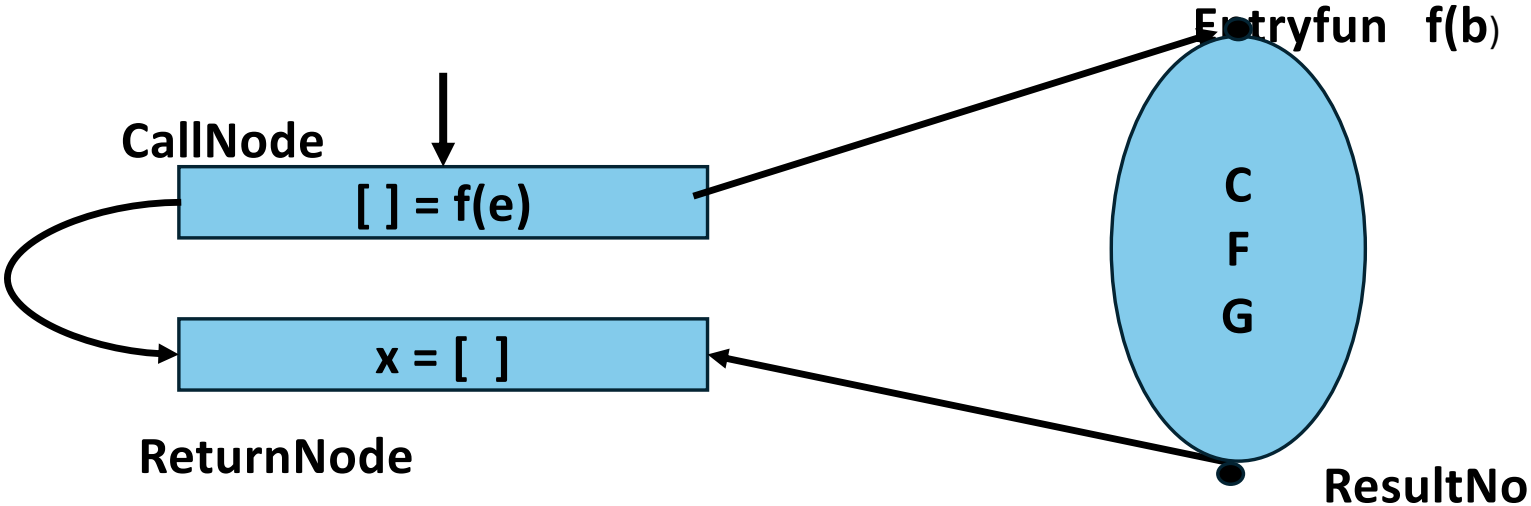
The analysis



Assumption: no gobal. Variables, no hep data, and no psths

↳ no heap ↳ no paths.

The analysis



Monotone FW

node of entry

$$[Entryfun] = \sqcup_{w} \perp [\, b_1(eval([w], e_1))], w \in pred(Entryfun)$$

HERE

compose info about local stacks w/ parameter passing. Static scope: no info we have at the beginning, because info we had is all of the caller. No globals etc.

① ② ③

$$[ReturnNode] = [CallNode]\,[x = eval([w], result)], w \in [pred(ReturnNode)$$

$$= [callNode][x = eval([ResultNode), e_r]$$

- Bottom: no info on all untainted. ① We have to evaluate input parameter e in taint status W which is predecessor in CFG of entry node (so the caller).

② We keep status of call node and you extend info w/ mapping of x (in which we put the result). Thus for all the predecessors of return node (which is result node!)

