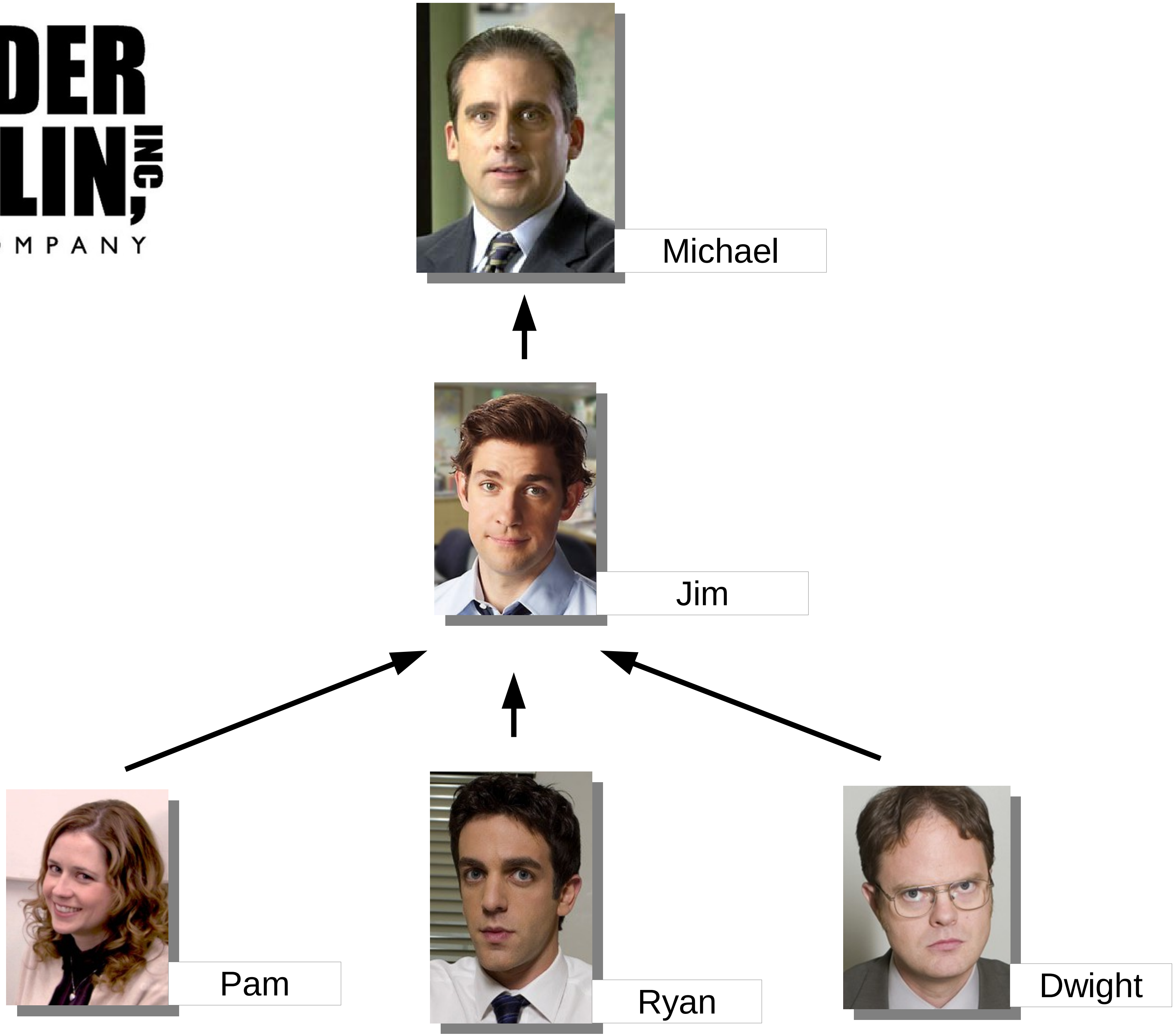
Very Static Enforcement
of
Dynamic Policies

