A posteriori taint-tracking for demonstrating non-interference in expressive low-level languages

Peter Aldous and Matthew Might
University of Utah

This title is a little much, so …

Faster automated proofs that information doesn't leak in Android apps

Peter Aldous and Matthew Might
University of Utah

here's a version that's a little less dense.

A posteriori taint-tracking for demonstrating non-interference in expressive low-level languages

Peter Aldous and Matthew Might
University of Utah

Where does my password go?

Is my password being sent to someone?
Will it send tweets on my behalf?
Who can find out where I am?

Denning & Denning

Back in the 1970s, Denning and Denning created a dynamic analysis called taint tracking. Essentially, we put dye in pipes and see where it goes. But what we need has to be different from this formulation of D&D in a few ways:

- Fully static
- Provably correct - that is, that it catches _all_ information leaks
- Effective even with rich languages - conditional gotos, functions, exceptions
    - D&D mention conditional gotos but omit functions and exceptional flow

To achieve these goals, we perform a Denning-and-Denning-style analysis …

Denning & Denning + Cousot & Cousot

… inside a small-step abstract interpreter.

D&D + C&C

… inside a small-step abstract interpreter.

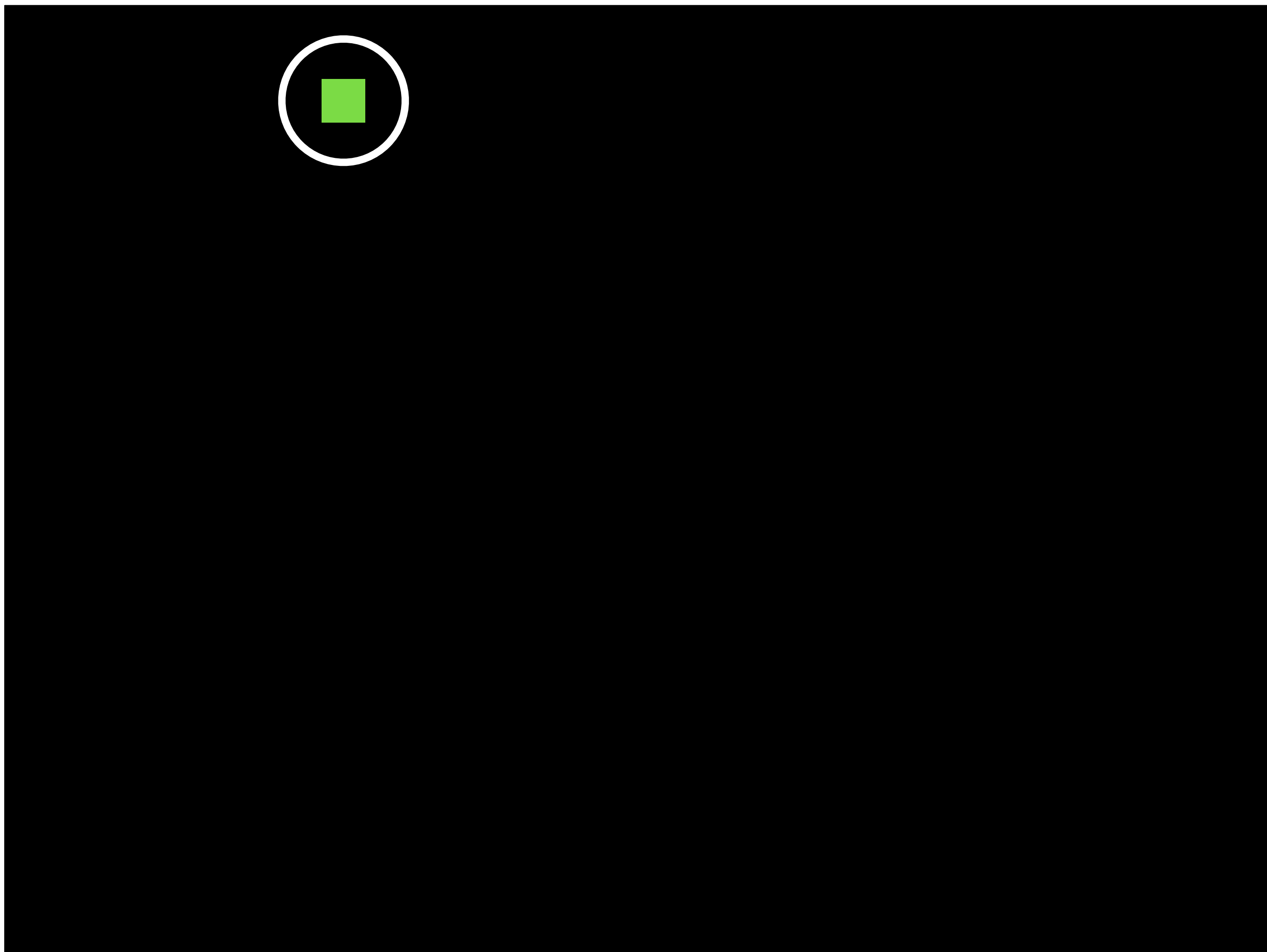
Denning and Denning did not, however, prove that this system catches all possible information leaks.

In order to prove a notion like this, we need a suitable formalism for leak identification. This formalism, which is well known, is called non-interference.

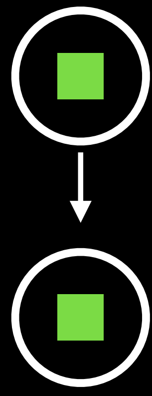
Non-interference

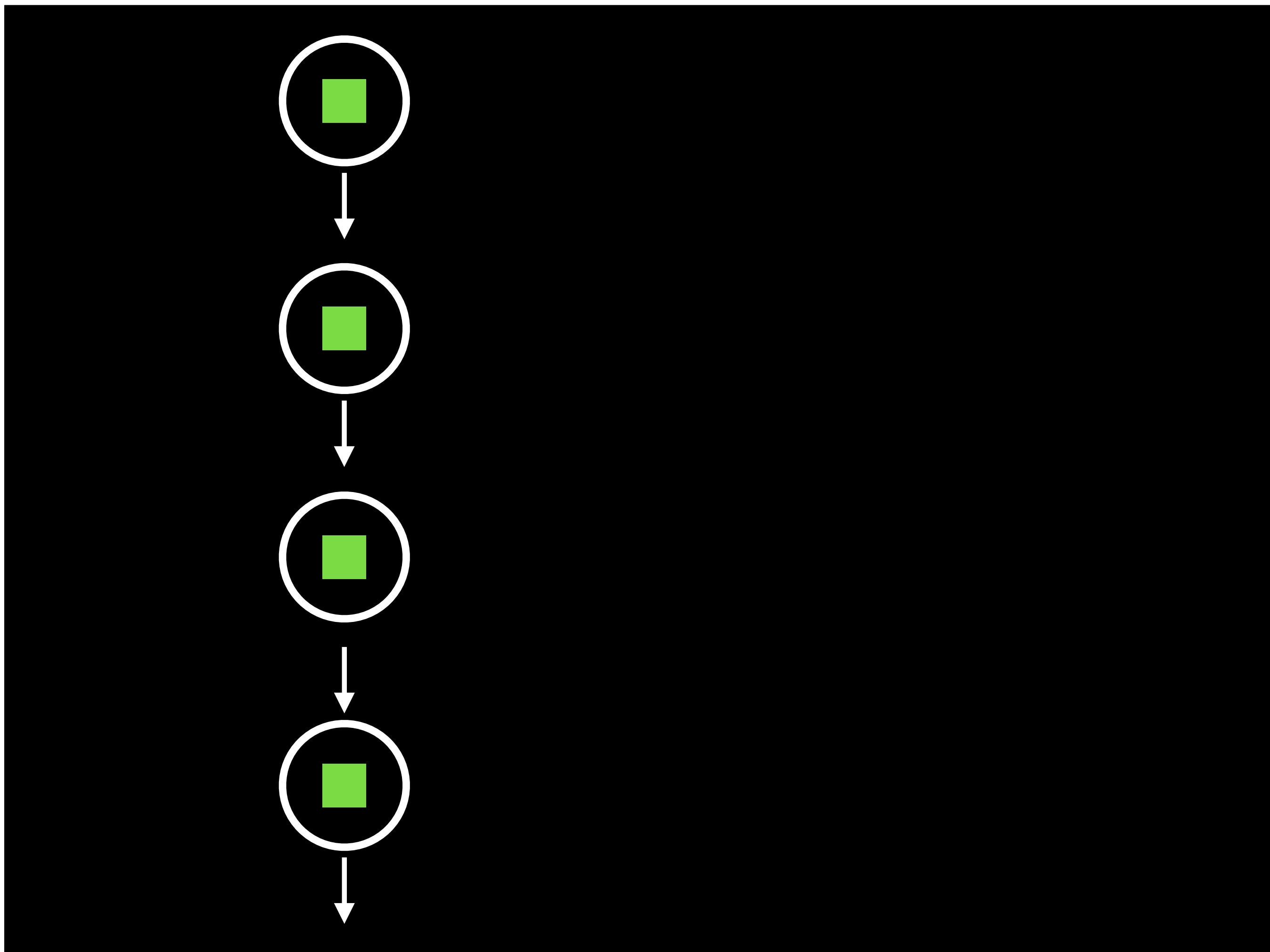
No sensitive information may affect the observable behavior of a program

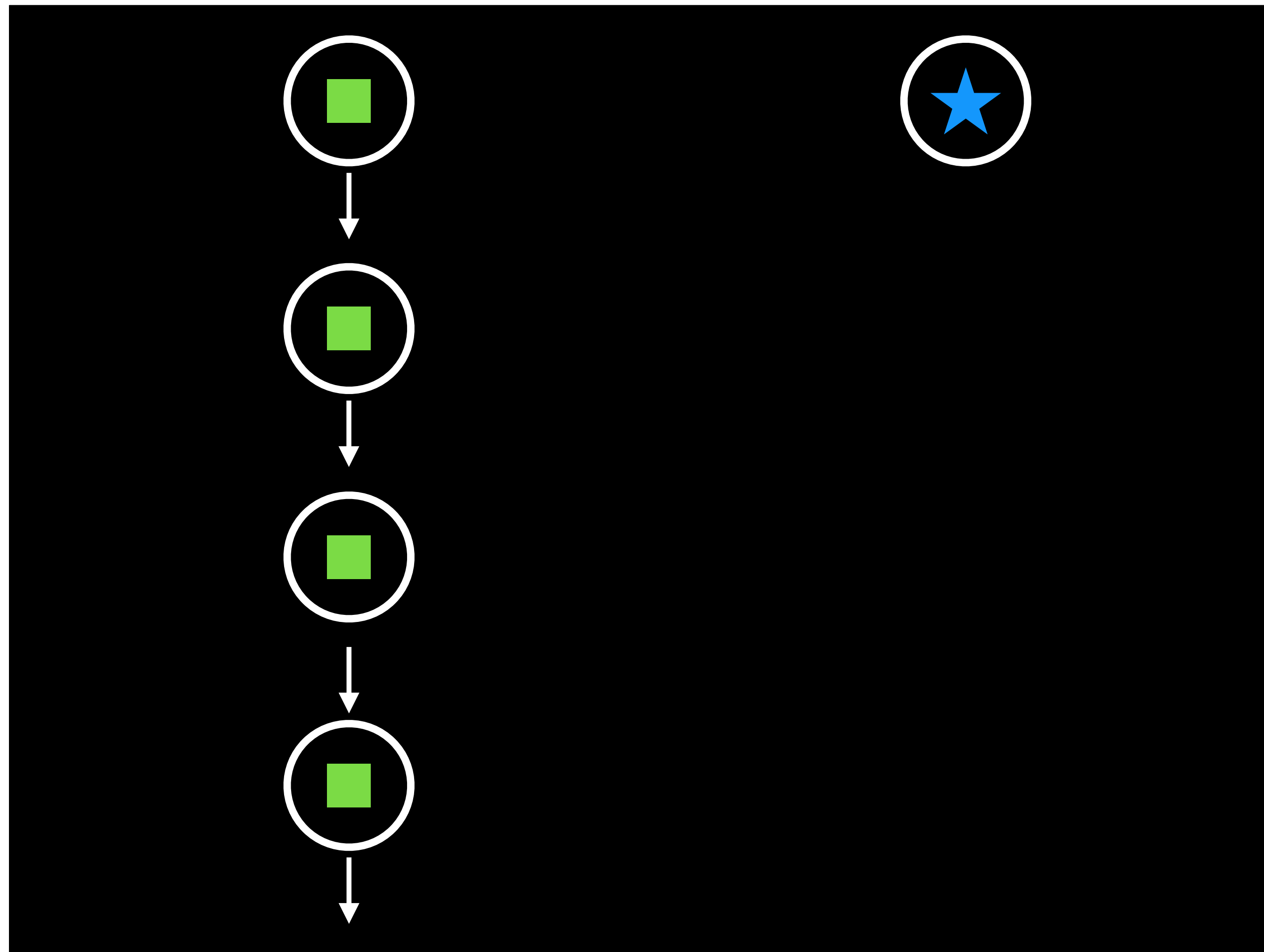
Formally, any two program traces that differ only in their sensitive values must exhibit the same observable behaviors.



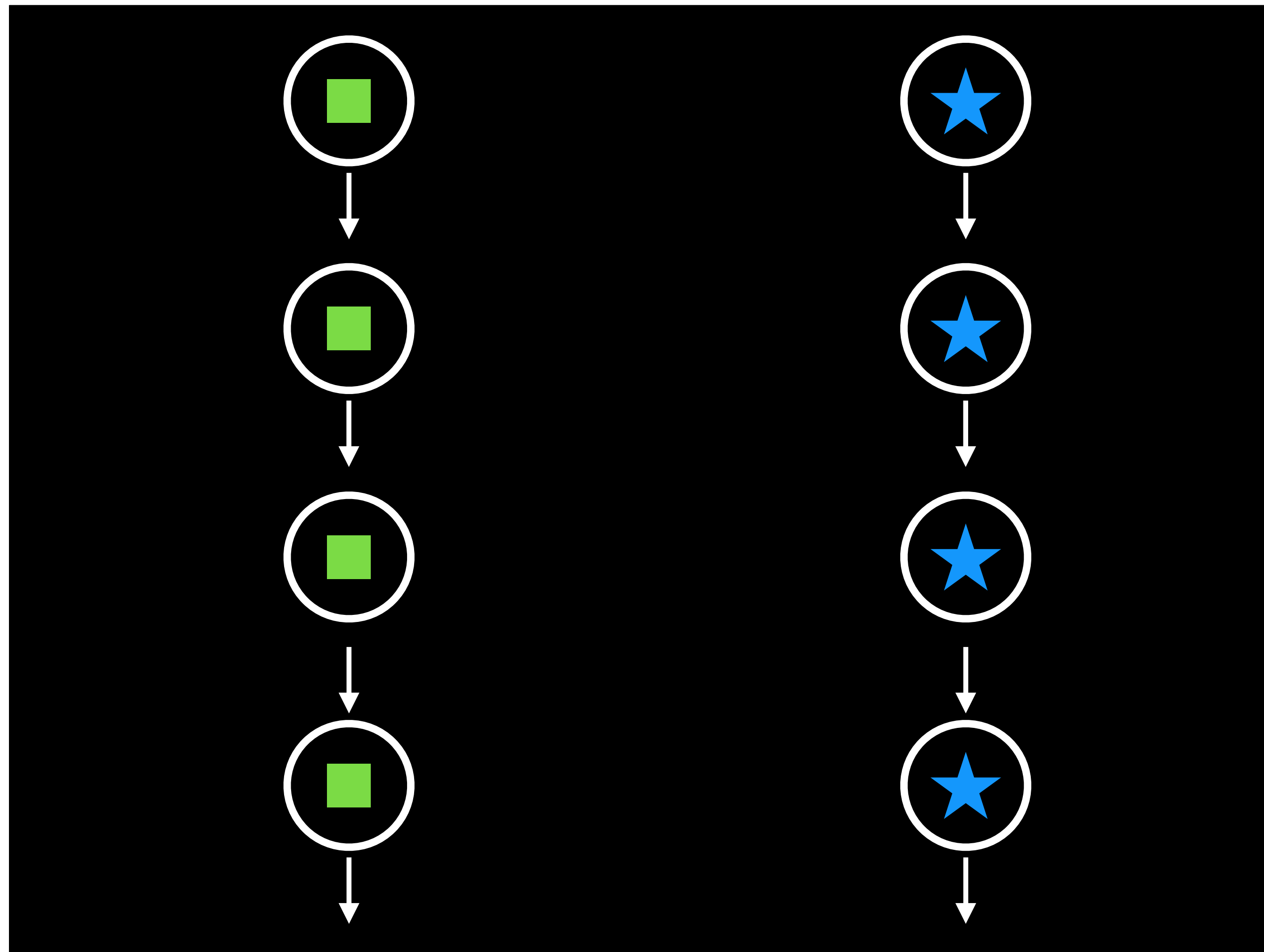
So if we have one program trace with a sensitive value whose value is a green square, …







… a program trace that starts identically (except with a different sensitive value) …



… must behave the same (for some definition of observable behavior).

CESK

We start with a CESK machine with small-step semantics.

Control
Environment
Store
Kontinuation

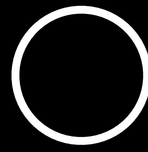
(Felleisen and Friedman, 1987)

C is a code point or control state that indicates where we are in the program. E is an environment that maps variables to addresses. S is a store that maps addresses to values. And K is a continuation or the program stack.