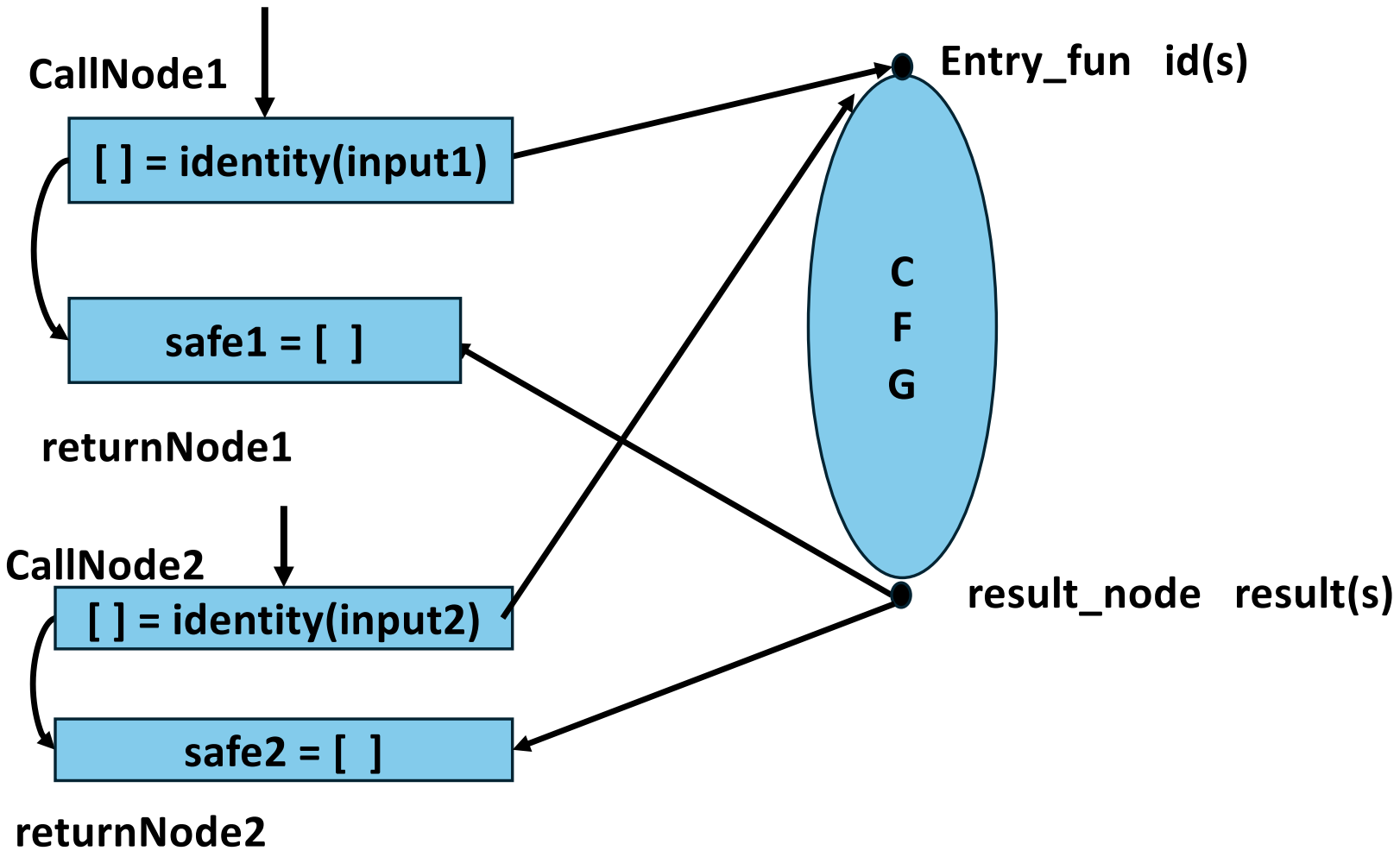
③ Return node uses right associated call node because that return node is indeed associated to that call node.