COINS seminar 2014

Bart van Delft, Sebastian Hunt, David Sands

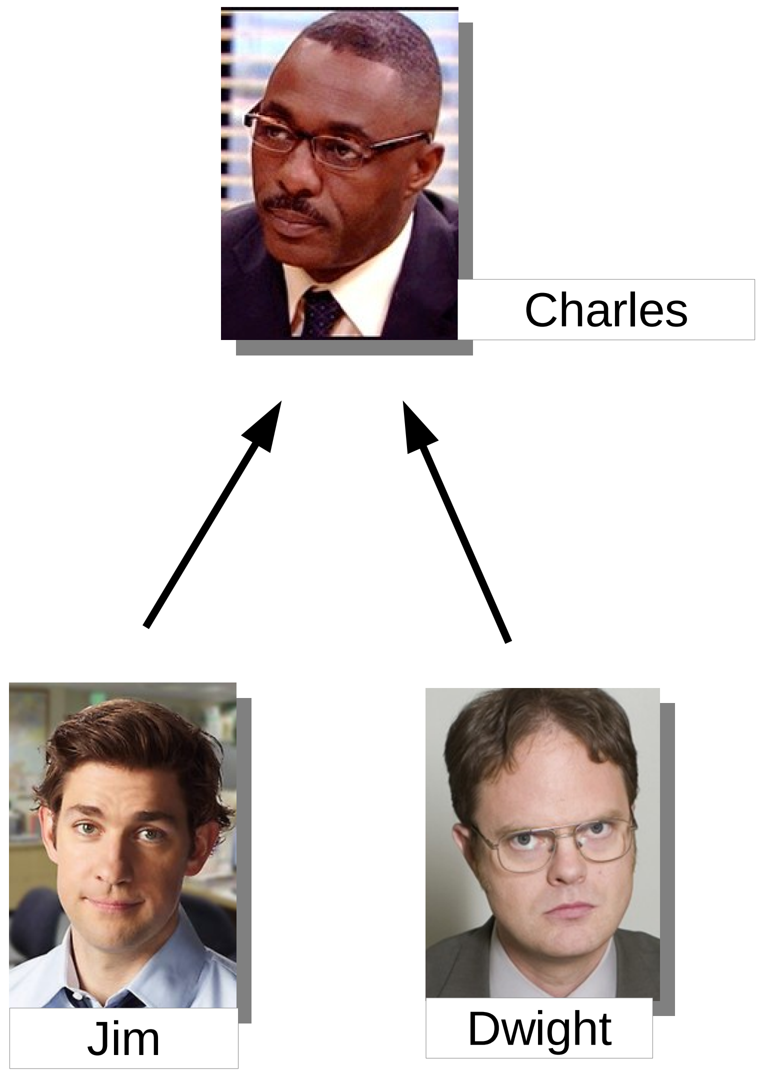

the office
DUNDER MIFFLIN
Staff Newsletter