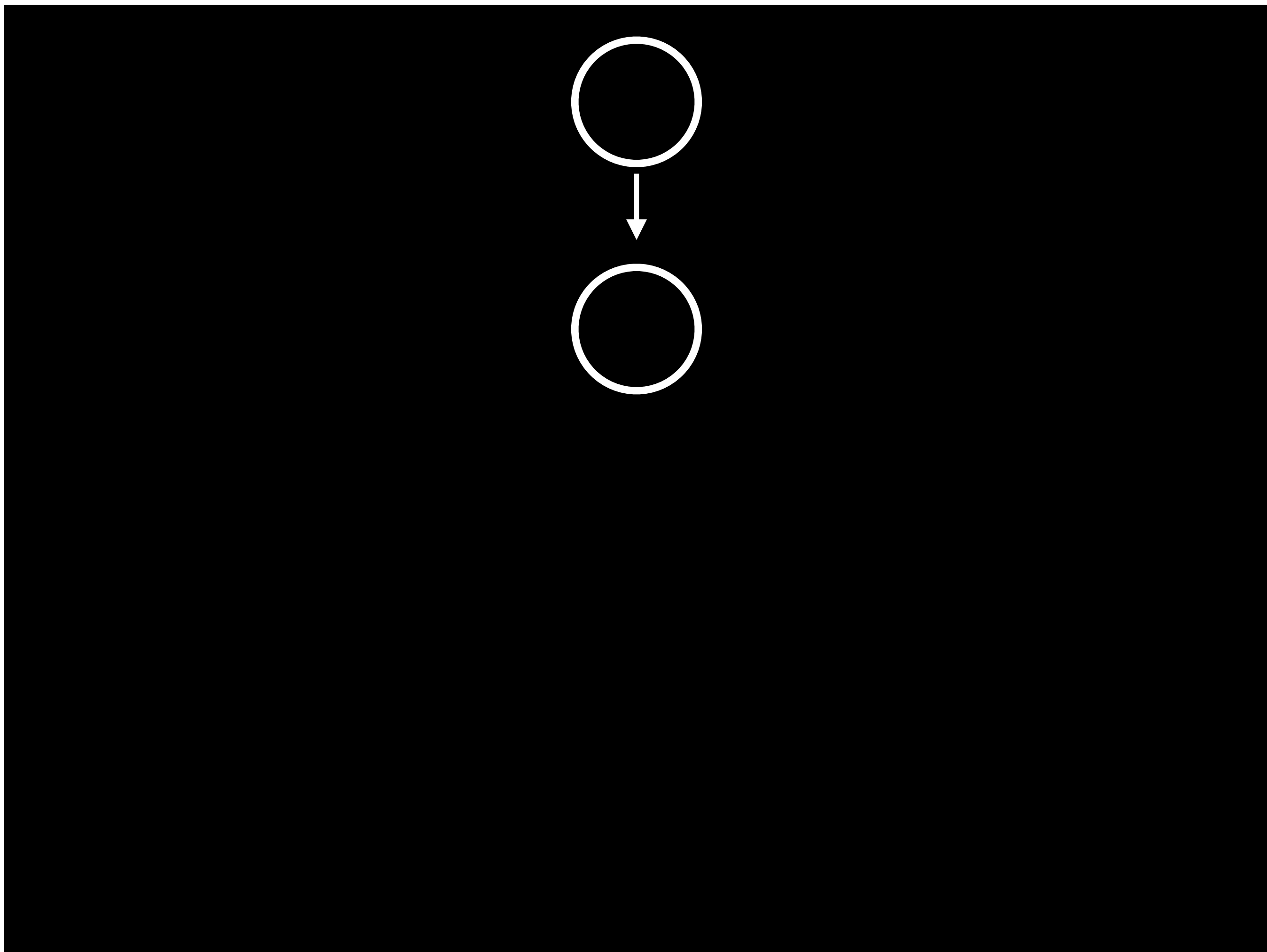
Instruction
Registers
Heap
Stack

For those of us with a more imperative background, these terms may be more comfortable.

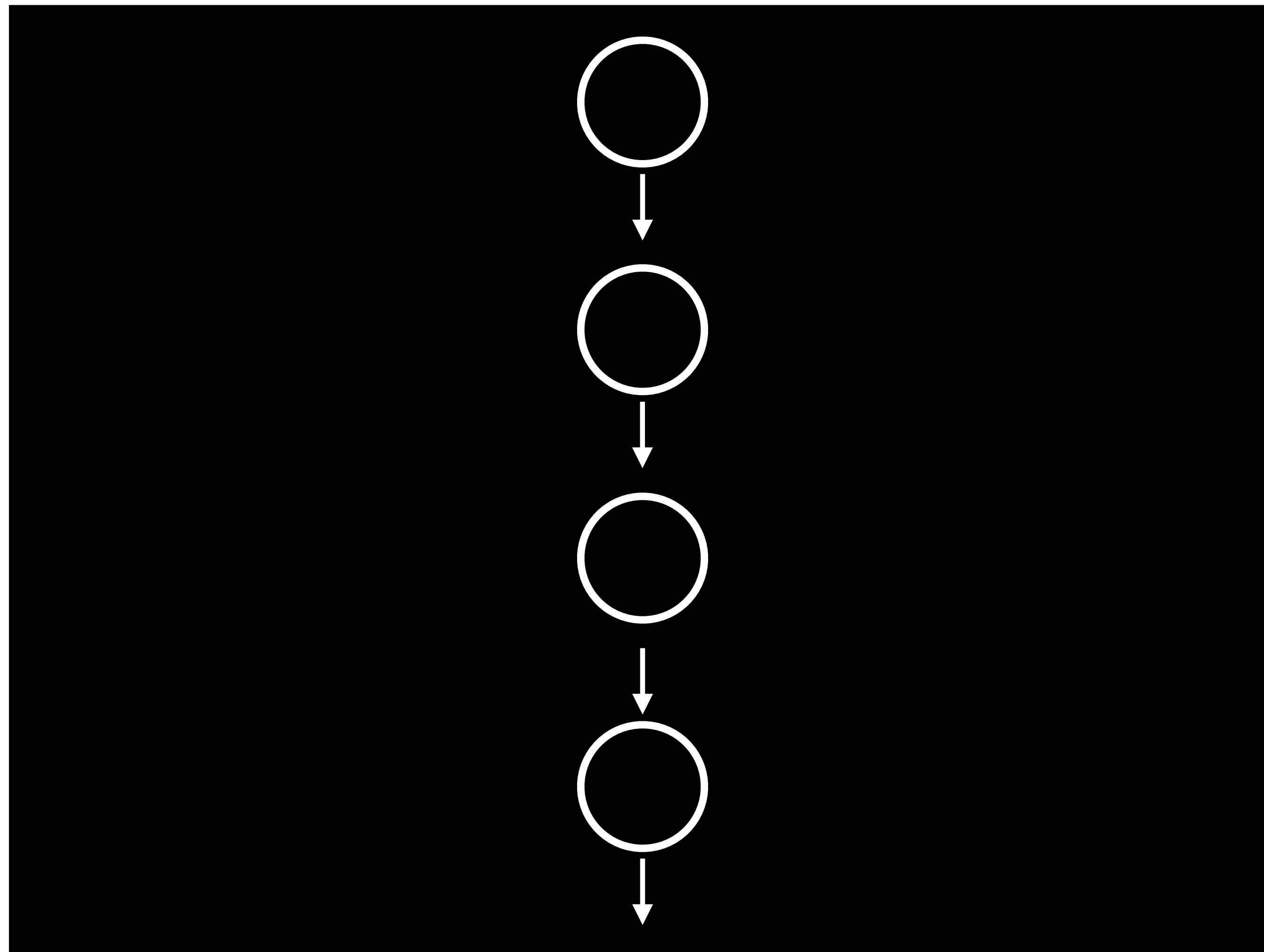
C is a code point or control state that indicates where we are in the program. E is an environment that maps variables to addresses. S is a store that maps addresses to values. And K is a continuation or the program stack.



We start with a program state, which contains the C, E, S, and K components we just described.



From that state and using our small-step semantics, we can calculate the successor to that state.



We continue to calculate the successors to states until we reach the end of the program - unless the program diverges, in which case we keep going forever.

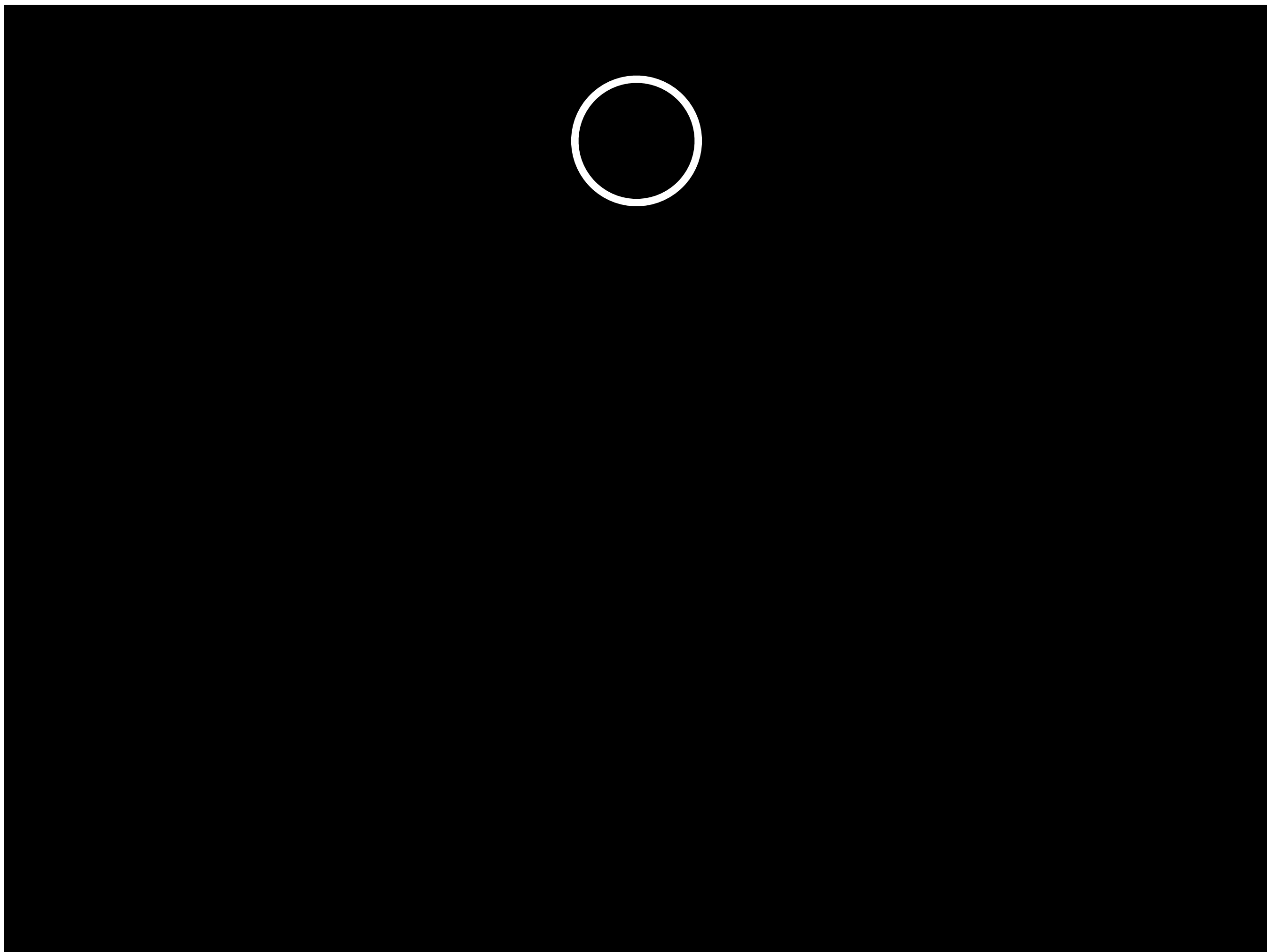
(Van Horn and Might, 2010)

If we want to turn this interpretation into a static analysis, we need to abstract our state space. Happily, Van Horn and Might have already shown that we can abstract the individual components …

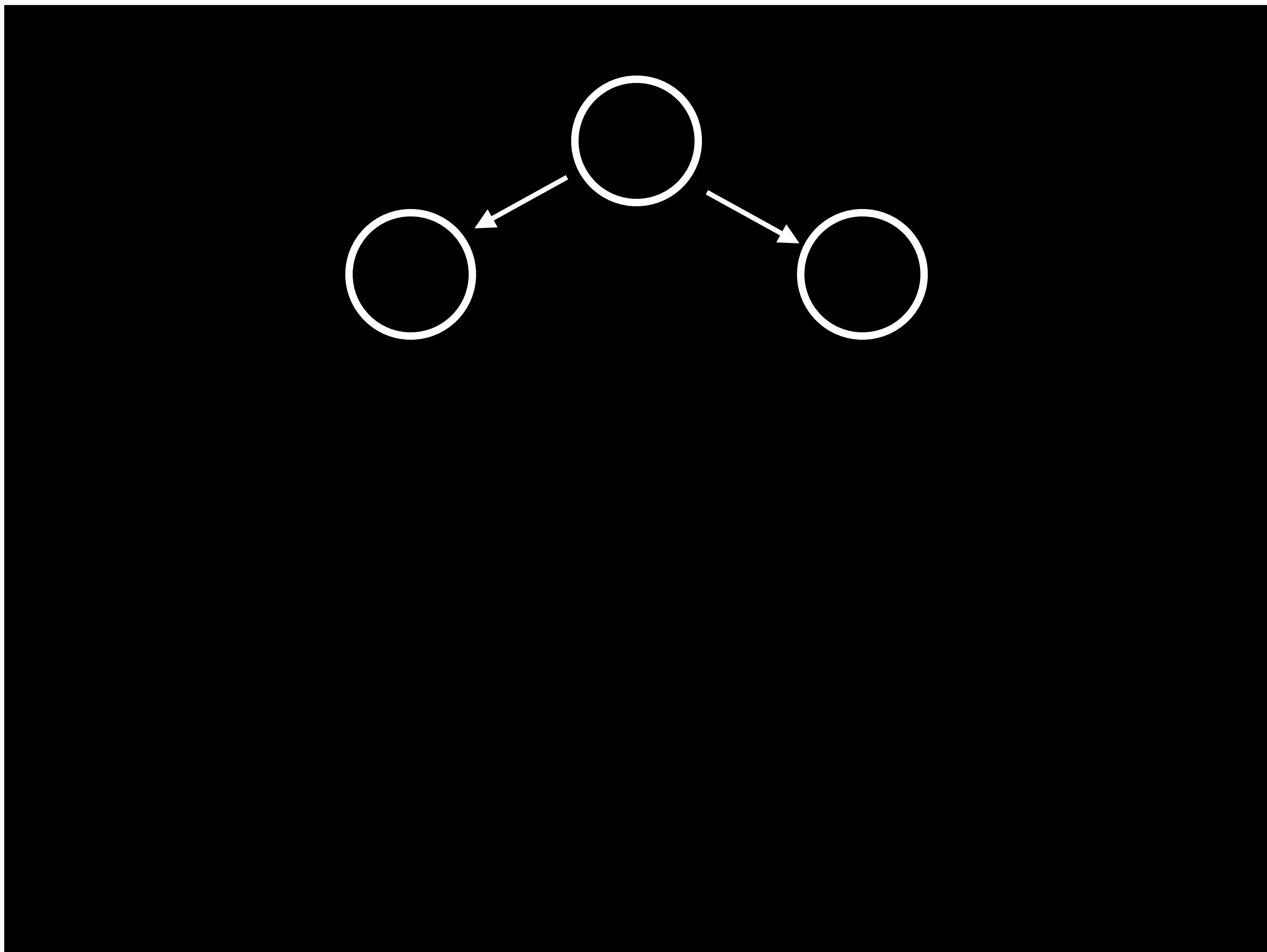
$\mathrm{C}\widehat{\mathrm{E}}\widehat{\mathrm{S}}\widehat{\mathrm{K}}$

(Van Horn and Might, 2010)

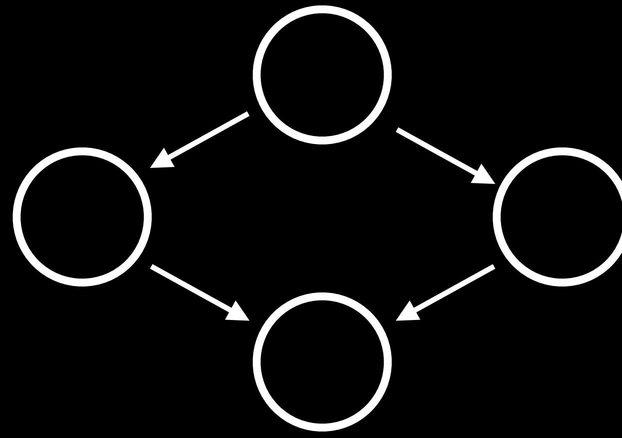
… to create a suitable small-step abstract interpreter.



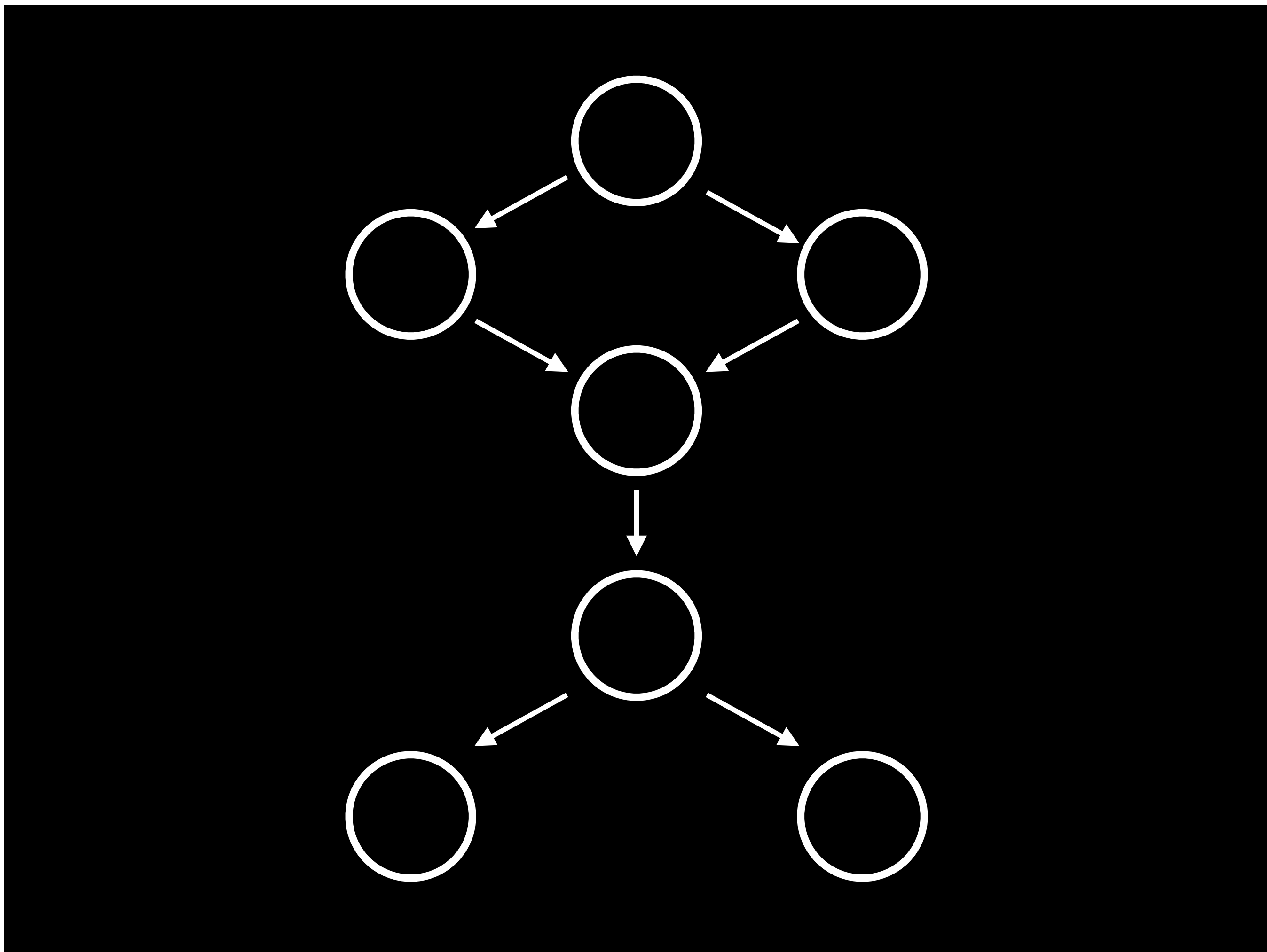
Our abstract transition graph looks a little different from our concrete transition graph.