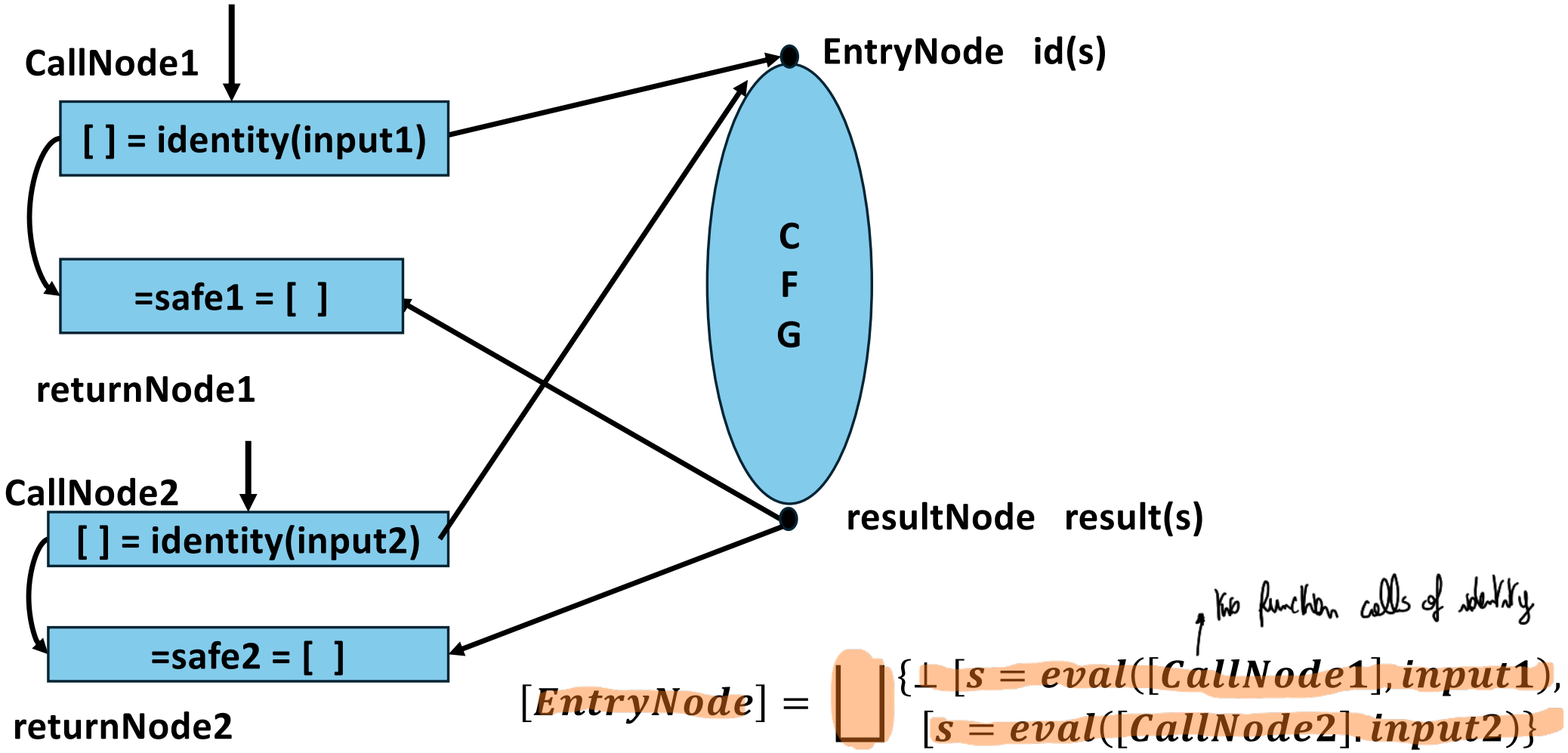
```
fn identity(s: String) -> String {
    return s;
}

fn process() {
    let input1 = source();                  // tainted
    let safe1 = identity(input1);           // tainted (taint propagates)

    let input2 = "static text".to_string(); // untainted
    let safe2 = identity(input2);           // untainted

    sink(safe1);
    sink(safe2);
}
```