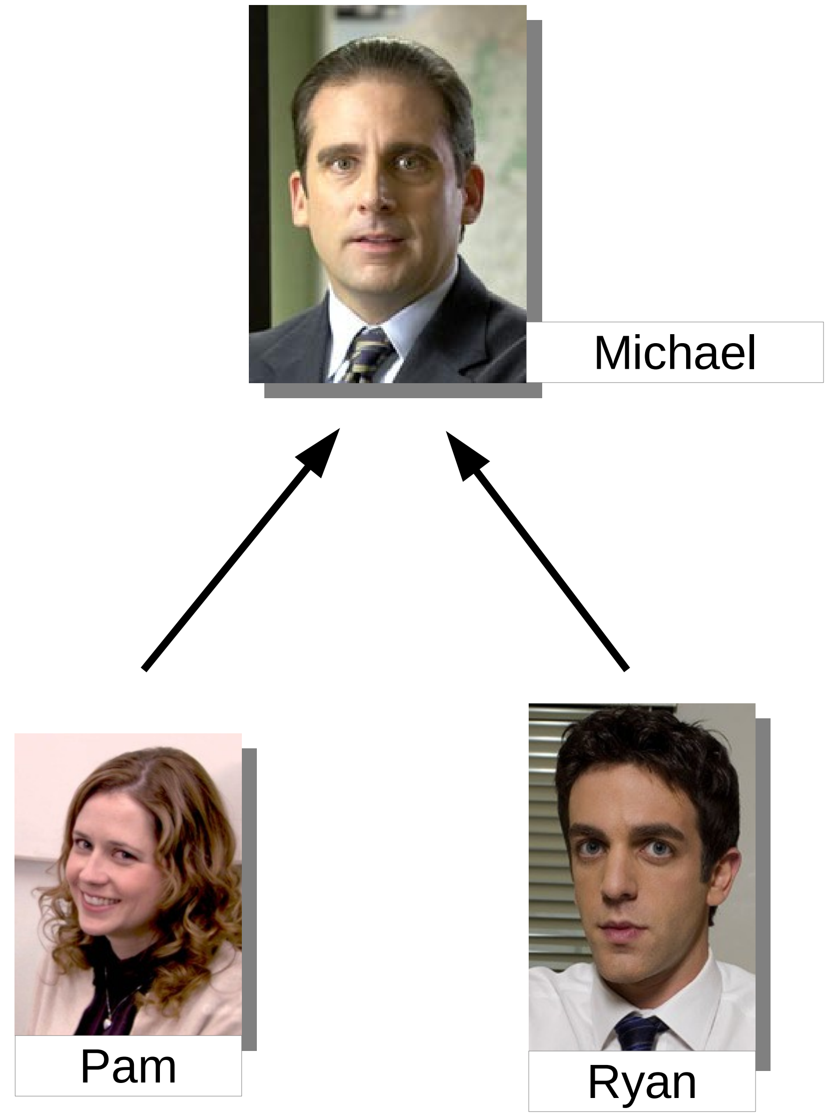
DUNDER MIFFLIN, INC
PAPER COMPANY



DUNDER MIFFLIN, INC.
PAPER COMPANY

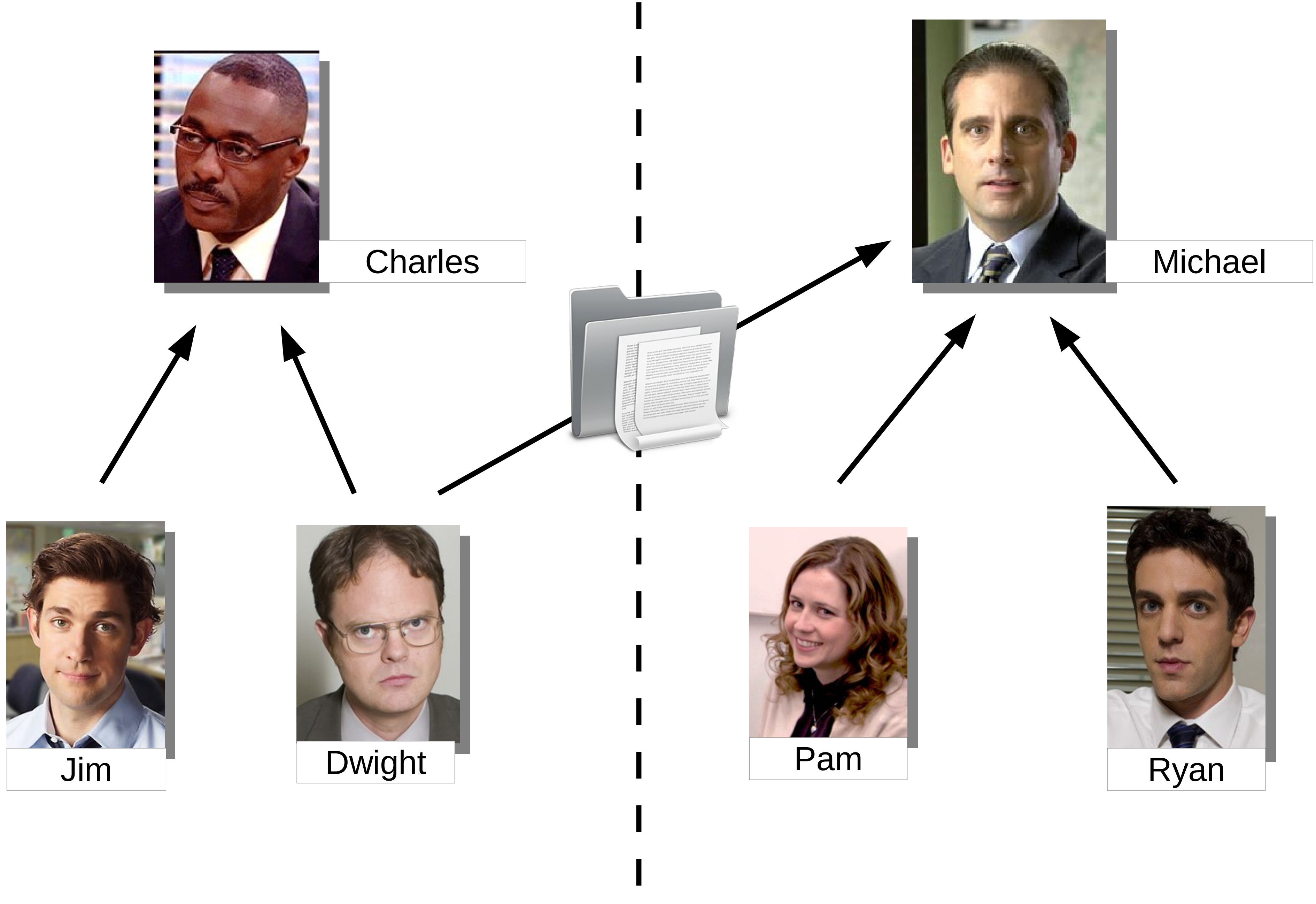
MICHAEL SCOTT
PAPER COMPANY INC.