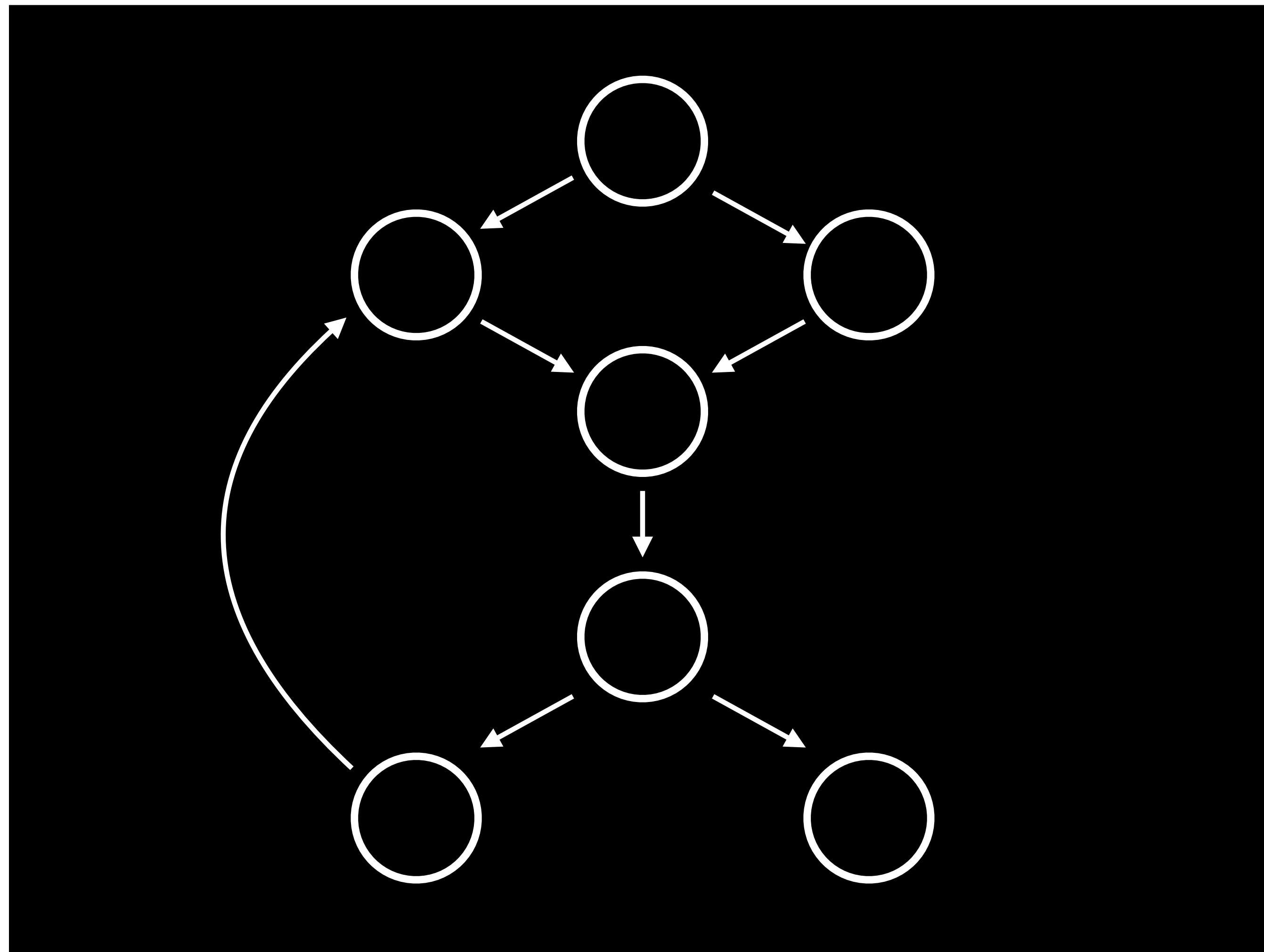
Often, we won't know for certain which way the program's control flow will proceed at a branch. Instructions that may cause branching are not limited to conditional jumps; calling a virtual function, returning from a function, and throwing an exception can all cause branching behavior.



Having states that contain the whole state of an interpreter allows us to explore the state space nondeterministically.



Abstract interpretation continues along each branch until each branch reaches a state with no successors …



… or a state it has already seen before.

This small-step abstract interpreter is guaranteed to terminate. Also, it overapproximates possible program behaviors; that is, it calculates all possible program behaviors - and may also calculate additional, spurious behaviors.

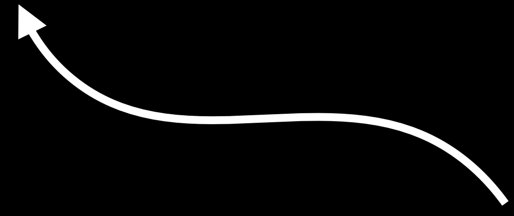
CESK

To add taint tracking to our interpreter, we add a component to each state.

CESKT

T is a taint store.

S = Addr → Val



CESKT

In the same way that a store maps addresses to values, …

S = Addr → Val

CESKT

… the taint store maps addresses to taints. Whenever a value is written to the store, the aggregate taint from the source addresses in the store are combined in the destination address in the taint store.

Thanks to the work of Dorothy Denning, we know that we can use any lattice for taint values. For our purposes today, we’ll just use a binary value.

So if “secret” has special information,

```
x = secret
```

The address in the store associated with secret will be tainted.

`x` = `secret`

When x copies secret’s value in the store, it also copies secret’s taint value in the taint store.

```
0 if (secret) {
1   x = true;
2 } else {
3   x = false;
4 }
5 y = 3;
```

But a taint store only tracks explicit information flows. This program would copy secret to x in effect but our taint tracking mechanism would fail to catch it.

```
0 if (secret) {
1   x = true;
2 } else {
3   x = false;
4 }
5 y = 3;
```

```
0 if (secret) {
1   x = true;
2 } else {
3   x = false;
4 }
5 y = 3;
```

Information flows that rely on a program’s control flow are called implicit information flows.

CESKT

To track control flow, we need to keep track of branches in the control flow whenever those branches depend on sensitive information.

CESKTB

We call this set of branches the context taint set; it is the set of branches that identify taint on the program's context.

CESKTB

$B = \mathcal{P}(C)$

The context taint set is nothing more than a set of code points at which branches occurred.

B = {}

```
0 if (secret) {
1   x = true;
2 } else {
3   x = false;
4 }
5 y = 3;
```

Assignments use information both from source addresses and from the context. So when an assignment happens when there is a non-empty context taint set, we create an implicit taint value with two code points: the code point of the branch (from the context taint set) and the code point at which the assignment occurs.

Similarly, any observable behaviors that occur when there is a context taint are unsafe.

B = {0}

```
0 if (secret) {
1   x = true;
2 } else {
3   x = false;
4 }
5 y = 3;
```

So when we branch on line 0, we add 0 to the context taint set.

B = {0}

```
0 if (secret) {
1   x = true;      (0,1)
2 } else {
3   x = false;
4 }
5 y = 3;
```

On line 1, x gets a taint value that indicates the location of the branch (from the context taint set) and of the assignment

B = {0}

```
0 if (secret) {
1    x = true;        (0,1)
2 } else {
3    x = false;       (0,3)
4 }
5 y = 3;
```

On line 3, the same thing happens

B = {0}

```
0 if (secret) {
1   x = true;        (0,1)
2 } else {
3   x = false;       (0,3)
4 }
5 y = 3;             (0,5)
```

But y also gets an implicit taint value, even though it is clearly unaffected by the value of secret.

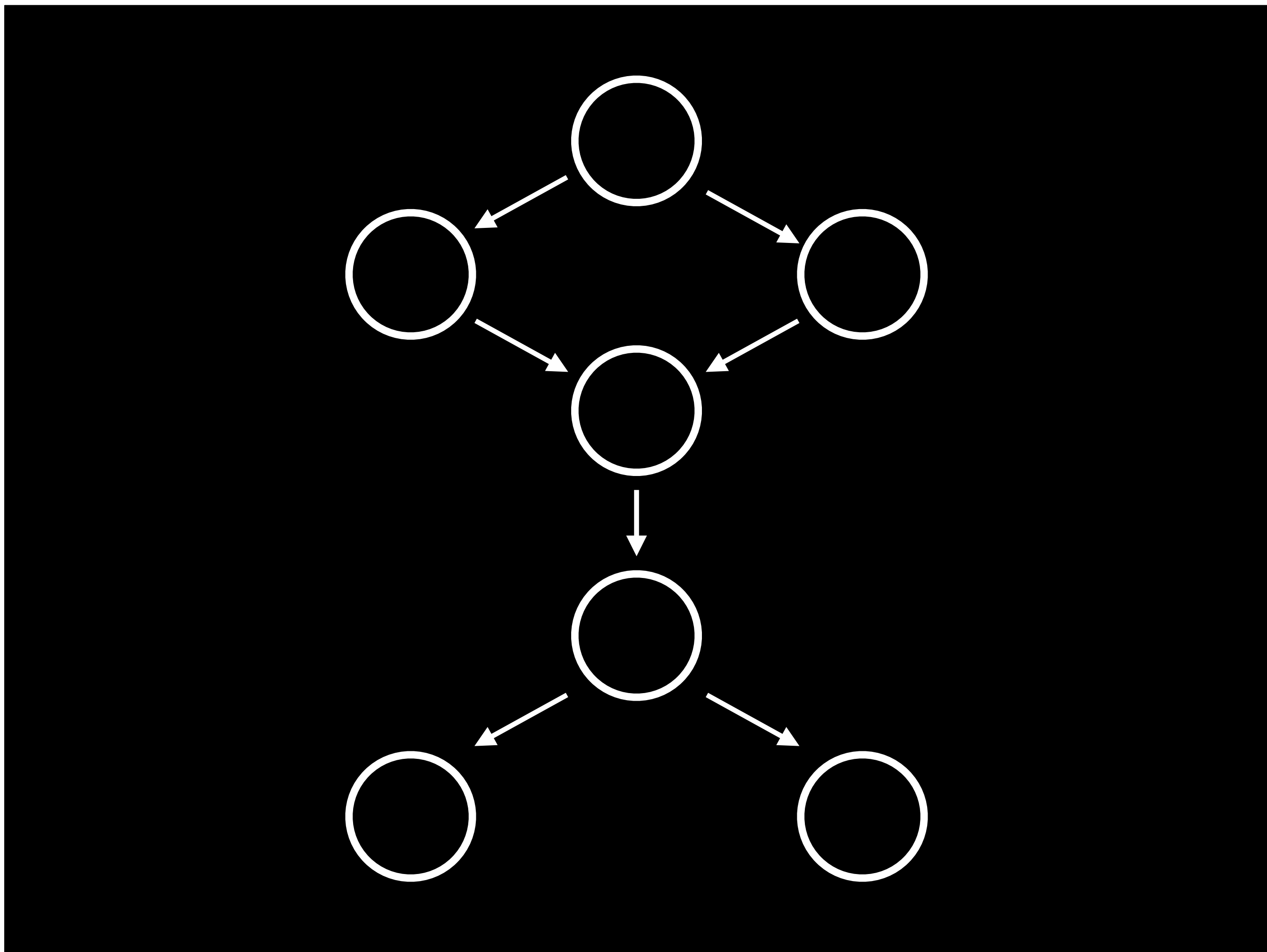
It’s fairly easy in an if statement in Java, as the syntax shows that the if statement ends by line 4. But what about conditional jumps? What about exceptional flow?

Denning & Denning

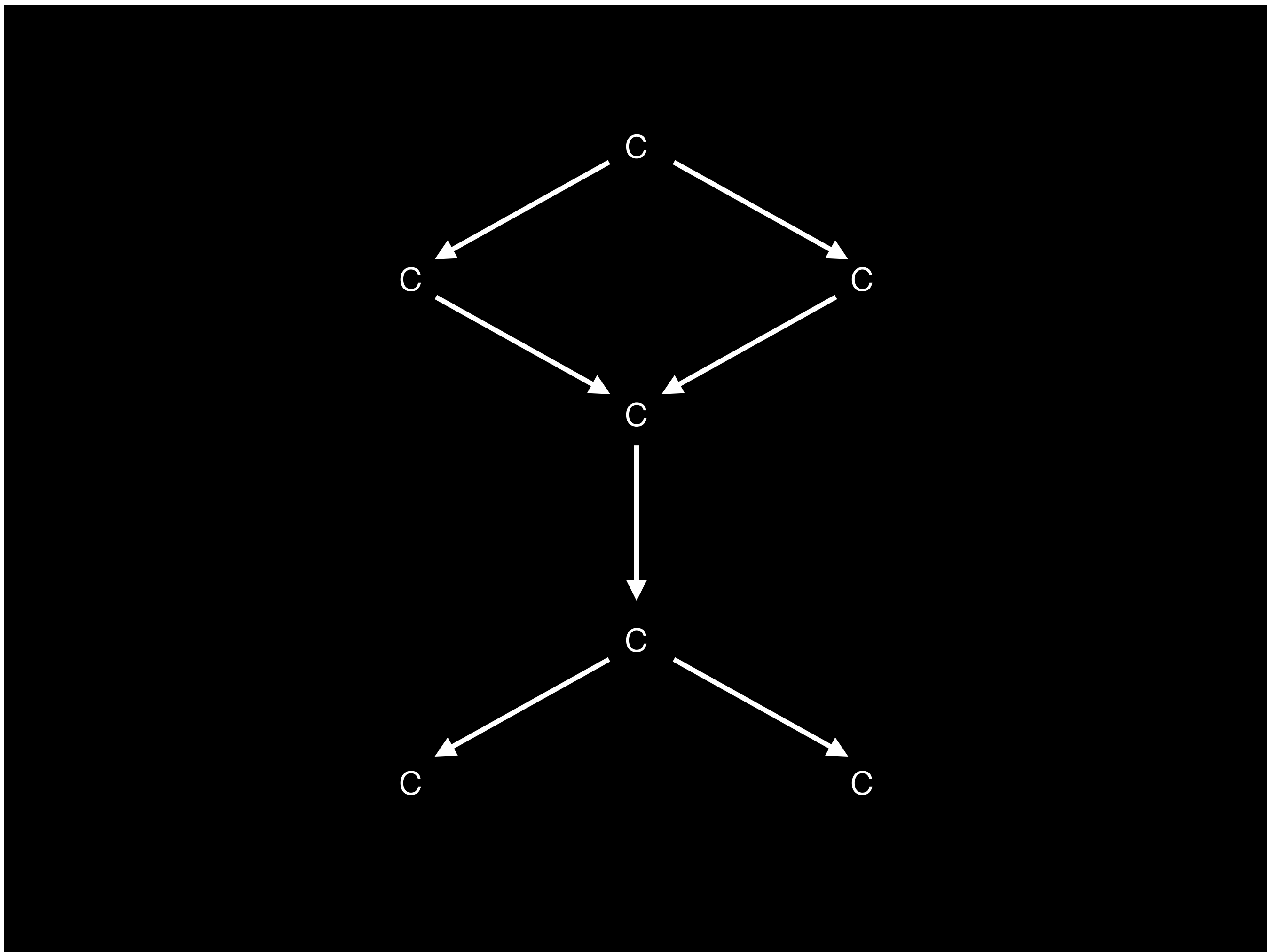
So what do Denning and Denning say?

Use the control-flow graph

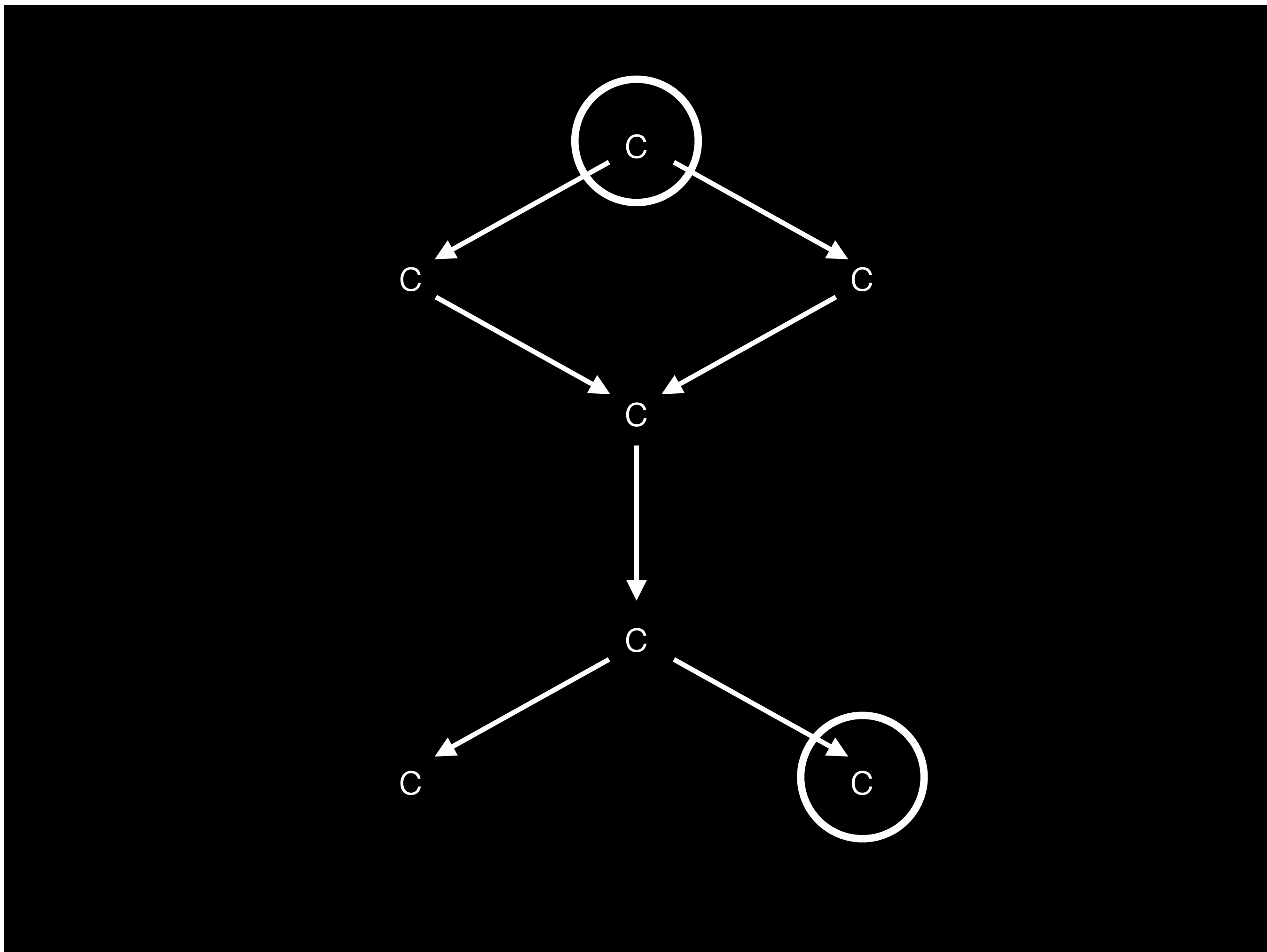
Denning and Denning also have an answer for this problem: use the control flow graph to find the immediate postdominator of the branch, or the first node that appears along every path from the branch to the exit node. At the IPD, the branch is over. Using the IPD means that we limit our proof to termination-insensitive non-interference, as it only examines paths that terminate.