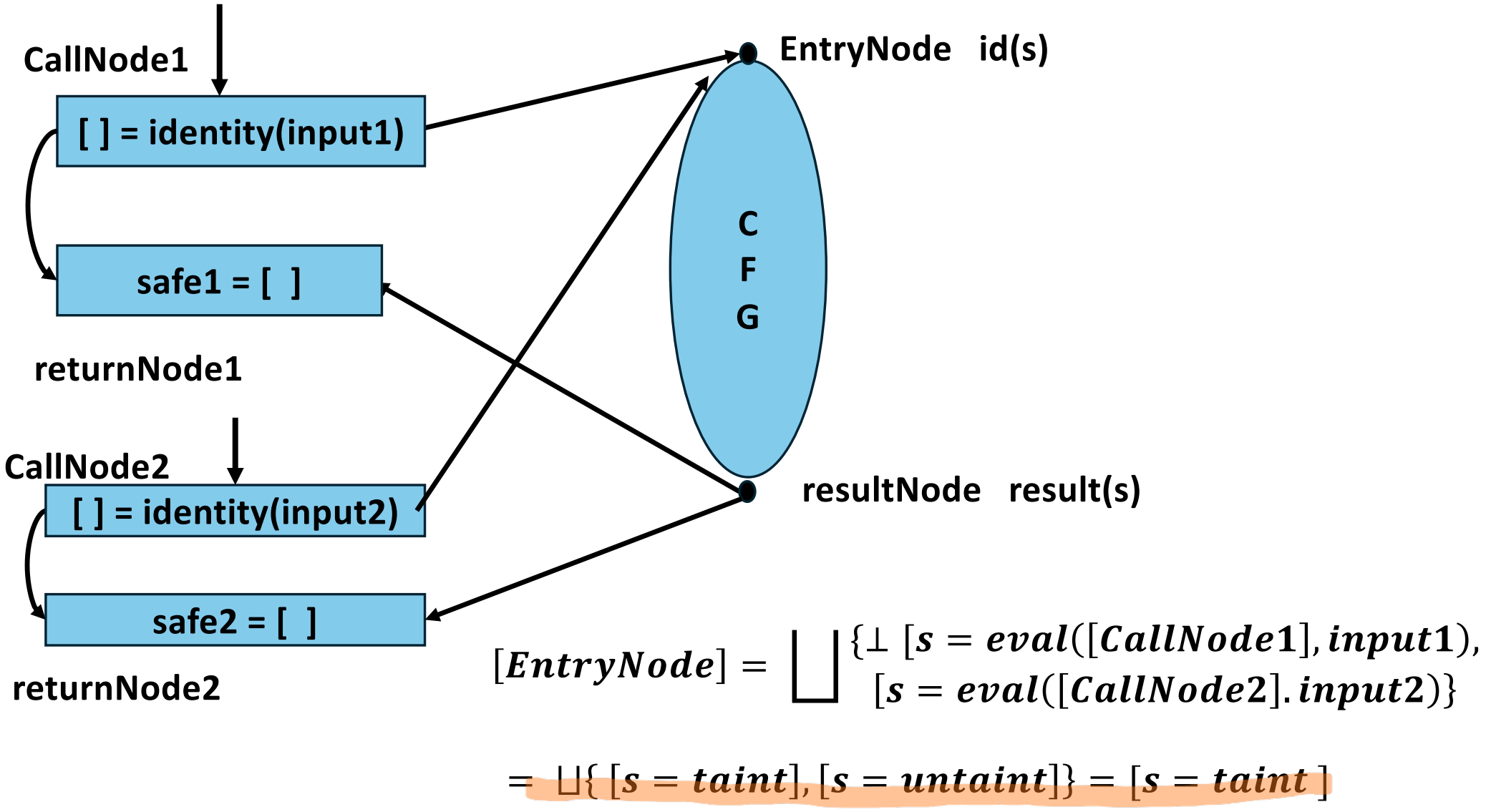
We want to analyze:

Whether sink(safe2) is correctly **not flagged**.

Whether identity()'s behavior is **correctly tracked per input**.


CallNode1
[ ] = identity(input1)
safe1 = [ ]
returnNode1
CallNode2
[ ] = identity(input2)
safe2 = [ ]
returnNode2
Entry_fun id(s)
C
F
G
result_node result(s)


CallNode1
[ ] = identity(input1)
=safe1 = [ ]
returnNode1
CallNode2
[ ] = identity(input2)
=safe2 = [ ]
returnNode2
EntryNode id(s)
C
F
G
resultNode result(s)
two function calls of identity
$[EntryNode] = \sqcup \{\bot [s = eval([CallNode1], input1)], [s = eval([CallNode2], input2)]\}$



$$[EntryNode] = \bigsqcup \{\bot\ [s = eval([CallNode1], input1),\ [s = eval([CallNode2].input2)\}$$

$$= \sqcup\{\ [s = taint], [s = untaint]\} = [s = taint\ ]$$

Analysis Result

This analysis generates **one global summary** for the identity() function: “identity() may return tainted data.”