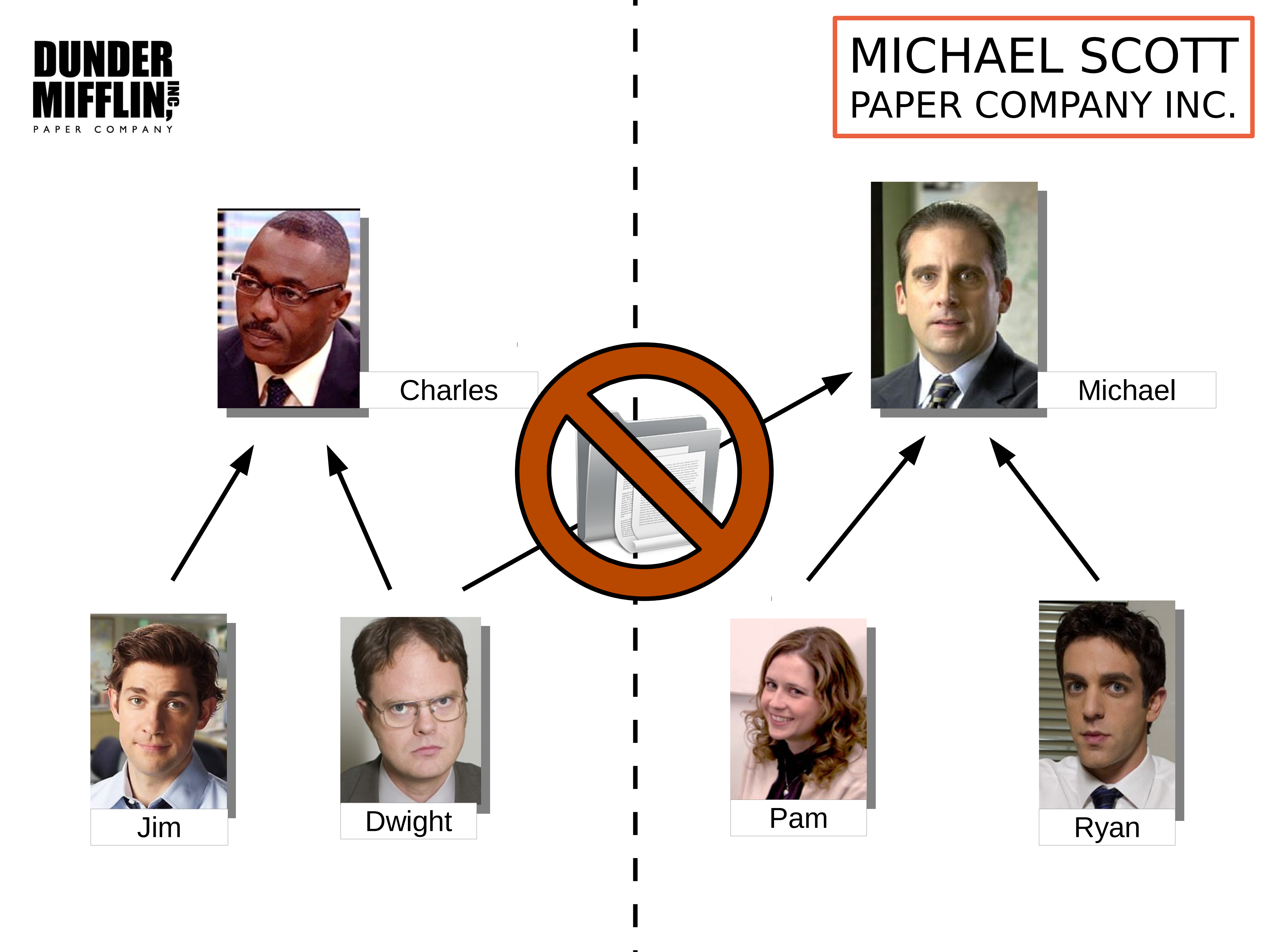




DUNDER MIFFLIN, INC.
PAPER COMPANY