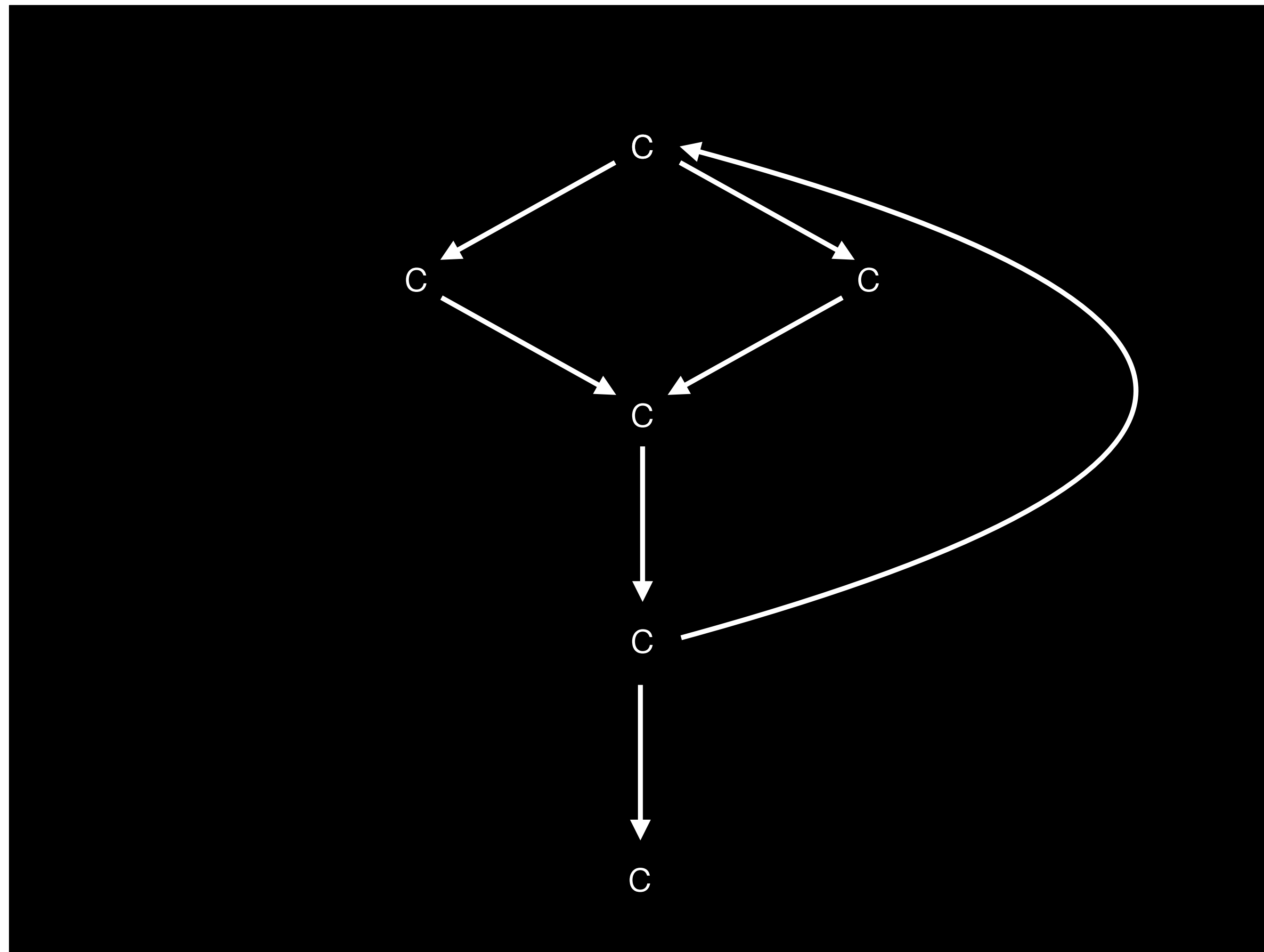
We already have an abstract transition graph



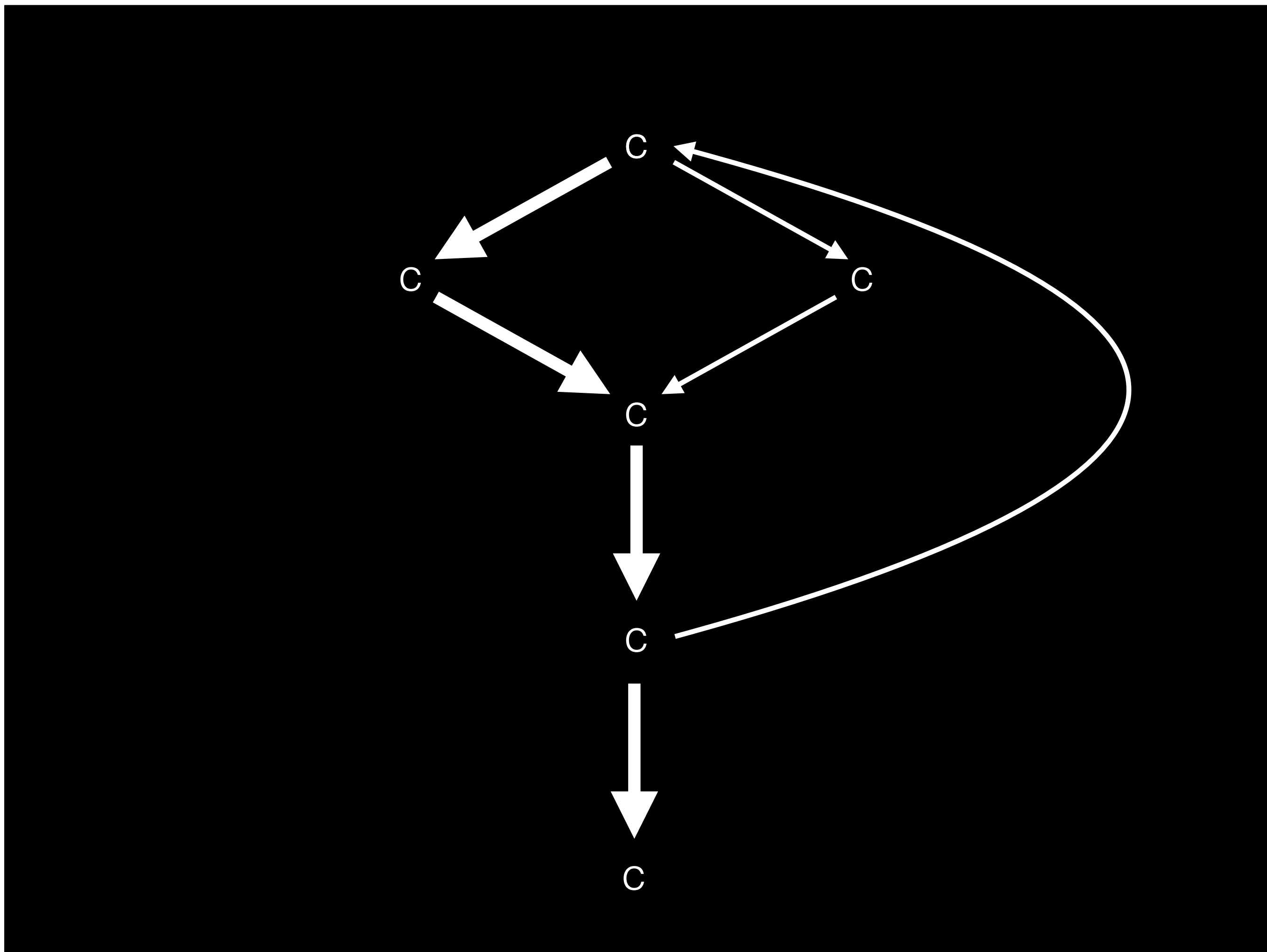
We can project it to a CFG



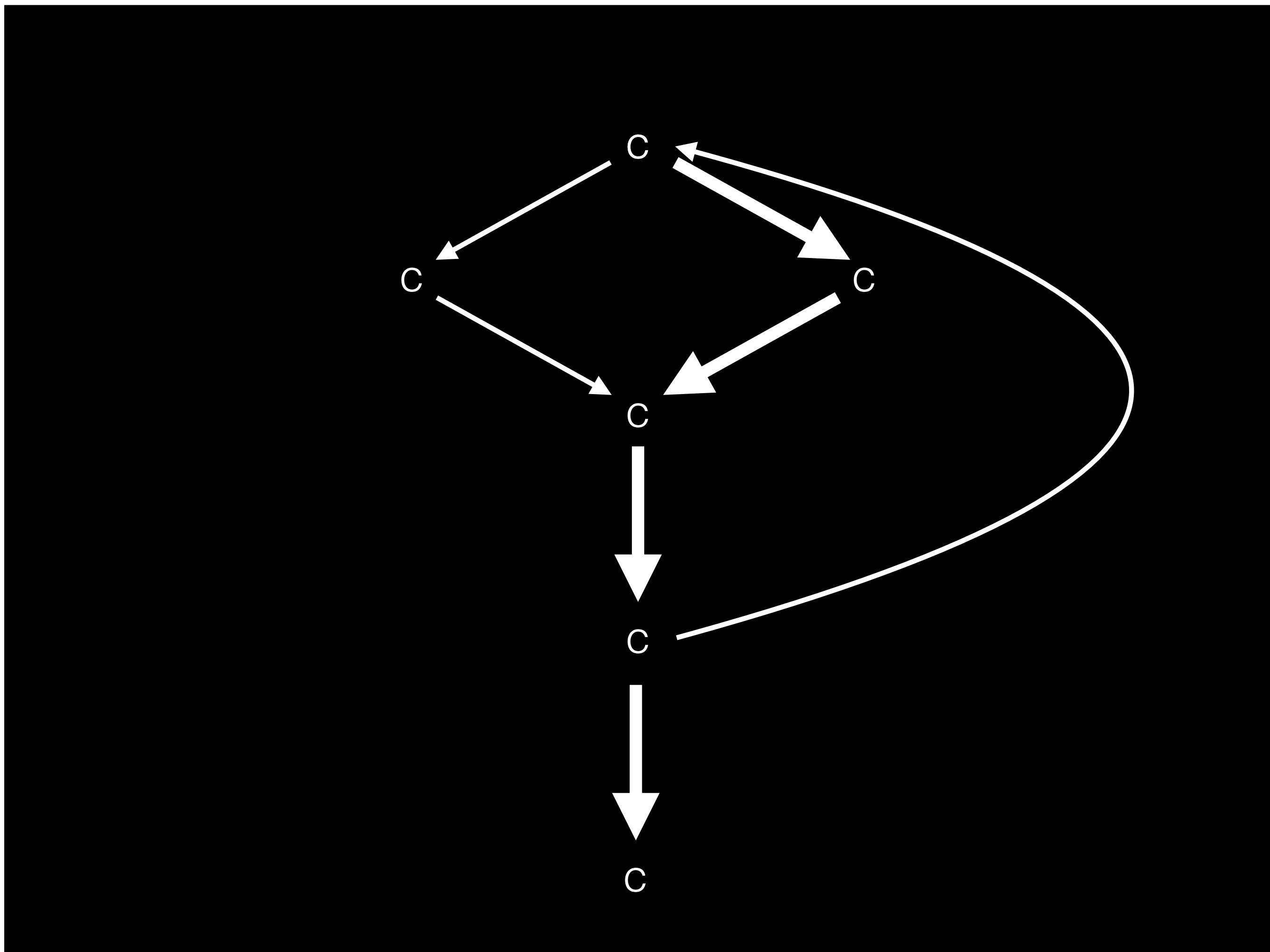
Of course, different states might share the same code point



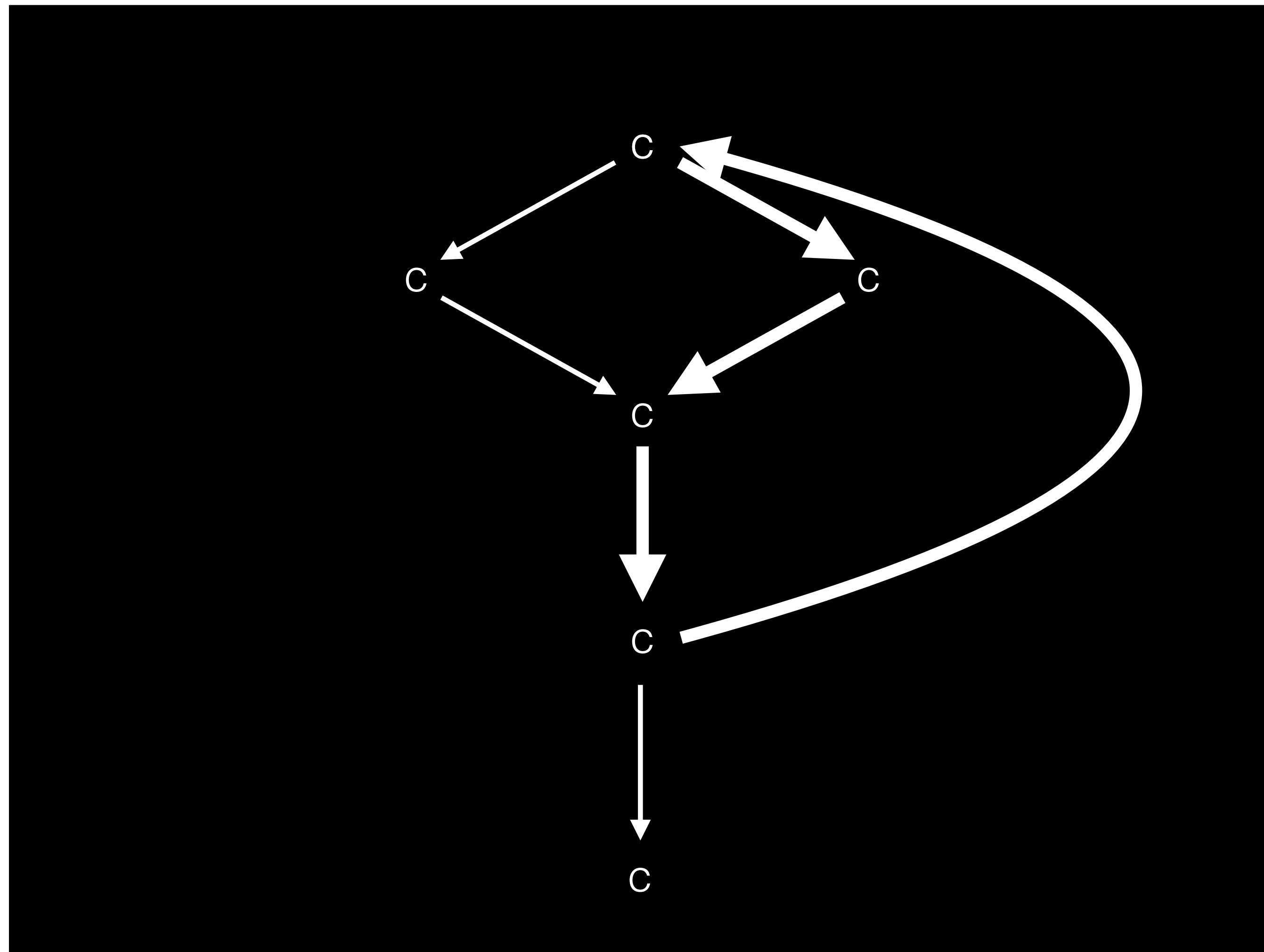
Which means that our CFG may be structurally simpler than our state graph. With a control-flow graph, we can find the immediate postdominator of a line of code. The immediate postdominator of node A is the first node encountered along every path from A to the exit.



In this graph, we can go from the topmost node to the exit this way …

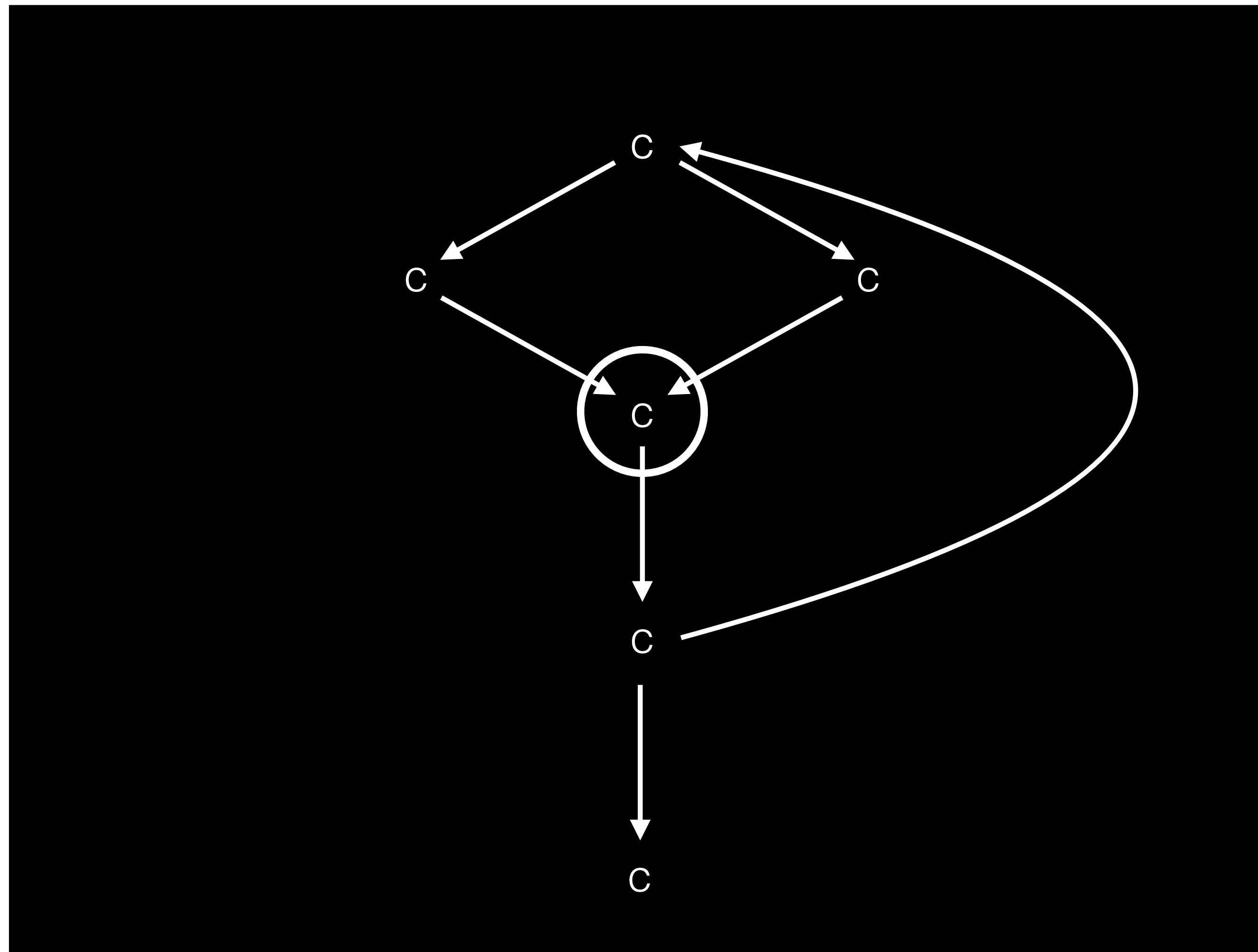


… or this way …



… or loop and then proceed along any of the paths we’ve explored.