The analysis overapproximates the behaviour of function:

- Any call to identity() might return taint.
- Therefore, both safe1 and safe2 are **tainted**.

MAY ANALYSIS: NOT SURPRISING: We have two paths in which one is tainted!

False Positive

sink(safe2); // **incorrectly flagged tainted**

Global Summary

- The **function summary** is an abstract description of how a function **affects taint flow**:
  - It captures how taint in the **inputs** affects the **outputs**, **globals**, or **sinks**.
  - It is designed to be **reusable across all call sites**, so that the function body doesn't need to be re-analyzed every time.
- Overall: It combines **all observed or possible behaviors** into **one single summary**, which is **safe** (no missed vulnerabilities), but **imprecise**:
  - It’s sound but not precise.
  - Even calls with **untainted input** now **return possibly tainted** results.
  - This leads to **false positives**, like sink(safe2) in our example.

How do we manage? Global summary pulling together all possible calls, so we are keeping an over approx of all possible calls and if there's one kind evaluates, then ok, kept.

AVOID
FALSE POSITIVE

Function Cloning : We put together function for different calls

Summary for each function call.

- Clone functions such that each function has only one callee
  - Can avoid interprocedurally invalid paths ☺
  - For high nesting depths, gives exponential blow-up ☹
  - Doesn't work on (mutually) recursive functions ☹

a calls b,
b calls c,
c calls a.

Depth can be a problem

You don't know statically how many recursive calls

Context Sensitive Analysis

↑ We move towards a CSA.

- Function cloning provides a kind of context sensitivity (also called polyvariant analysis)
- Instead of physically copying the function CFGs, do it logically
- Extend the lattice of abstract states by including the Context. Of course you need to extend lattice in a good way
  - The context is the abstractions of the state at function entry ①
  - Contexts must be finite to ensure finite height of the analysis lattice
- Different strategies for choosing the set Contexts…

① Abstraction of the possible calls for each of the functions.
You kind of clone the calls: for each call, you store context in terms of values. It's like not taking least upper bound to do global analysis, but consider each call to do a local analysis (JMP 19 (HERE))

CONTEXT: collection of past info related to a specific call.

Our Running Example: Context Sensitive Analysis

Idea: distinguish the two calls based on input:

Call 1: identity(input1) Call 1 context: we called identity with input 1.

- input1 is tainted
  - **GEN = {safe1}**
  - **KILL = Ø**
- Output: safe1 is tainted, Hence **sink(safe1) = violation**

Call 2: identity(input2)

- input2 is untainted
  - **GEN = Ø**
  - **KILL = {safe2}** safe2 is not tainted
- Output: safe2 is untainted. **Hence sink(safe2) = safe**

No warning is raised on safe2 (kill here to say that safe2 is not a tainted value)

We switched to gen/kill set

Analysis Summary

Line	GEN	KILL	Comments
input1 = source()	{input1}	∅	Introduces taint
safe1 = identity(input1)	{safe1}	∅	Propagates taint
input2 = "static text"	∅	∅	Constant, untainted
safe2 = identity(input2)	∅	{safe2}	Keeps it clean
sink(safe1)	–	–	Violation — taint reaches sink
sink(safe2)	–	–	Safe — no taint reaches sink

Trade off

Approach	Precision	Performance
Context-insensitive	Lower	Faster
Context-sensitive	Higher	Slower

Context Sensitivity: Call Graphs

To do this, static analyzers introduce a call graph.