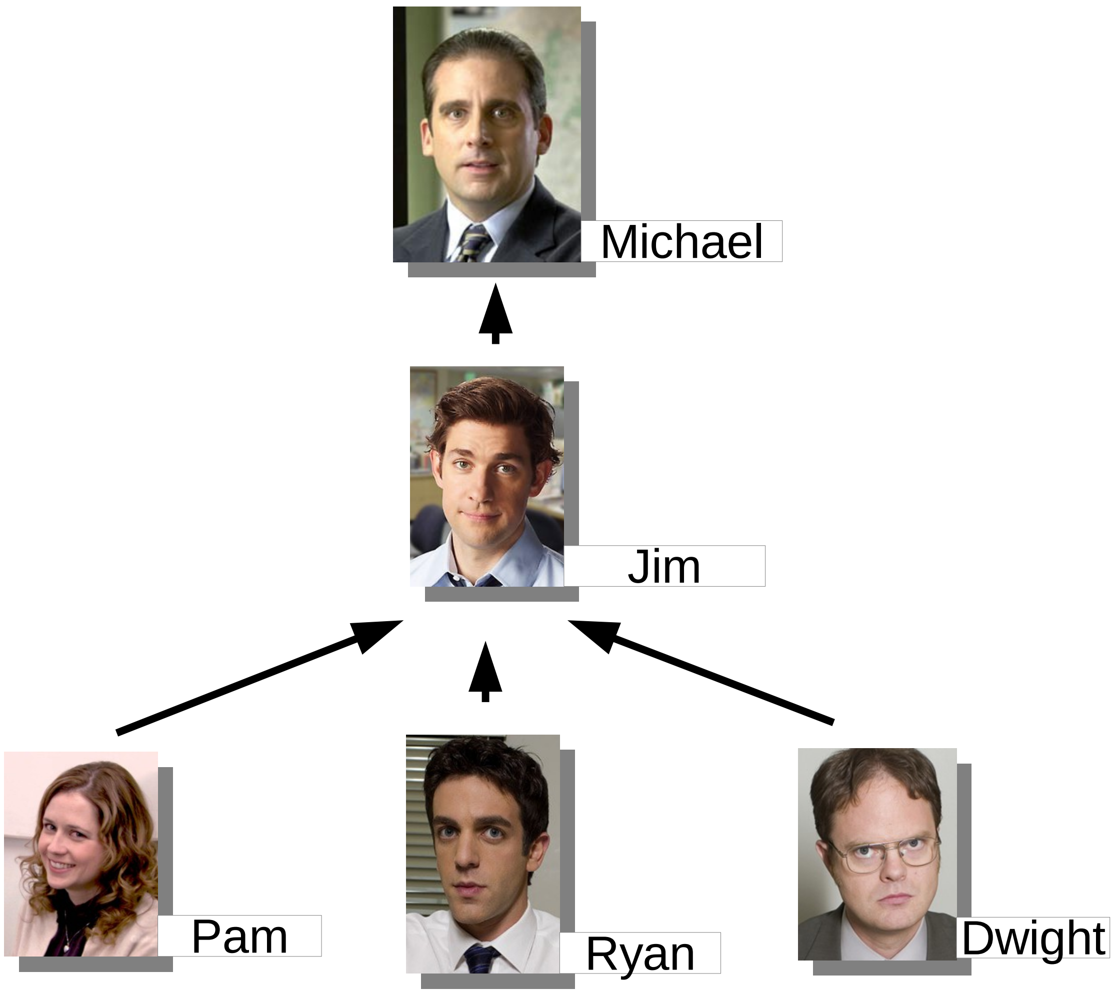
MICHAEL SCOTT
PAPER COMPANY INC.