The nature of postdominance is that it doesn’t consider infinite loops, so our proof of non-interference is a proof of termination-insensitive non-interference. Adding termination analysis would allow us to prove unqualified non-interference.

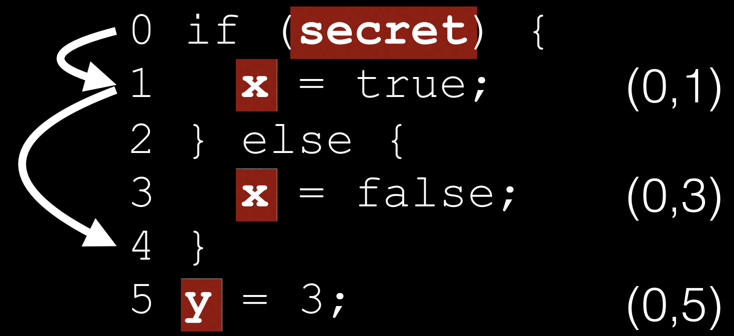


All of our paths go through this node. They also go through other nodes, which are also postdominators, but this is the first. So it is the immediate forward dominator or immediate postdominator. Equivalently, we say that this node postdominates the node at the top.

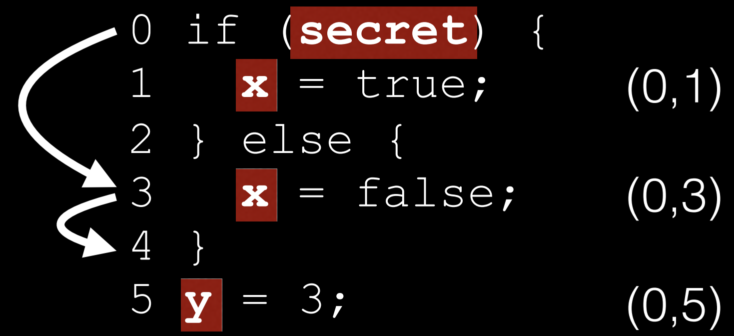
```
0 if (secret) {
1   x = true;        (0,1)
2 } else {
3   x = false;       (0,3)
4 }
5 y = 3;             (0,5)
```

Let’s walk through this in our code sample.

There are two paths from 0:



The true branch, which goes to line 1 and then to line 4, and …



… the false branch, which goes to line 3 and then to line 4.

```
0 if (secret) {
1     x = true;        (0,1)
2 } else {
3     x = false;       (0,3)
4 }
5 y = 3;               (0,5)
```

Accordingly, we remove or ignore the implicit taint (0,5).

```
0 if (secret) {
1     x = true;        (0,1)
2 } else {
3     x = false;       (0,3)
4 }
5 y = 3;               (0,5)
```

So we remove or ignore the implicit taint (0,5).

Of course, there is still the question of abstraction.

As we said, we already know how to abstract CESK machines from Van Horn and Might.

What remains is to abstract the taint store and the context taint set. The context taint set is just a set of code points and needs no abstraction.

$C\hat{E}\hat{S}\hat{K}\hat{T}B$

What remains is abstraction of the taint store.

$$C\widehat{E}\widehat{S}\widehat{K}\widehat{T}B$$

$$\widehat{T} = \widehat{Addr} \rightarrow \widehat{Taint}$$

The taint store must be abstracted as is the value store; it needs to map abstract addresses to abstract taint values. Binary taint values need no abstraction, but richer taint values might.

We have an analysis

Now that we've taken Denning and Denning's work and paired it with that of Cousot and Cousot (among others), we have a small-step abstract interpreter that tracks taint values. But we haven't demonstrated that it identifies all possible information leaks.

Now that we’ve taken Denning and Denning’s work and paired it with that of Cousot and Cousot (among others), we have a small-step abstract interpreter that tracks taint values. But we haven’t demonstrated that it identifies all possible information leaks.

Almost

It turns out that we can construct a counterexample, although it's a little convoluted.

```
private boolean secret;
```

Consider two different executions of a function. In one, secret is true. In the other, secret is false.

```
private boolean secret;

void printSecret(int frame) {
  if (frame == 0)
    printSecret(1);
```

If we create a function that recurs on itself (with a value that differs at different stack frames) …

```
private boolean secret;

void printSecret(int frame) {
  if (frame == 0)
    printSecret(1);
  else if (secret)
    return;
```

… and returns conditionally on the secret value, then we have created a situation where two different traces would end up at the same code point but are on different levels in the stack and, consequently, have different but untainted local values.

```
private boolean secret;

void printSecret(int frame) {
  if (frame == 0)
    printSecret(1);
  else if (secret)
    return;

  if (frame == 1) {
    System.out.print("not ");
    return;
  }
```

It’s then trivial to behave differently based on one of those local values.

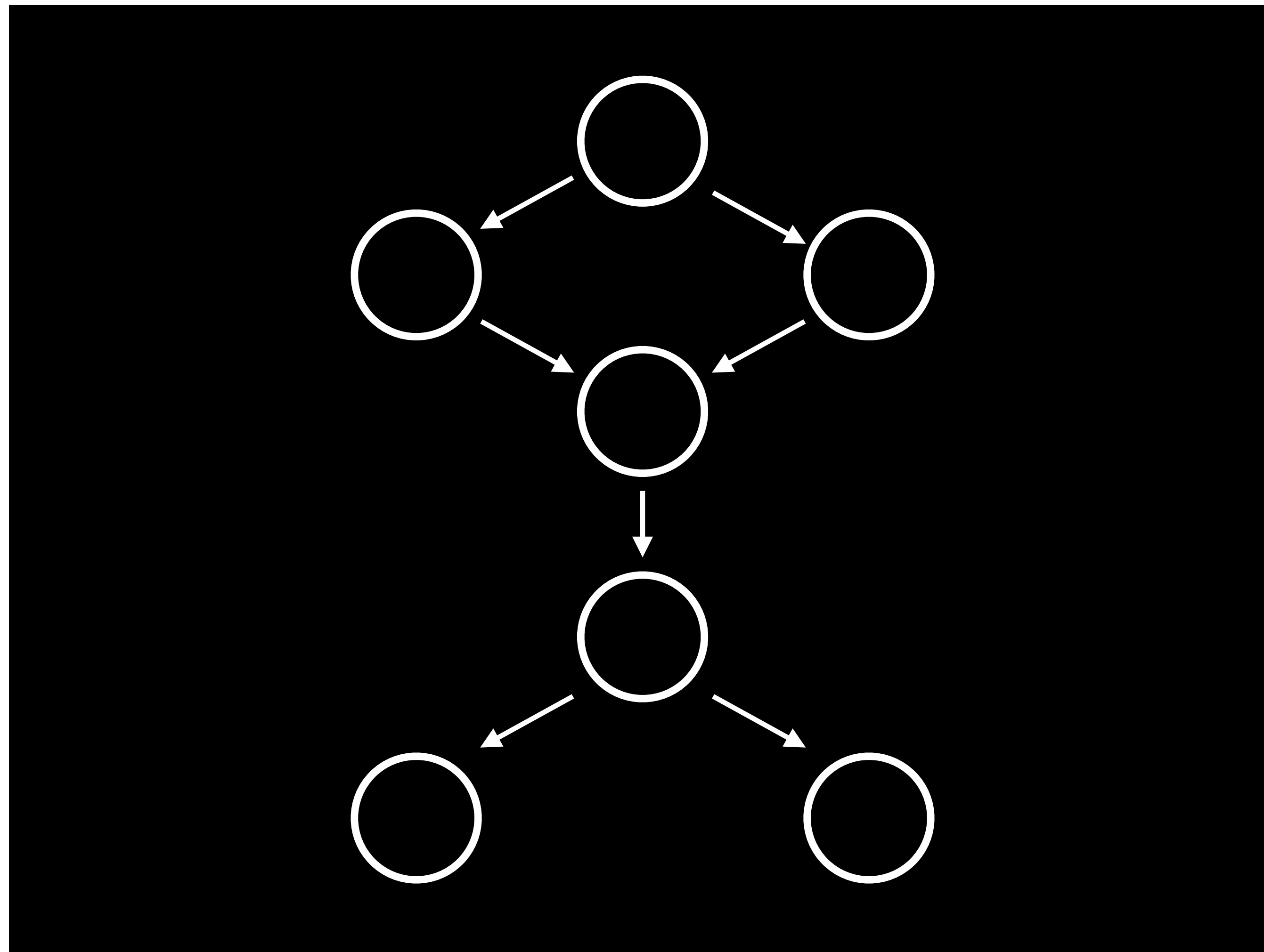
```
private boolean secret;

void printSecret(int frame) {
  if (frame == 0)
    printSecret(1);
  else if (secret)
    return;

  if (frame == 1) {
    System.out.print(“not “);
    return;
  }

  System.out.println(“true”);
}
```

Predictably, the execution of printSecret(0) where secret is true results in the text “true” being printed. The other execution, where secret is false, results in the text “not true” being printed.

This technique allows us to leak a bit without detection. Putting this in a loop would allow for arbitrary amounts of leakage.

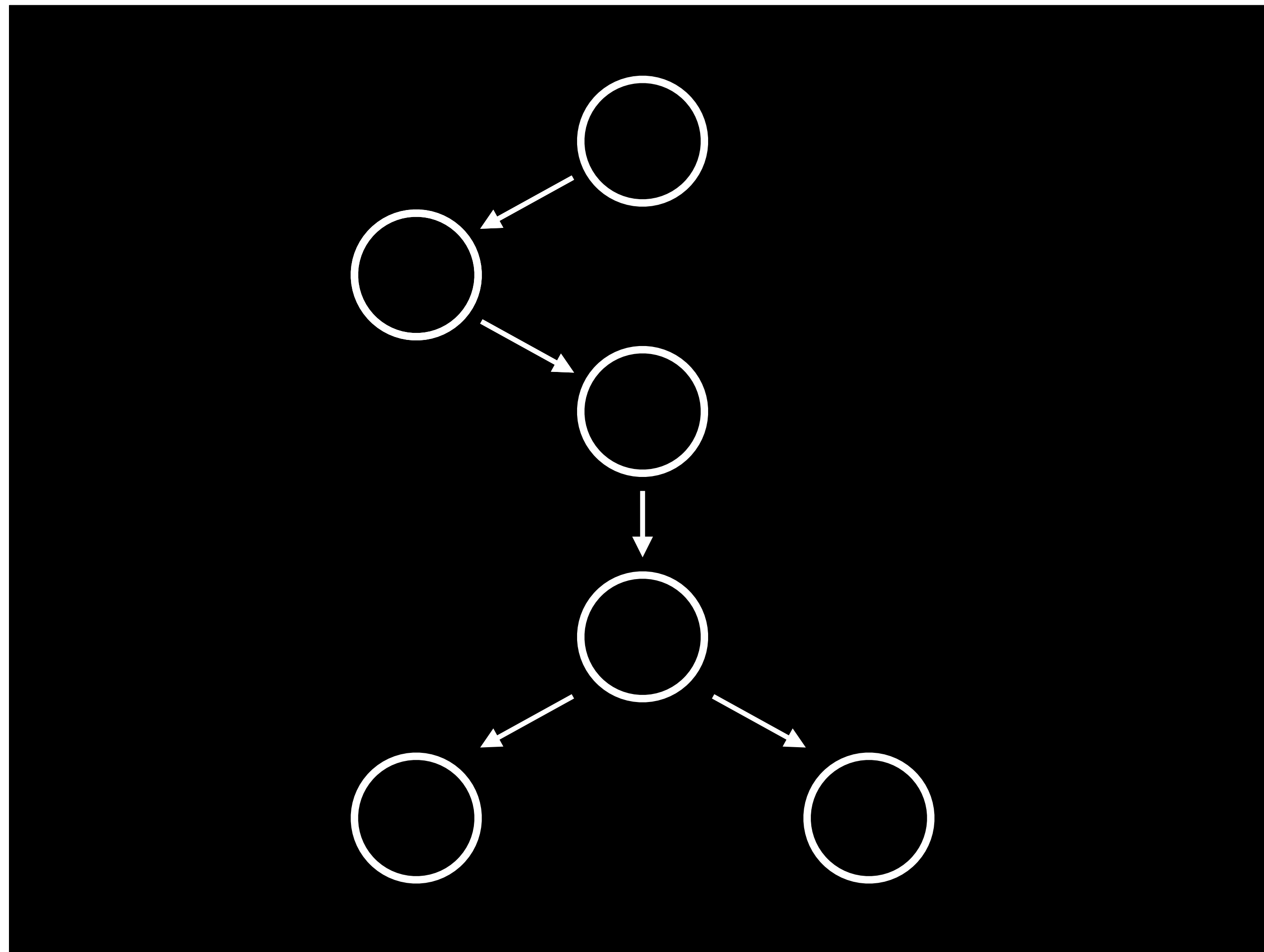
D&D

The problem with the system, as formulated, is that the projection to a control flow graph is insufficiently precise. Two executions reached the same code point but in different contexts. We need information about the stack.



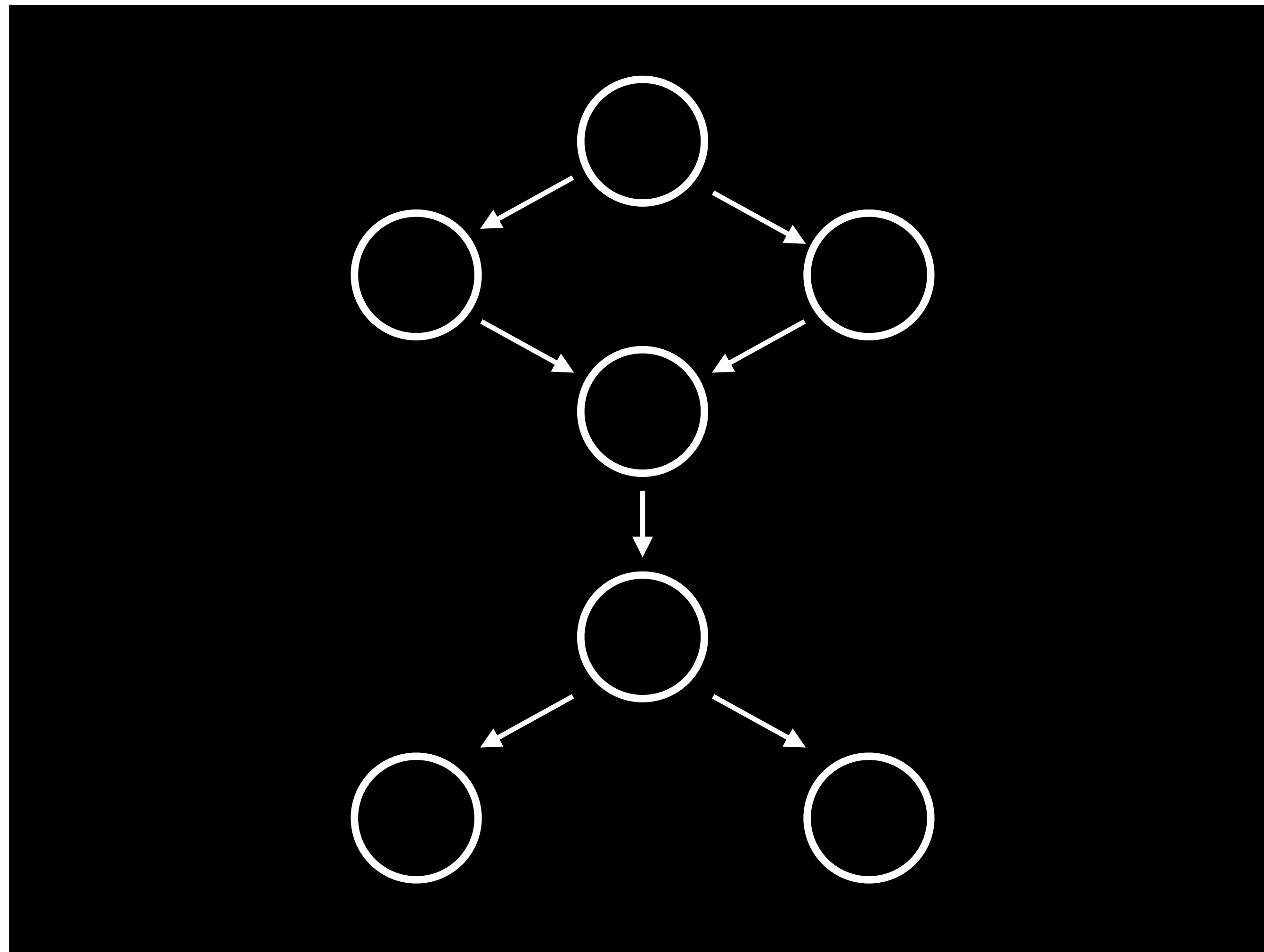
The intuition is to keep both code points and continuations when we project from our abstract transition graph.

We want to use as little information as possible in creating this graph, because less information means we can merge nodes together …



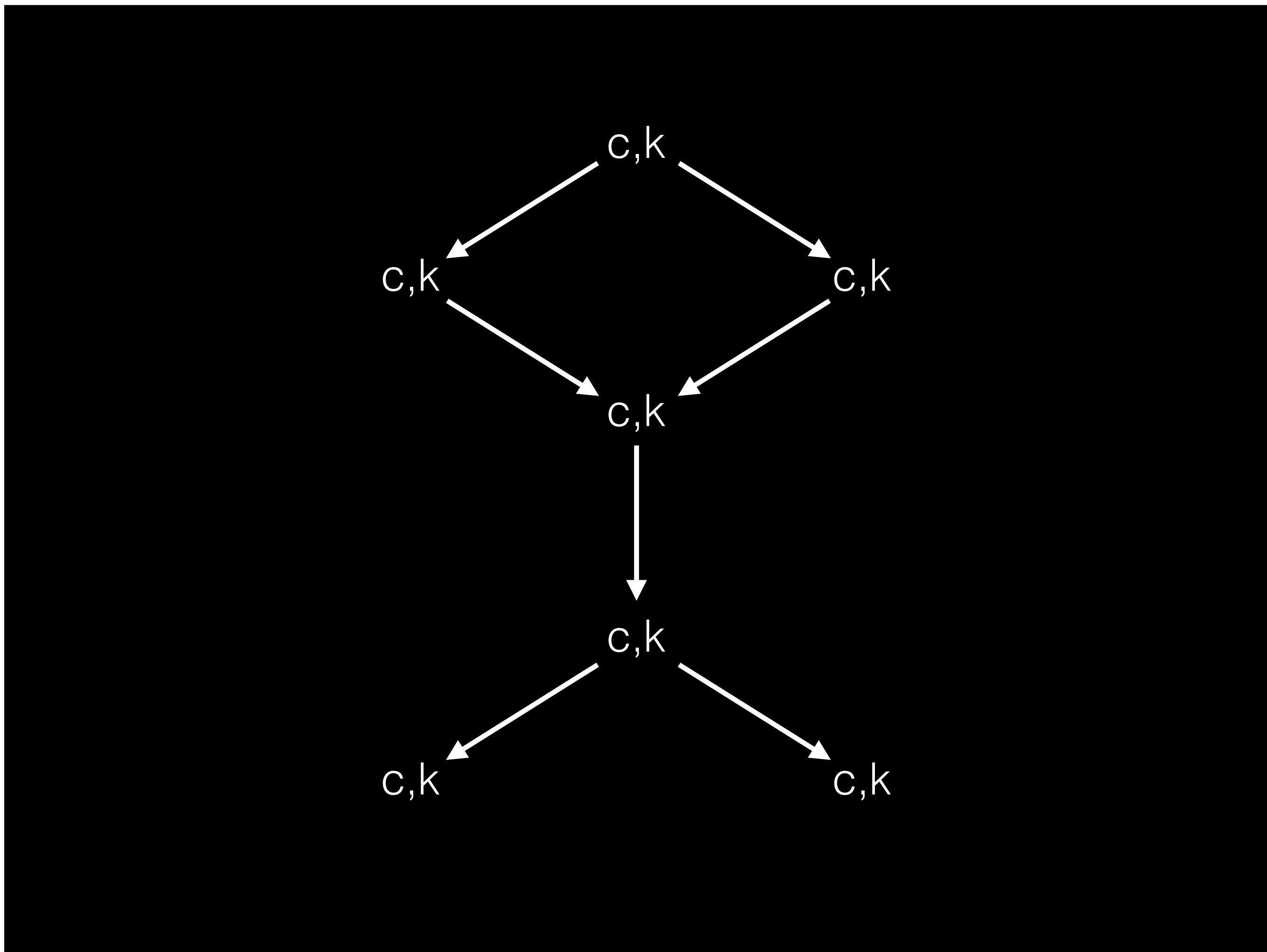
… and merging nodes can lead to an earlier postdominator. This, in turn, means that taint will not propagate as far.