- A **call graph** is a directed graph that represents the **calling relationships between functions** in a program.
- **Structure:**
  - **Nodes** = functions or methods
  - **Edges** = function calls (caller to callee)

↳ contain info of calling patterns in a program

Example

```
fn main() {
    let input = get_user_input();
    process(input);
}

fn read_input() -> String { get_user_input() }
fn process(data: String) { display(data); }
```

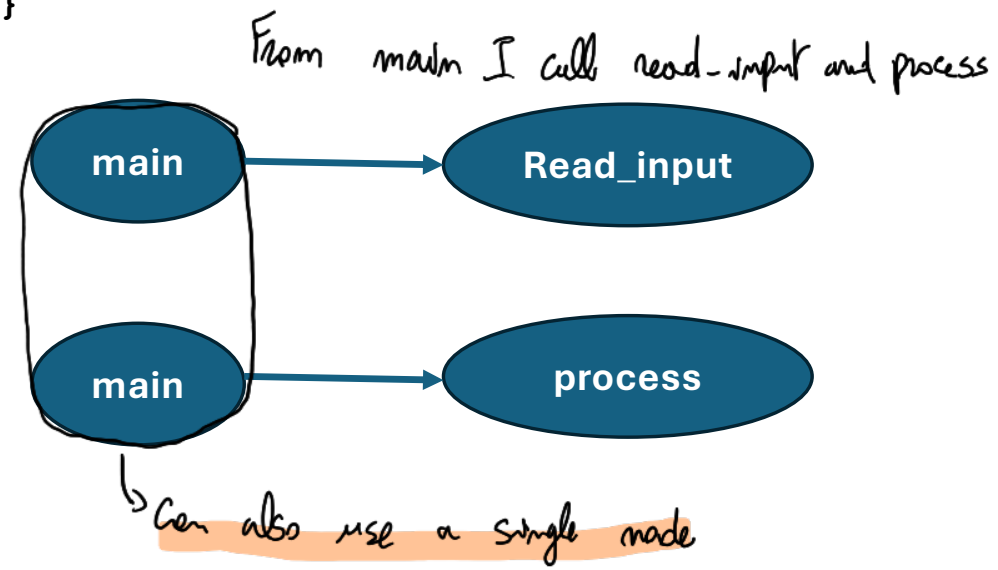
Example

```
fn main() {
    let input = get_user_input();
    process(input);
}

fn read_input() -> String { get_user_input() }
fn process(data: String) { display(data); }
```

The Call Graph

main - - > read_input

main - - >process



```
fn id(x: String) -> String {
    return x;
}

fn foo() {
    let a = source();       // tainted
    let b = id(a);          // tainted
    sink(b);                // violation
}

fn bar() {
    let c = "hi".to_string(); // untainted
    let d = id(c);            // should be untainted
    sink(d);                  // false positive if
context-insensitive
}
```

bar satisfies, foo does not.
foo should give a warning

```
fn id(x: String) -> String {
    return x;
}

fn foo() {
    let a = source();       // tainted
    let b = id(a);          // tainted
    sink(b);                // violation
}

fn bar() {
    let c = "hi".to_string(); // untainted
    let d = id(c);            // should be untainted
    sink(d);                  // false positive if
context-insensitive
}
```

Context-Sensitive Use of Call Graph

Use the call graph to **trace individual call sites**.

Analyze id() **per calling context** (i.e., per edge in the graph).

At foo: id: tainted input: we track the taint propagation of the input

At bar: id: untainted input: no taint propagated

The analysis applies **precise summaries per call site**, avoiding **false positives**.

```
fn id(x: String) -> String {
    return x;
}

fn foo() {
    let a = source();       // tainted
    let b = id(a);          // tainted
    sink(b);                // violation
}

fn bar() {
    let c = "hi".to_string(); // untainted
    let d = id(c);            // should be untainted
    sink(d);                  // false positive if
context-insensitive
}
```

Site	Caller	Input Tainted?	Output Tainted?
Edge: foo → id	foo	Yes	Yes
Edge: bar → id	bar	No	No