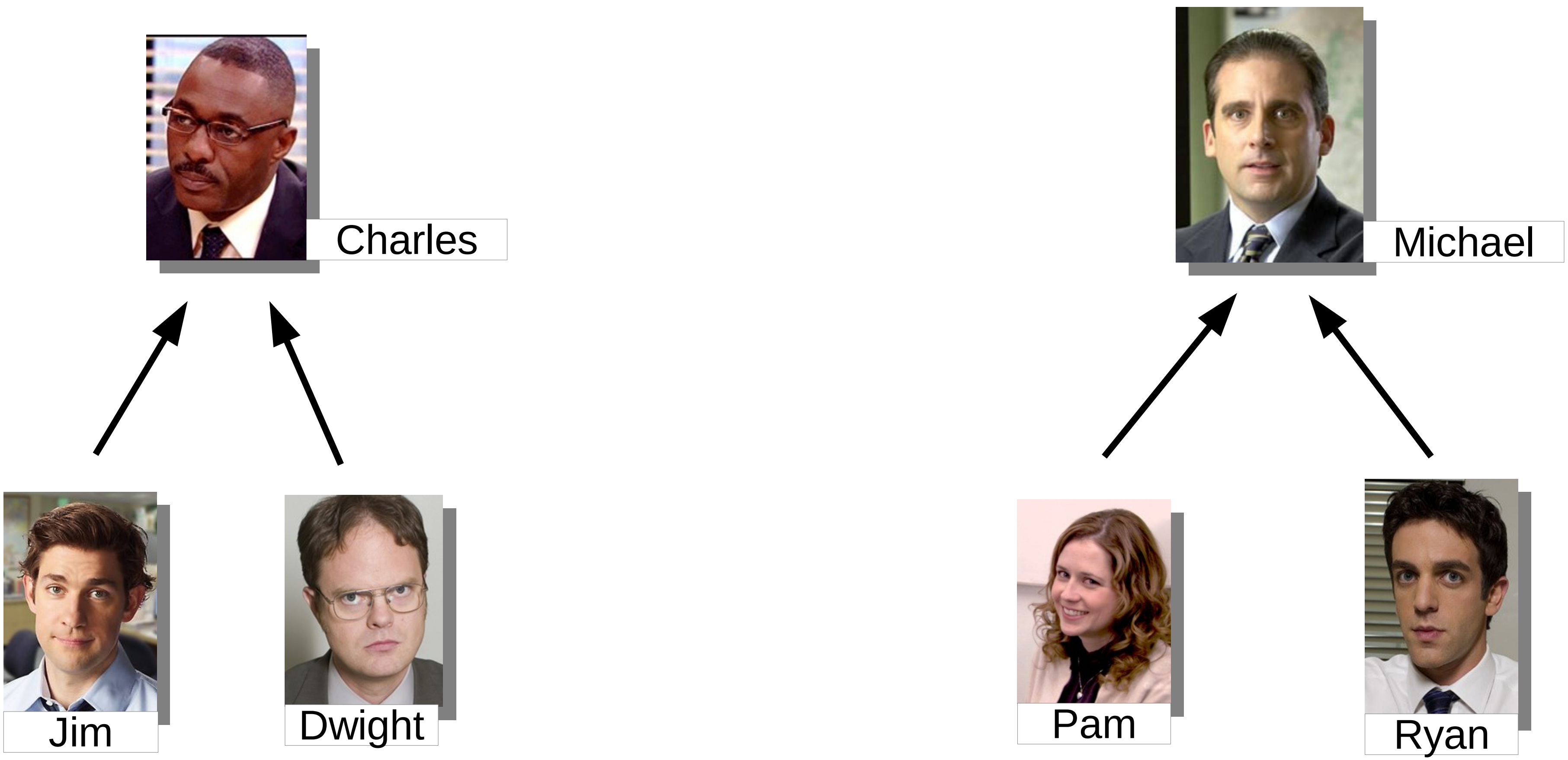

DUNDER MIFFLIN INC.
PAPER COMPANY
MICHAEL SCOTT
PAPER COMPANY INC.
Charles
Michael
Jim
Dwight
Pam
Ryan

security policies are
dynamic

```
prices := in from Dwight;
out prices to Michael;
```