What we want is the least precise execution point graph possible that still allows us to prove non-interference.

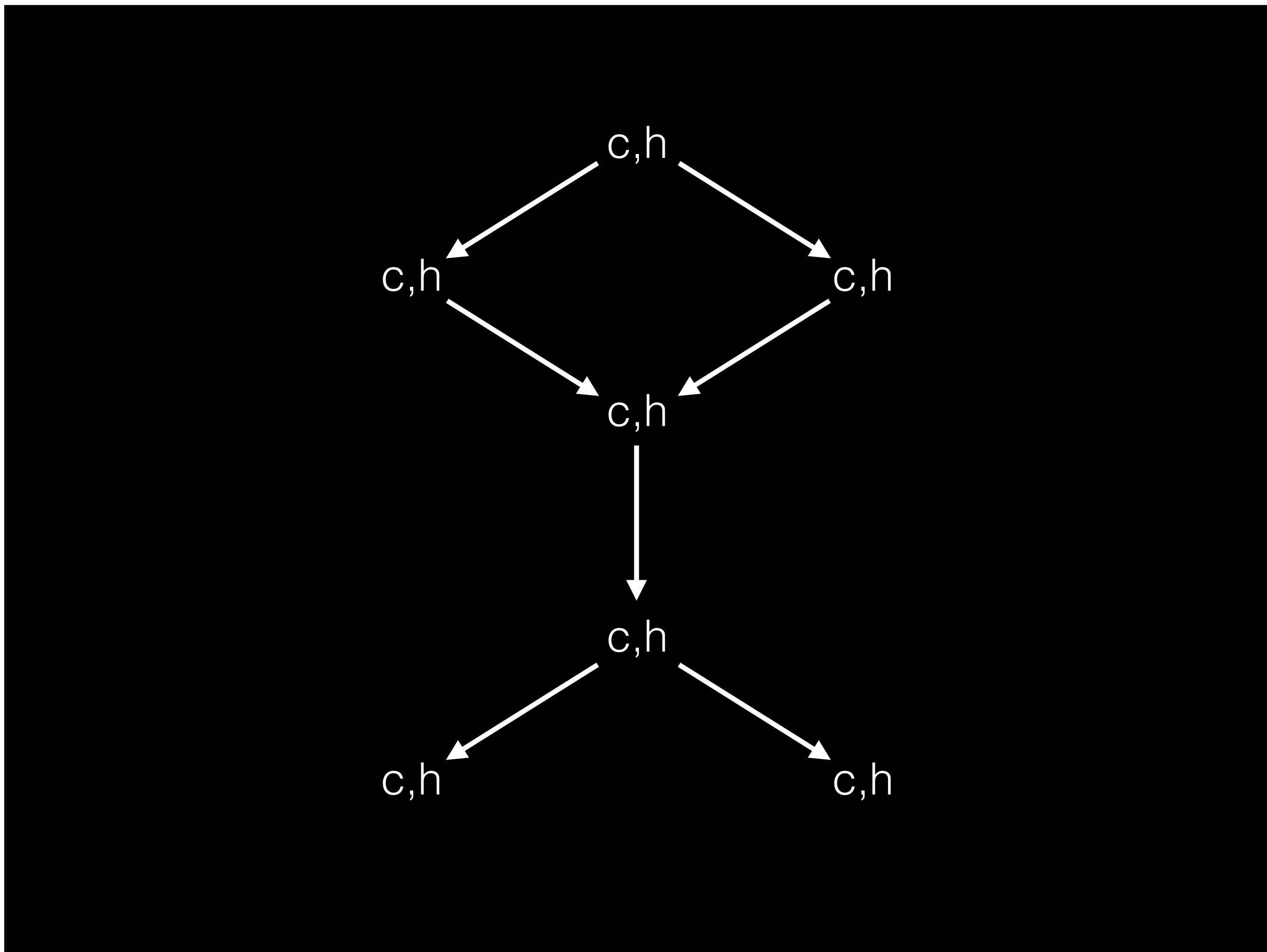


The intuition is to keep both code points and continuations when we project from our abstract transition graph.

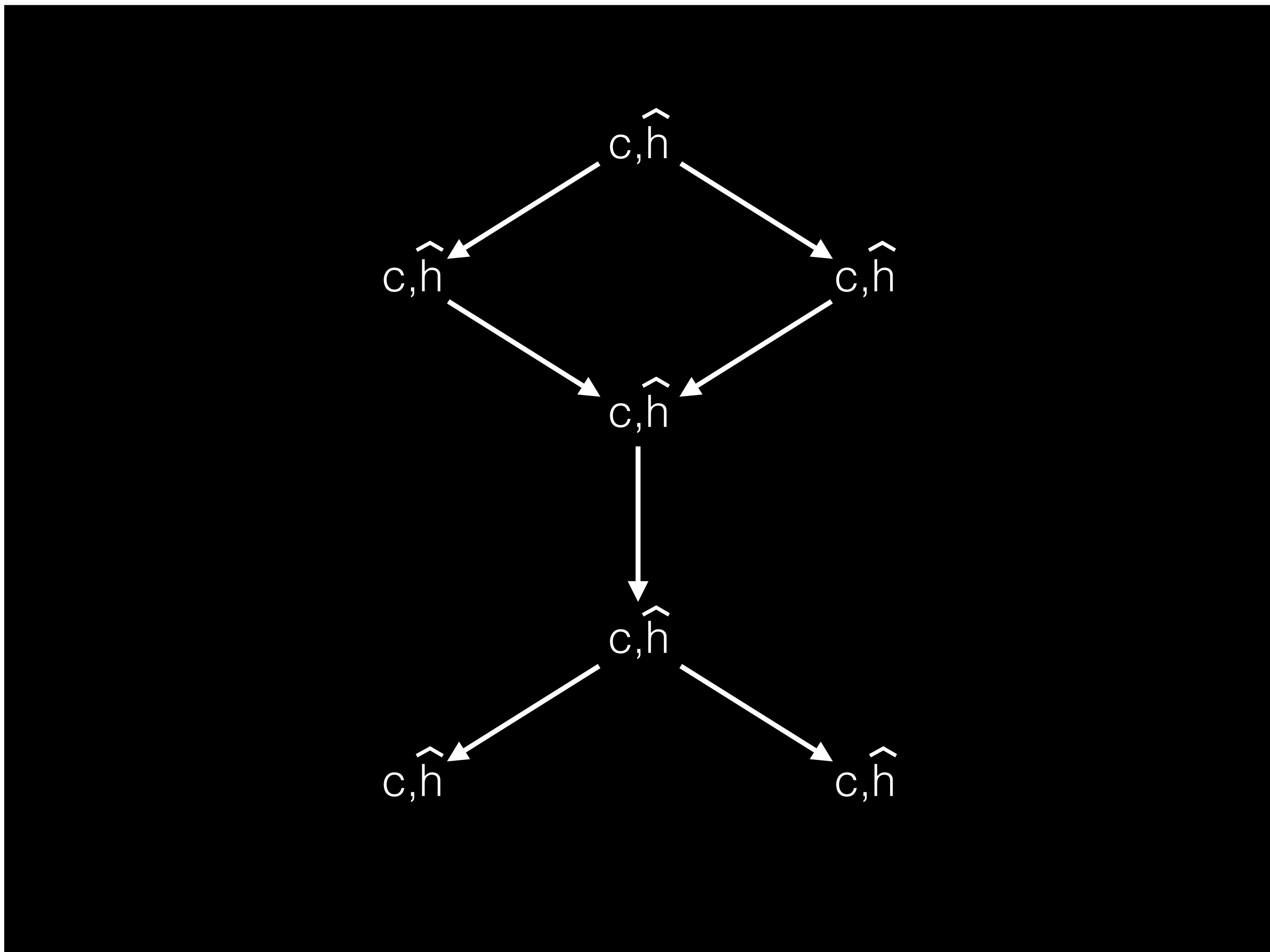
We want to use as little information as possible in creating this graph, because less information means we can merge nodes together …



We actually want to make our graph as imprecise as possible; the more merging there is in the graph, the faster control flows converge. And the faster control flows converge, the fewer taints we propagate.



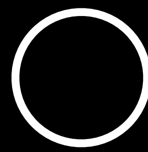
So we actually project to code points and stack heights, which are trivially calculated from continuations. We call this graph the “execution point graph”. It might have been more properly called a control flow graph, but the term was already taken.



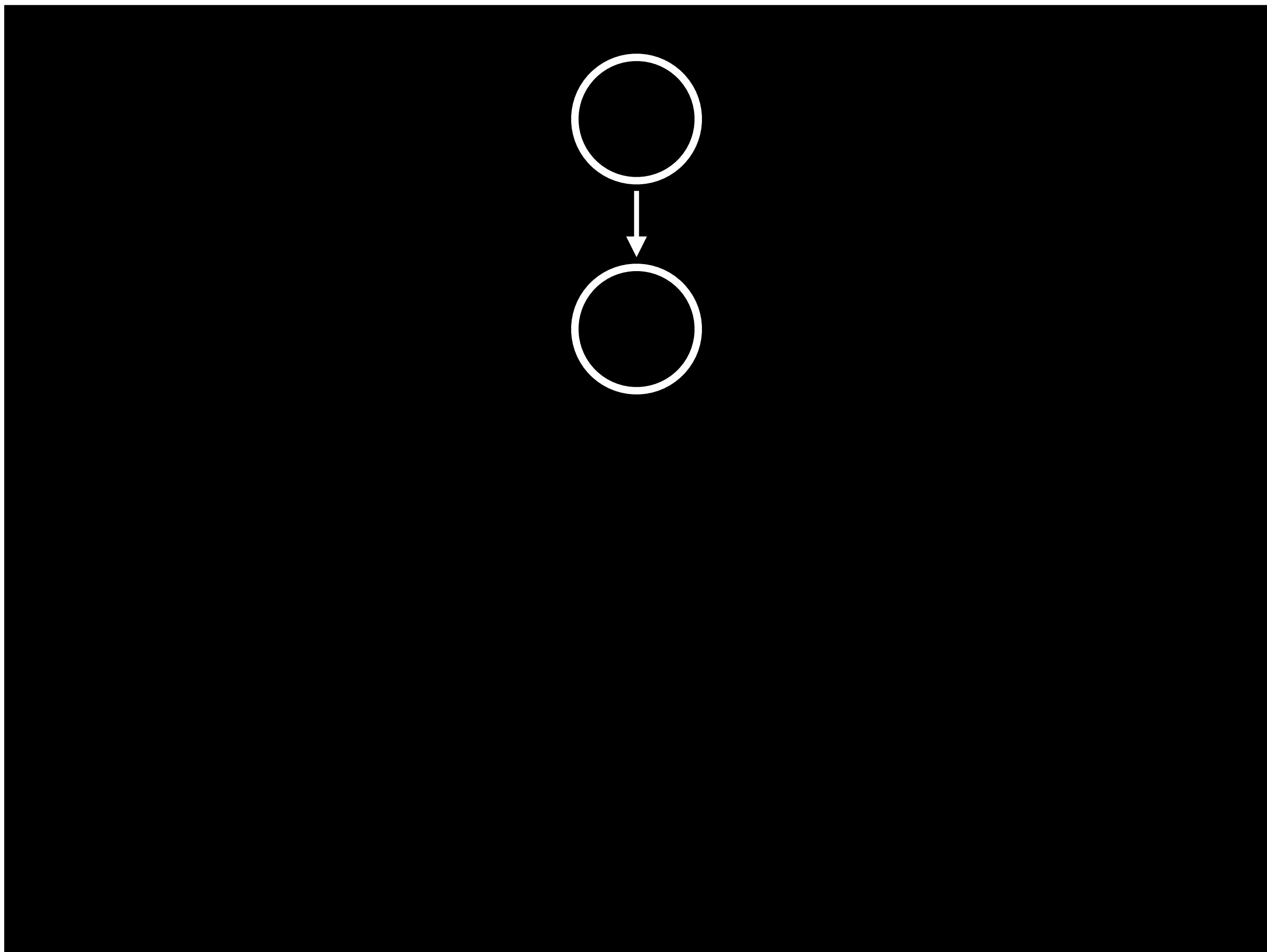
Of course, the stack height has to be abstracted. Happily, we have already abstracted stacks, so most of the work is taken care of. But even systems like CFA2 and PDCFA can't always calculate the precise height of the stack, so our abstract stack height needs to include a special value for indeterminate stack heights. Nodes with indeterminate stack heights are never postdominators for our purposes.

Our context taint set now needs to store execution points instead of just code points.

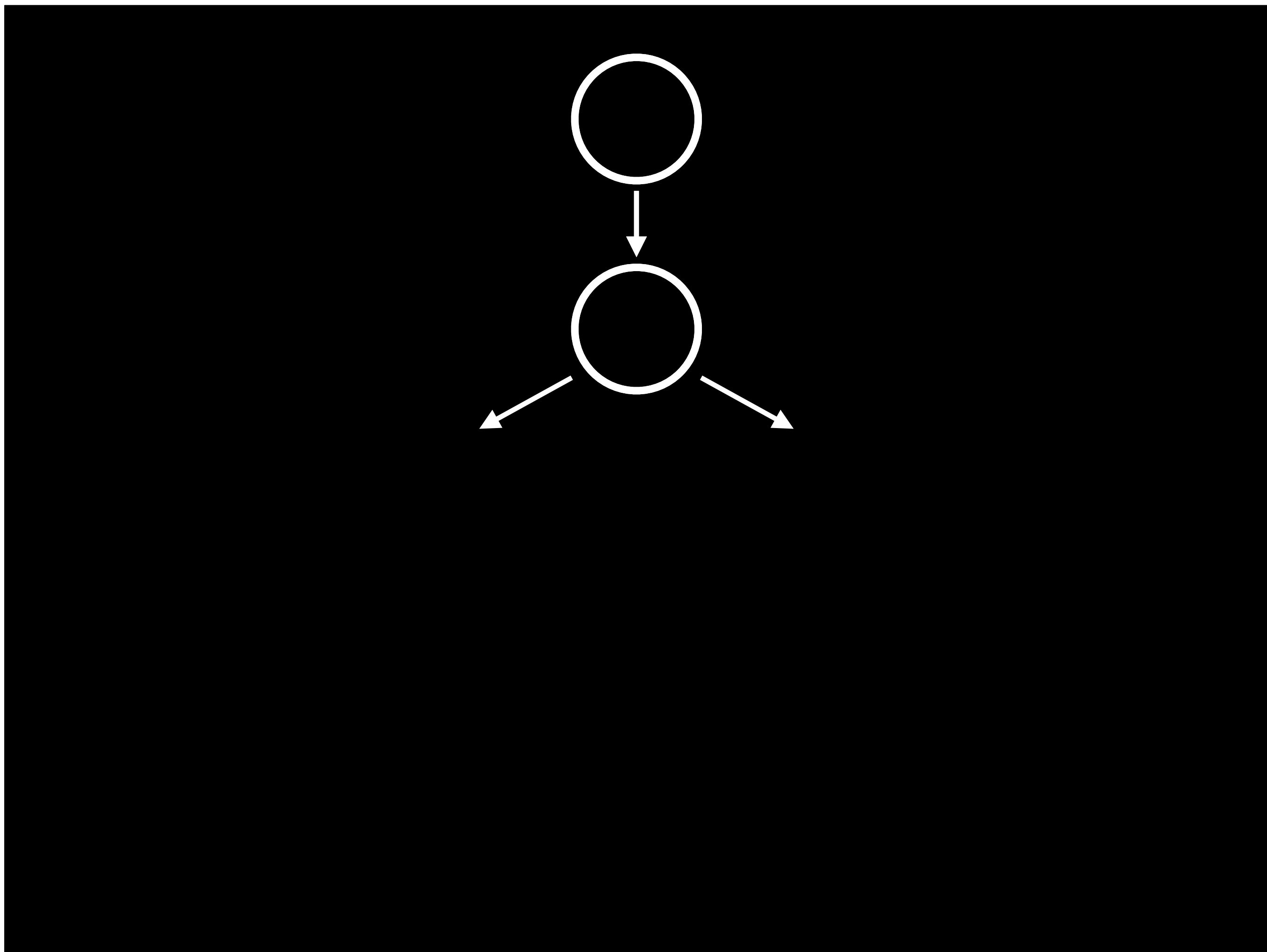
The proof (which is in the paper) uses a weakened bisimulation notion. Two program traces must act the same except when they've branched on a sensitive value. In that case, a value in the context taint set labels observable behaviors as unsafe.