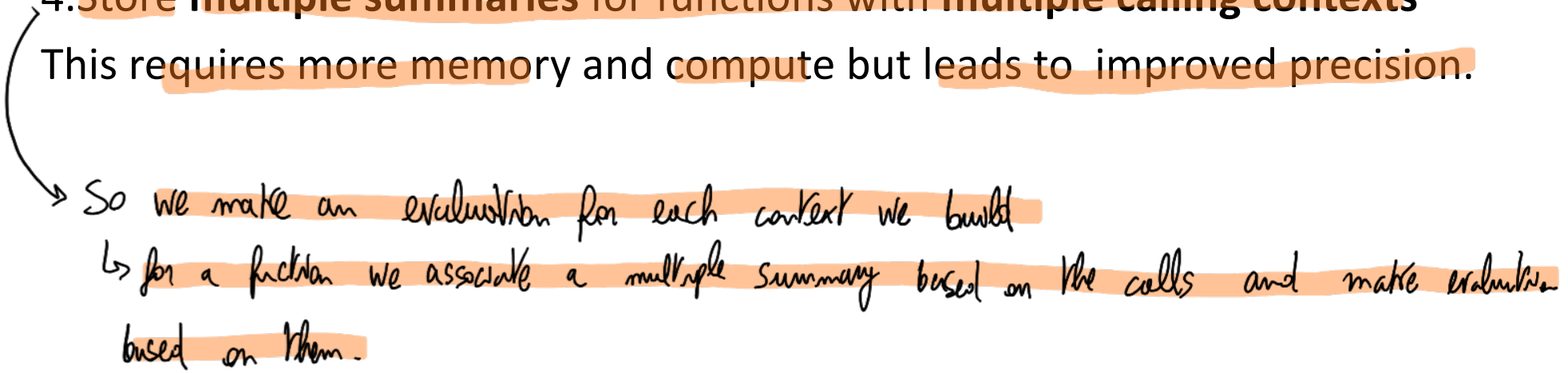
id won't always give tainted sink

The steps

Context-sensitive taint analyzer:

1. Build the **call graph statically**
2. Traverse it
3. For each function, generate a **summary per call context**
    1. Argument values/types
    2. Call-site location
    3. Possibly caller name
4. Store **multiple summaries** for functions with **multiple calling contexts**

This requires more memory and compute but leads to improved precision.



Intresting questions

Challenges and solutions related to call graphs in **static interprocedural taint analysis**:

1. **How many calls to consider**
2. **Handling recursive calls**
3. **Solutions and strategies**

How Many Calls

Direct calls:

- Straightforward: Function A calls B . The edge a to B is added to the call graph.

Indirect calls:

- Function pointers, virtual methods, trait objects, closures:
  - Static resolution may not be possible.
  - **points-to analysis** or **type inference** to resolve targets. ① ③

In C you can call a function through a pointer.

Polymorphic/Generic calls: ②

- Functions may be instantiated differently across types.

→ The code that gets run might depend on how func is used

Multiple call sites:

- A function may be called from many locations with different taint contexts (e.g., tainted vs. untainted inputs).

↳ additional source of complexity.

① Static resolution of indirect call not necessarily possible.

② Imagine sort function, that takes a generic types.

An **indirect call** is when the code calls a function, but **doesn't say directly which one**—instead, it uses some kind of variable to decide at runtime.

Let's look at a simple example in C:

C Copy code

```c
void say_hello() {
    printf("Hello!\n");
}

void say_bye() {
    printf("Goodbye!\n");
}

void execute(void (*func)()) {
    func(); // indirect call
}
```

Here, `func` is a **function pointer**. You can pass either `say_hello` or `say_bye` into `execute`, and it'll call whichever one you gave it. But **statically**, just reading the code, it's **not immediately clear** which function will run when you see `func()`.

Where do indirect calls show up besides function pointers?

1. Virtual Methods (Object-Oriented languages)

In something like Java or C++, if you do:

Cpp Copy code

```cpp
class Animal {
    virtual void speak();
};

Animal* pet = new Dog();
pet->speak(); // Which 'speak'? Depends on the object type
```

Here `pet->speak()` is an indirect call. The actual method depends on whether `pet` is a `Dog`, `Cat`, etc. Static analysis has to do **type analysis** to figure out what types `pet` could be.

How do analyzers deal with this?