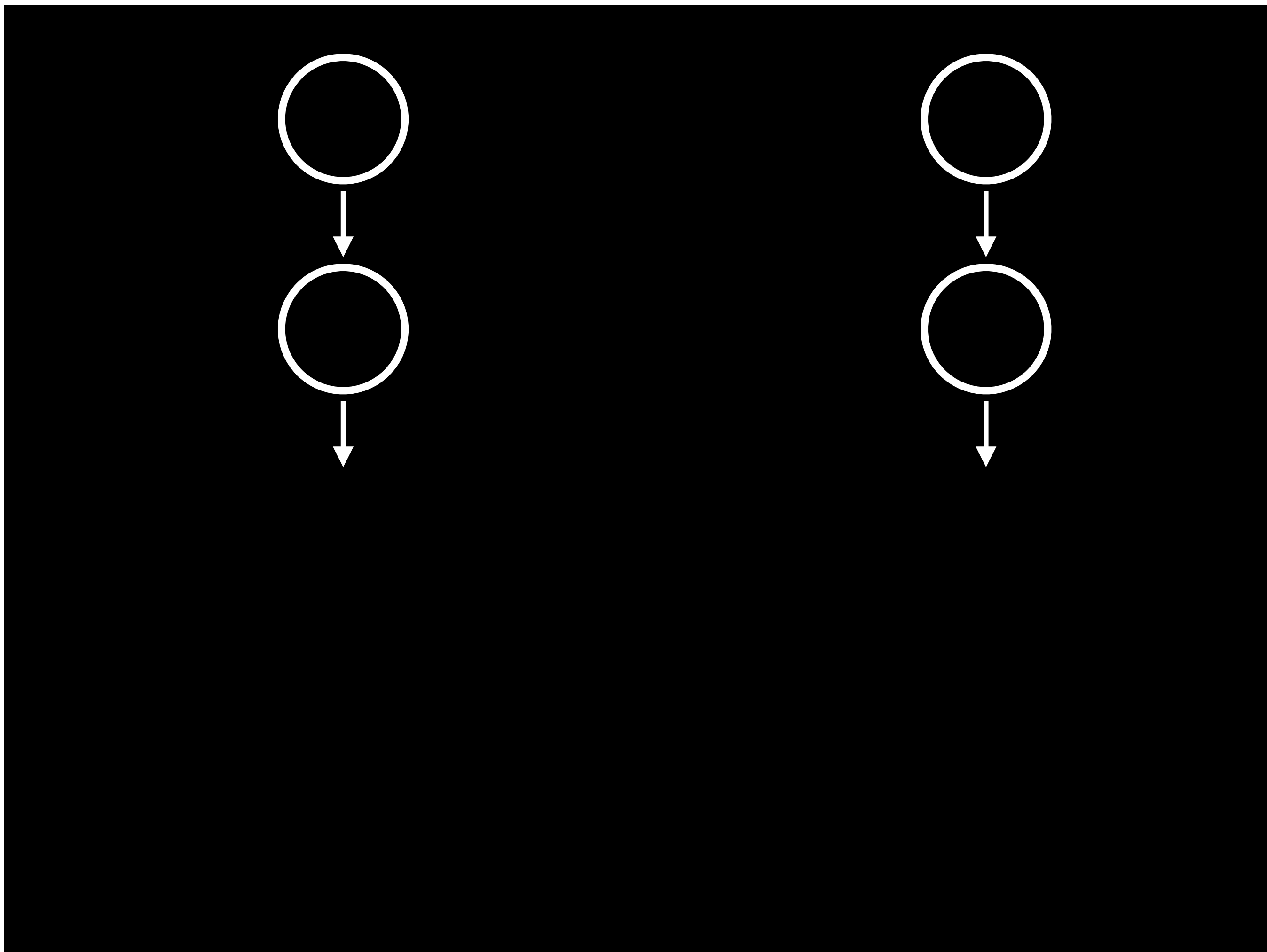
Our abstract interpreter’s exploration finds states.



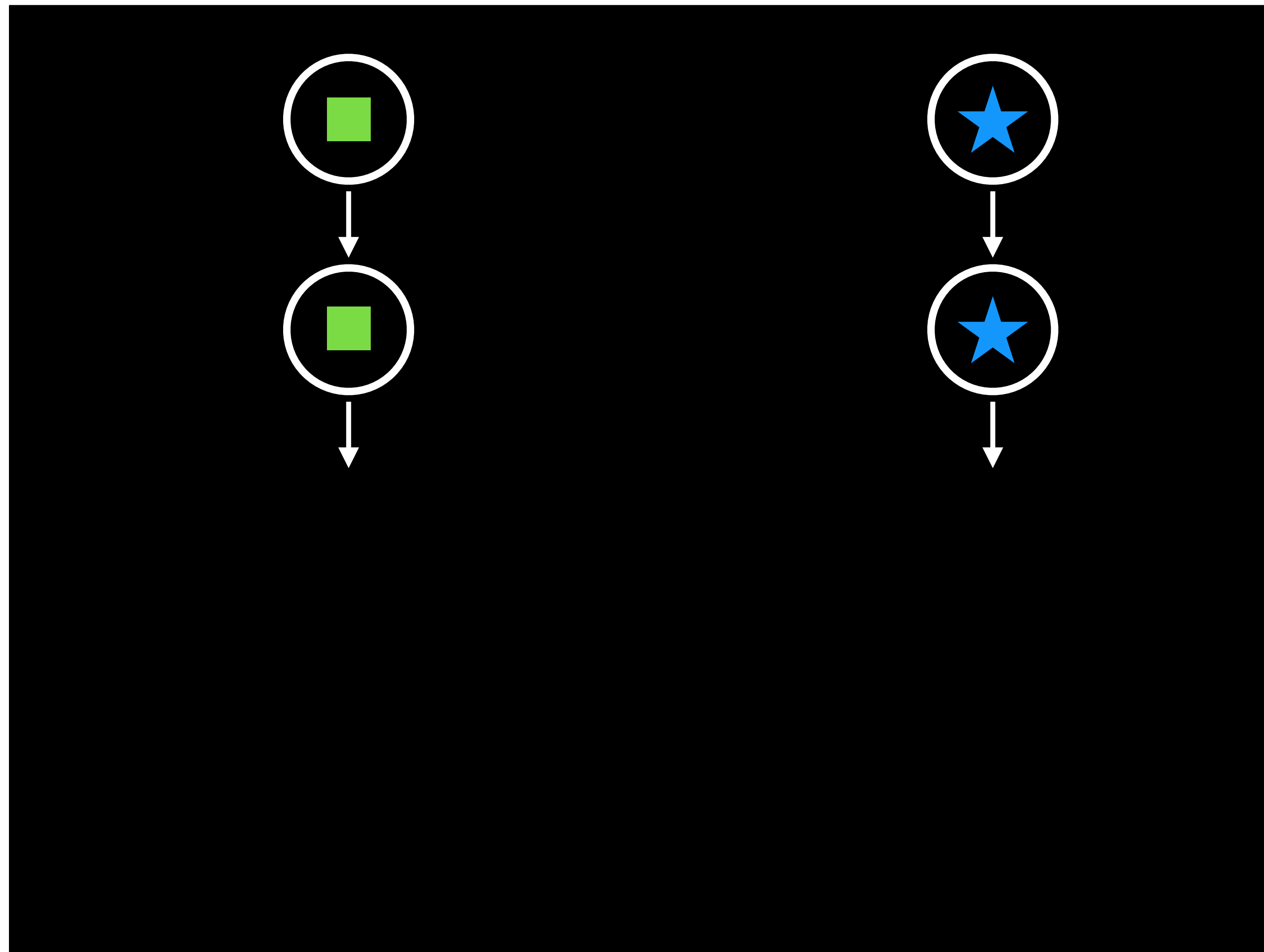
As exploration proceeds, …



… it may branch on a sensitive value. If we re-envision this as two **concrete** program traces that differ only on that sensitive value, …

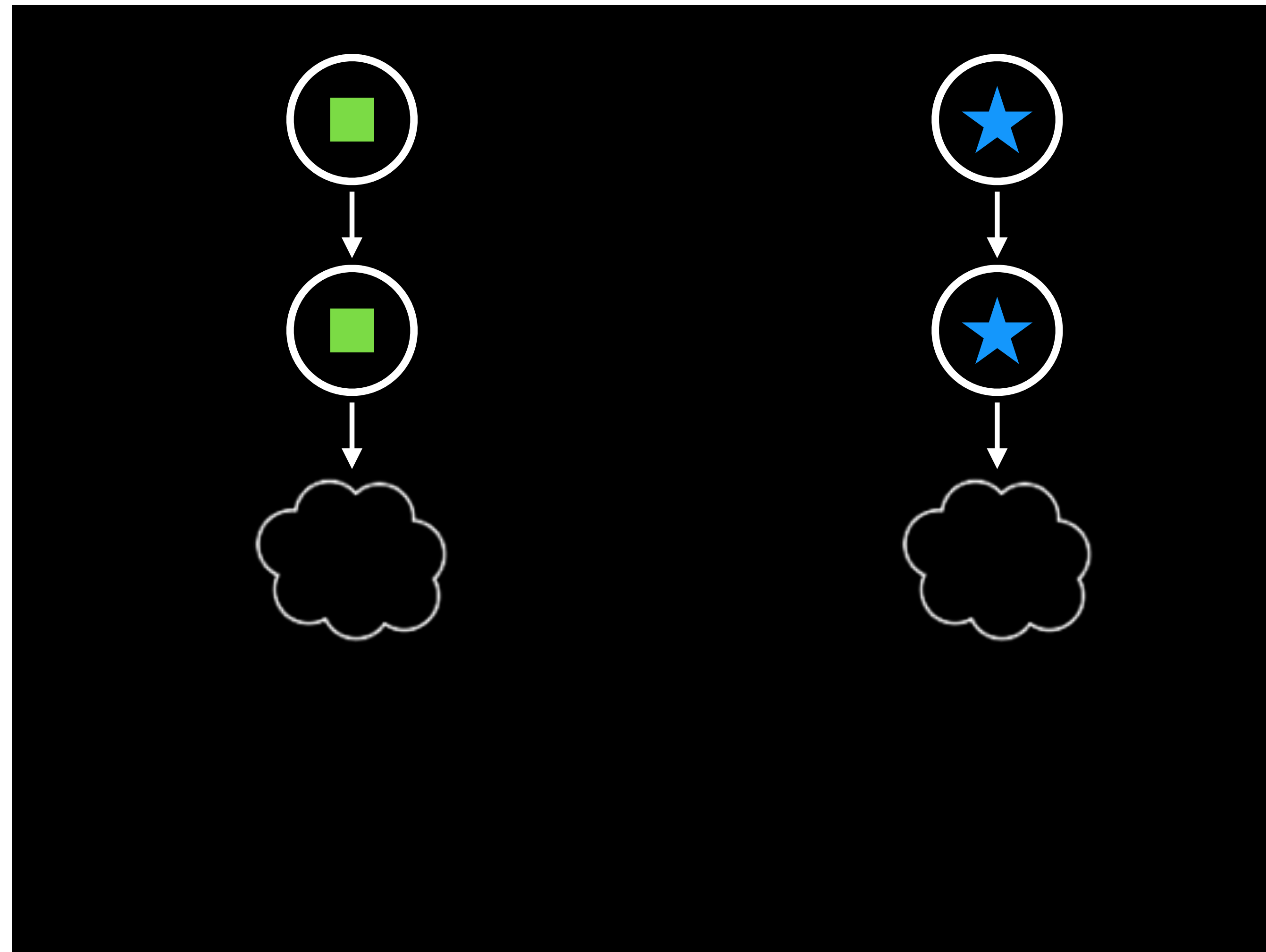


… we have exactly the type of situation described in the definition of non-interference.

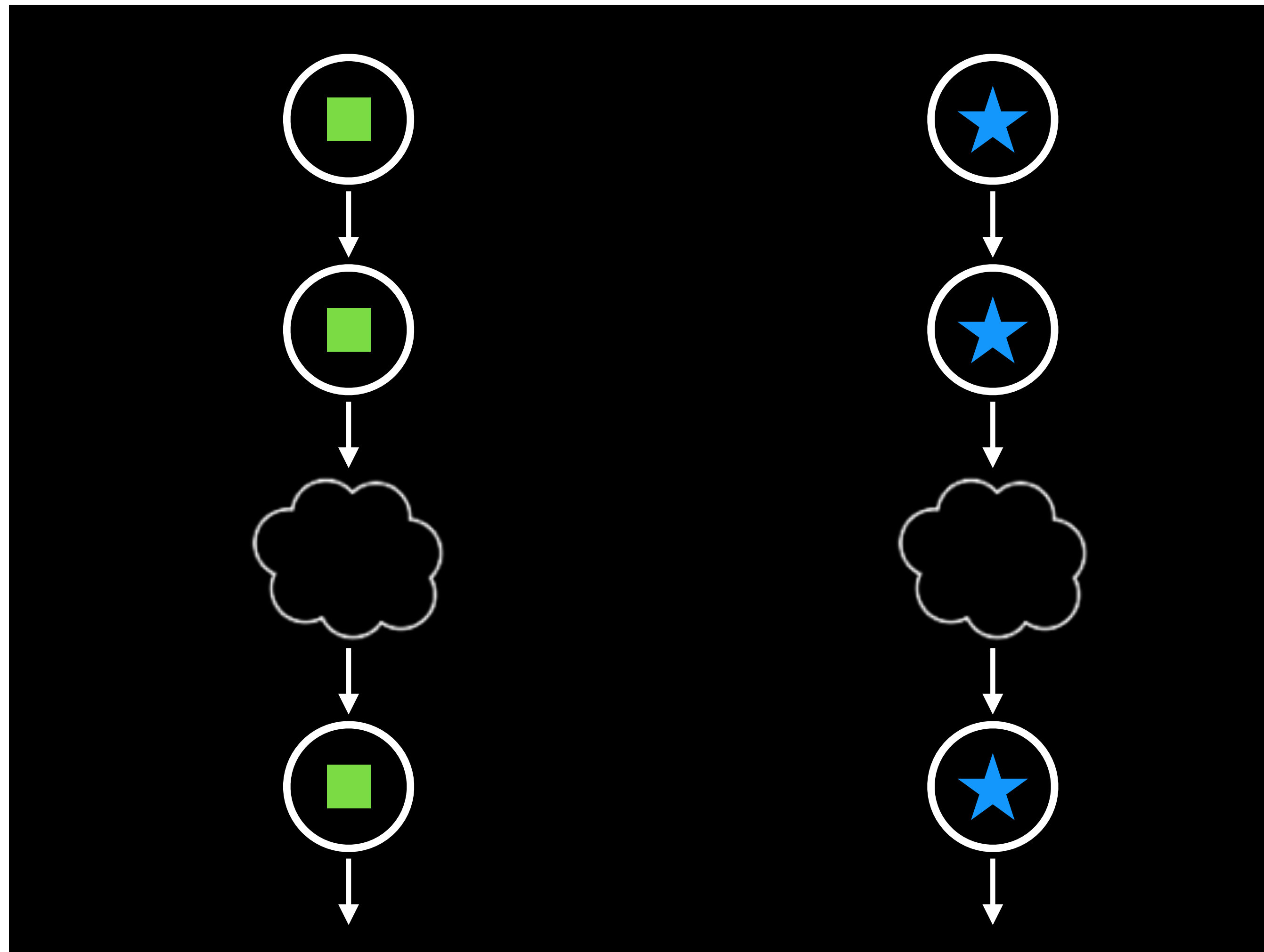


What we need to prove is that these two traces will differ only in detectable ways - either based on tainted values or in tainted contexts.

So we define a notion called similarity: states are similar if they are identical except for values at tainted addresses. We need to prove that similarity of one pair of (concrete) states implies the similarity of their successors.

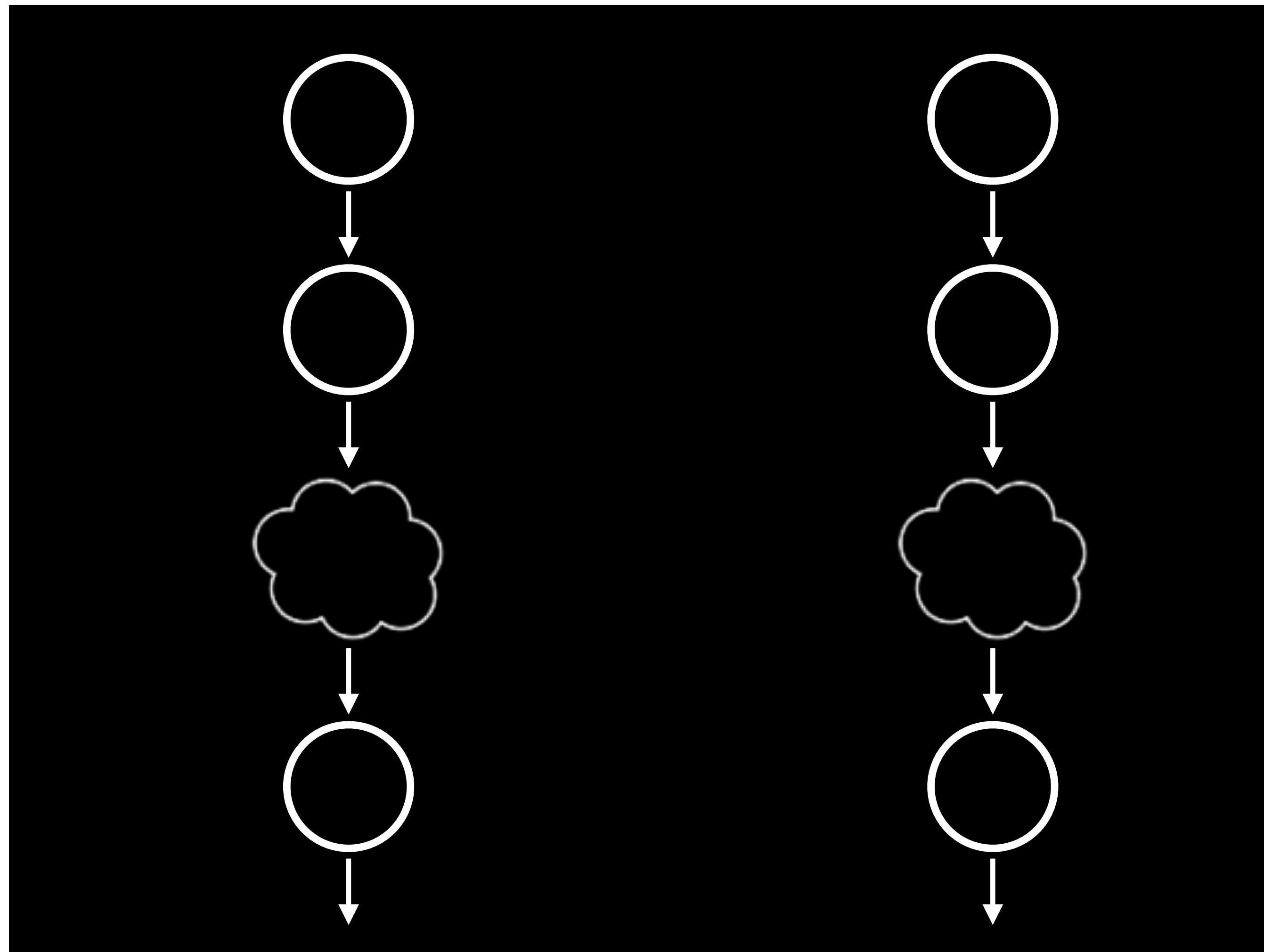


When they branch, this invariant is broken. So we weaken it slightly; it’s fine if they branch, as long as they don’t do anything observable during the branches …

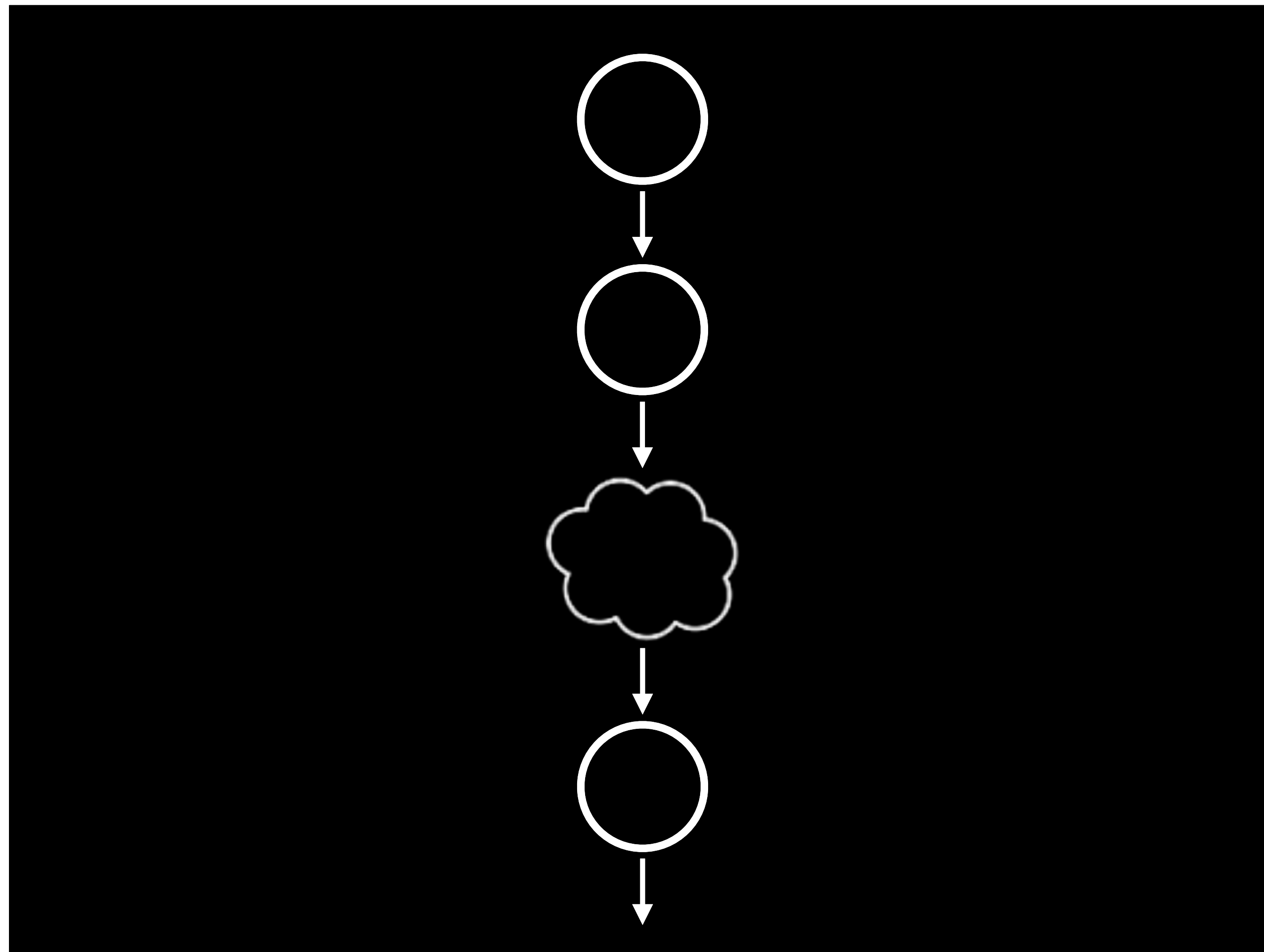


… and as long as values affected during the branches don’t affect observable behaviors downstream.