They can use:

- **Points-to analysis**: What functions could a pointer or variable point to?
- **Type inference**: What possible types does an object have at a certain point?
- **Call site cloning** (in context-sensitive analysis): Duplicate the function analysis per different calling context so you can keep track of taint properly.

But all this adds **complexity**—it's much harder than just following direct calls.

Polymorphic/Generic Calls

- This happens when you write a function that can work with **different types**, like `fn do_something<T>(input: T)`.
- It might be used in different places, with different types.
- That means the actual code that gets run could vary depending on how it's used, and the taint context could change too.

How Many Calls

- In a **context-sensitive analysis** , the number of **contexts explodes**:
  - For N functions, there may be **$O(N^2)$** or more call edges to consider.
  - This leads to scalability issues

You can say, I only consider a fixed amount of codes.

Recursive Calls

- The call graph contains **cycles**. (indirect calls or recursive calls)
- There may be **no fixed number of call steps**.
- The taint state must **stabilize** across recursive calls.

Complicated

STRATEGIES

Bounded Analysis

Context-Bounded Analysis

- Limit the **depth of the call chain** per function (e.g. k-limiting).
  - Example: k = 1, only direct call contexts tracked.
- Pro: **Controls explosion**
- Cons: **May reduce precision**

Widening

Idea: we only see what happens after x iterations and we stop

For recursive calls, apply **widening** to force convergence after a few steps.

- After 3 recursive calls, assume "worst case" (tainted).

Pro: **Guarantees termination**

Cons: **May introduce imprecision**

↓ If it does not occur after 3 calls, we don't report it.

↳ We might have false negatives! We don't know how many calls are needed to stabilize. UNDECIDABLE

Key insights

The **call graph** is the backbone of interprocedural taint analysis and it may become large, dynamic, or cyclic:

We must use **approximation**, **abstraction**, or **iteration** to balance precision and feasibility.

Recursive functions and libraries (like utility functions) are the most complex to model precisely.

We have to perform over approximation to achieve feasibility.

One solution is IFDS: algorithm developed for context dependent forward may analysis (so static analysis works)

IFDS (Interprocedural Finite Distributive Subset problems) [IFDS Algorithm]

- Precise Interprocedural Dataflow Analysis via Graph Reachability, Reps, Horwitz, Sagiv, POPL 1995
- **Great idea #1:** ①
  - Transfer functions are distributive and constraints can be represented compactly!
  - Distributivity closed under composition and least upper bound,

For our may analysis great!

- Function summaries can be represented compactly and without loss of precision!
- **Great idea #2:**
  - tabulation solver (smart algorithmic solution)

↳ better than worklist, extension that works effectively w/ context and transfer functions.

SAY THIS: ①
We exploit distributive property of transfer functions when it comes to least upper bound to simplify computations

DISTRIBUTIVE: $f(\bigcup_{n} \text{summaries}_n) = \bigcup(f_{s_0}, \ldots f_{s_m})$

IFDS (and IDE) at wortk

- Soot: https://github.com/Sable/heros
- WALA: https://github.com/amaurremi/IDE

- From a call graph you can assemble a way to each call and you save that way so if you see a mismatch you can notice problems. CONTROL FLOW INTEGRITY

Some References

- Nomair A. Naeem, Ondrej Lhoták, Jonathan Rodriguez: Practical Extensions to the IFDS Algorithm. CC 2010
- Eric Bodden: Inter-procedural Data-flow Analysis with IFDS/IDE and Soot. SOAP@PLDI 2012
- Jonathan Rodriguez, Ondrej Lhoták: Actor-Based Parallel Dataflow Analysis. CC 2011
- Steven Arzt, Eric Bodden: Reviser: Efficiently Updating IDE-/IFDS-based Data-Flow Analyses inResponse to Incremental Program Changes. ICSE 2014
- Magnus Madsen, Ming-Ho Yee, Ondrej Lhoták: From Datalog to Flix: A Declarative Language for Fixed Points on Lattices. PLDI 2016
- Johannes Späth, Karim Ali, Eric Bodden: IDEal: Efficient and Precise Alias-Aware Dataflow Analysis. Proc. ACM Program. Lang. 1(OOPSLA): 99:1-99:27 (2017)