So our two traces, with respect to observable behaviors, …



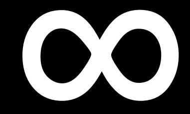
… are actually identical.



Therefore, two traces that start with two similar states will be similar throughout execution - except for states between branches on sensitive values and those branches' postdominators. Similar states' behaviors must be identical or identified as unsafe by the taint tracking mechanism.

Results?

TODO make a table showing the results of augmented state



Complexity?

Of 12 test applications, 12 time out

$\mathrm{C\hat{E}\hat{S}\hat{K}\hat{T}B}$

The complexity of small-step abstract interpretation is bounded by the size of the state space, …

ICI · IÊI · IŜI · IK̂I · IT̂I · IBI

… which is bounded by the number of possible states that can exist.

$$|\hat{T}| = |\widehat{Addr}| \cdot |\widehat{Taint}|$$

The number of possible taint stores is the number of possible addresses multiplied by the number of possible values at each address. Practically speaking, changes to the taint store will happen in lock step with the store and so this is unlikely to actually increase the number of states found.

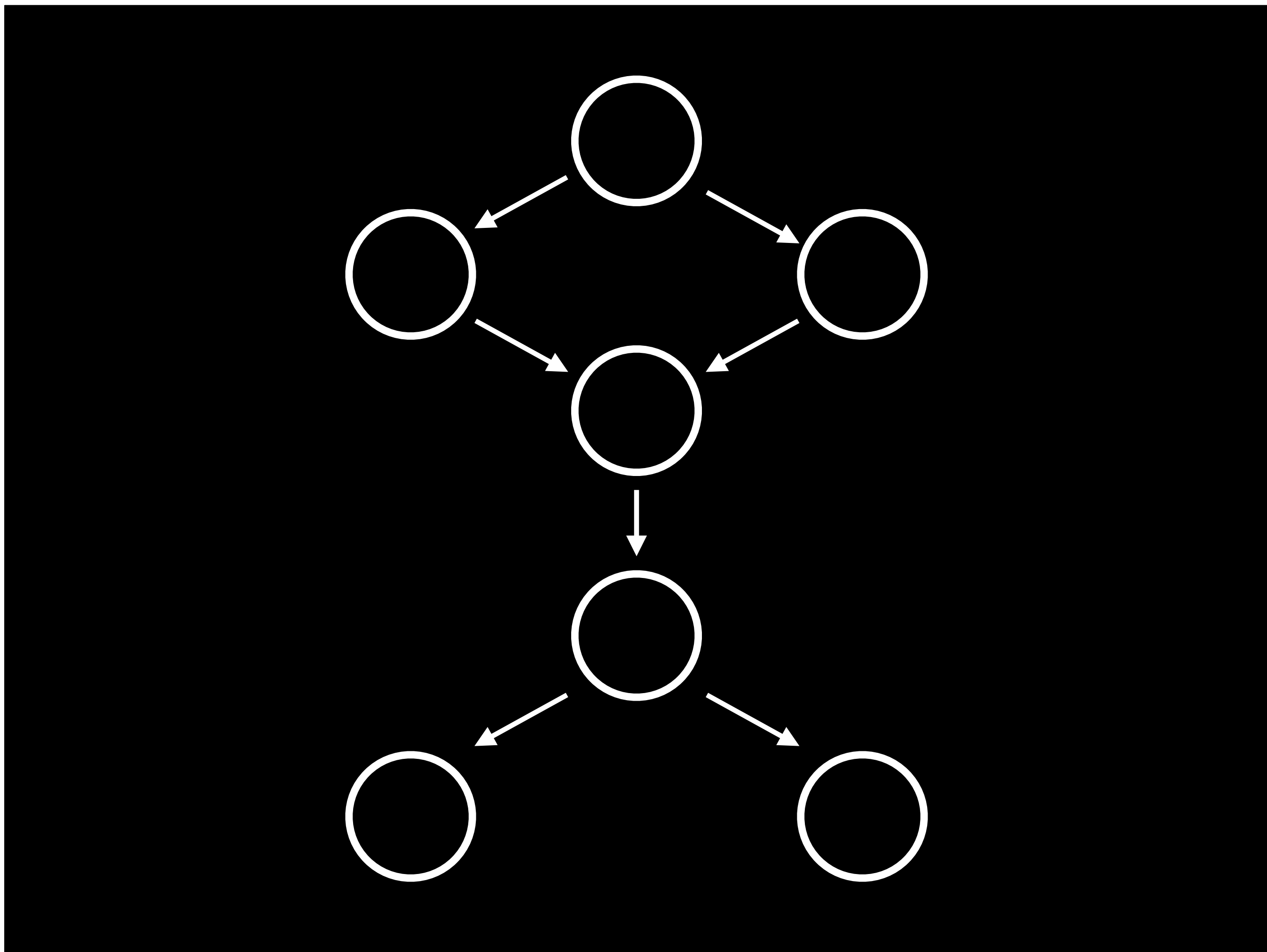
$$|B| = 2^{|C|}$$

On the other hand, the size of the context taint set is exponential in the size of the program.

Can we improve it?

CESKTB

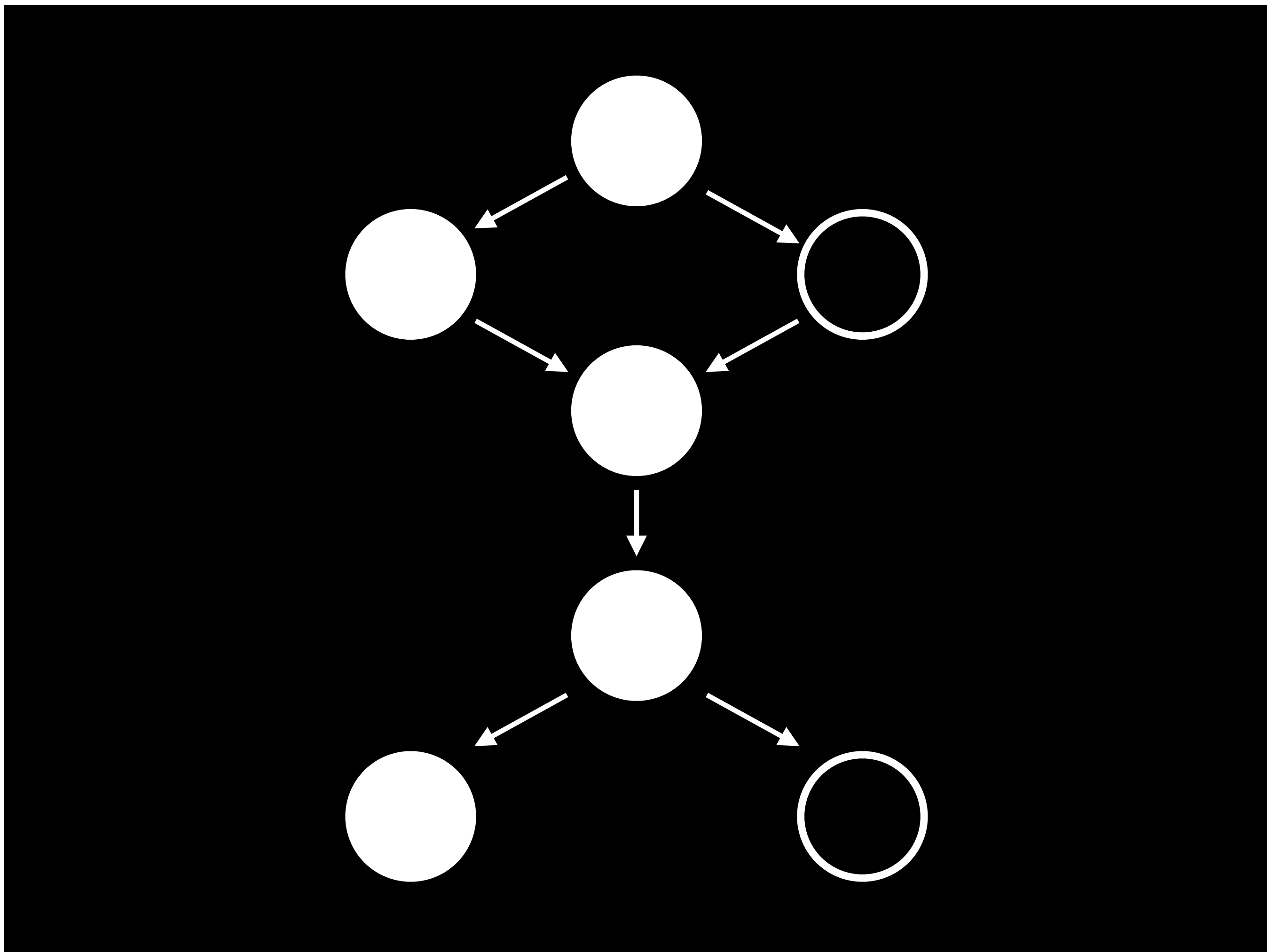
First, shrink the state space



So start with abstract interpretation and then do taint propagation over the finished abstract state graph.

CESKTB

This part does the TB.



So start with abstract interpretation and then do taint propagation over the finished abstract state graph.

Complexity?

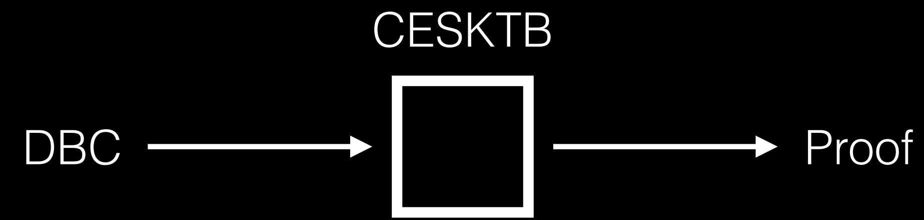
Unchanged

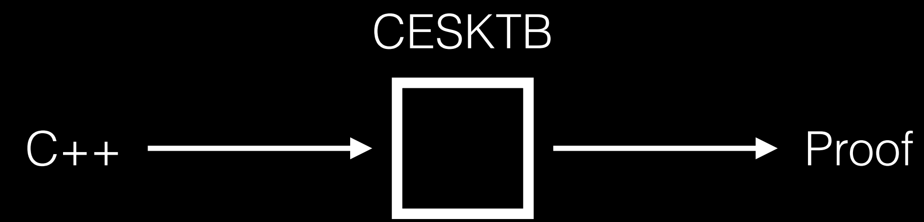
Same overall …

… but some parts are improved - it might help runtime performance

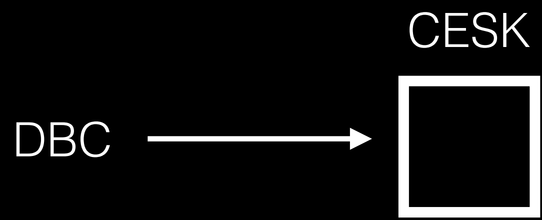
Also,

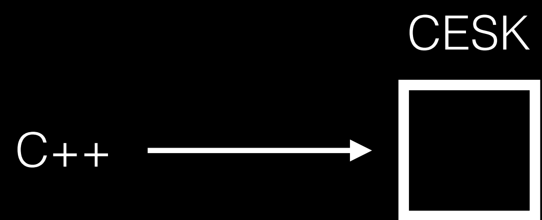
1) Projection to EPG has a much lower bound (CESK vs CESKTB)
2) EPG is already known when taint propagation happens
3) Intermediate results are available
4) This could be useful for other augmented state analyses; e.g., abstract counting



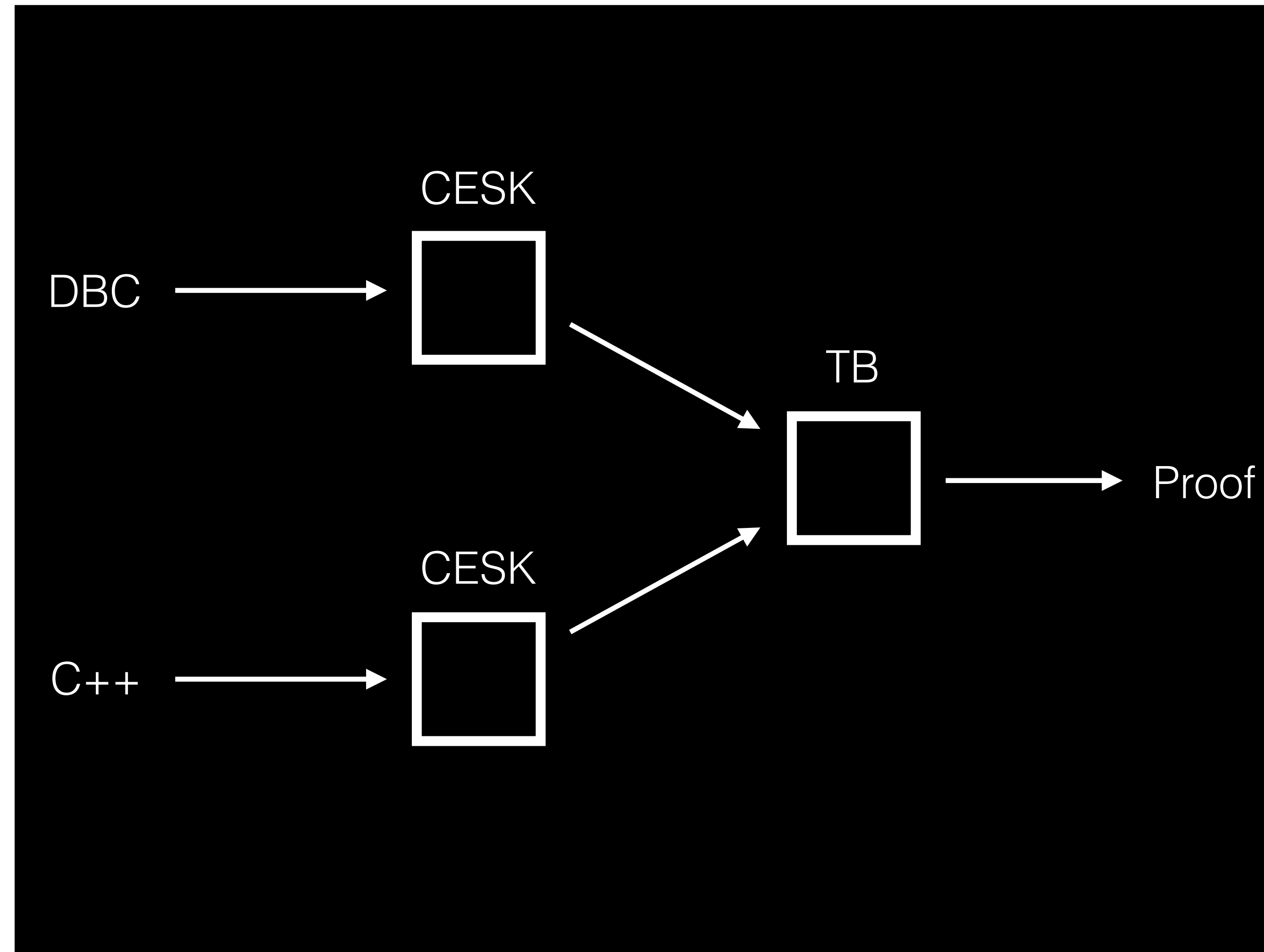


As future work, it may be possible to do a fully language-agnostic taint analysis. That is, instead of doing a separate analysis for each language, …





We can do abstract interpretation - any abstract interpretation …



… and then do the taint flow analysis later. This would allow us to merely create an abstract interpreter for our desired target language and then we could reuse the taint flow analysis and get a proof of non-interference without additional theoretical work.

Results?

5/12

Thank you

Thank you. Any questions?

	Total LOCs	LOCs seen	States	AI Time (m)	Total Time
BattleStat	3460				∞
chatterbocs	347618	1227	1612	217	410
ConferenceMaster	87512				∞
Filterize	2913	1451	3396	34	568
ICD9	88659	677	847	5	17
keymaster	48177				∞
Noiz2	17452	2238	2930	80	∞
PassCheck	17588				∞
pocketsecretary	290574				∞
rLurker	54921	680	918	3	18
splunge	520504				∞
Valet	362649	956	1263	37	164

ALL THE DATA (for the inevitable question).

Consider a joke about “be careful what you wish for”.

Please note that these results are preliminary.

	Flows	TPs	FPs	TNs	FNs
chatterbocs	23	1	1	20	0
Filterize	300	4	220	76	0
ICD9	25	0	0	25	0
rLurker	25	0	2	23	0
Valet	25	1	5	19	0