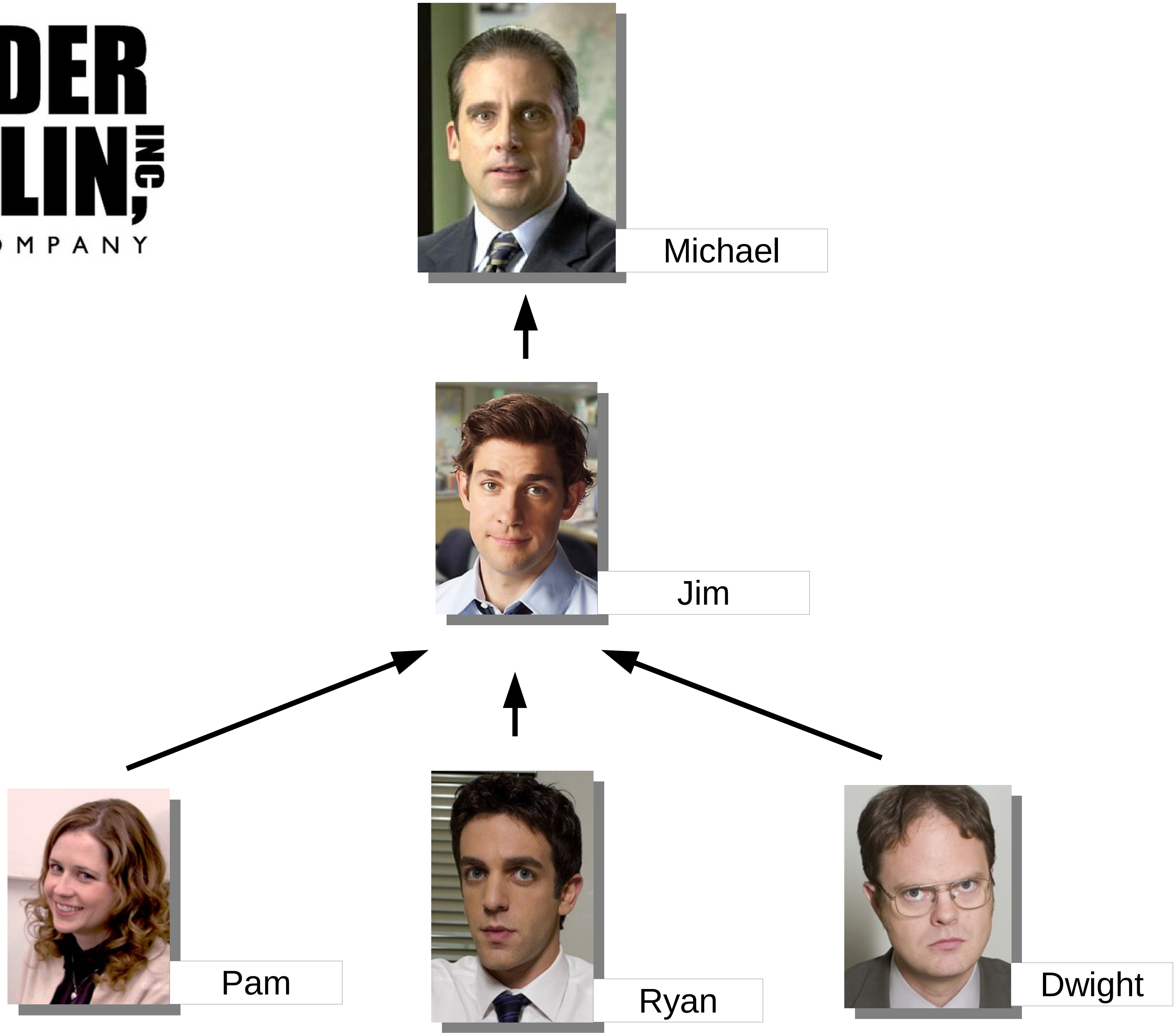
```
prices := in from Dwight;
out prices to Michael;
```



forget about

noninterference

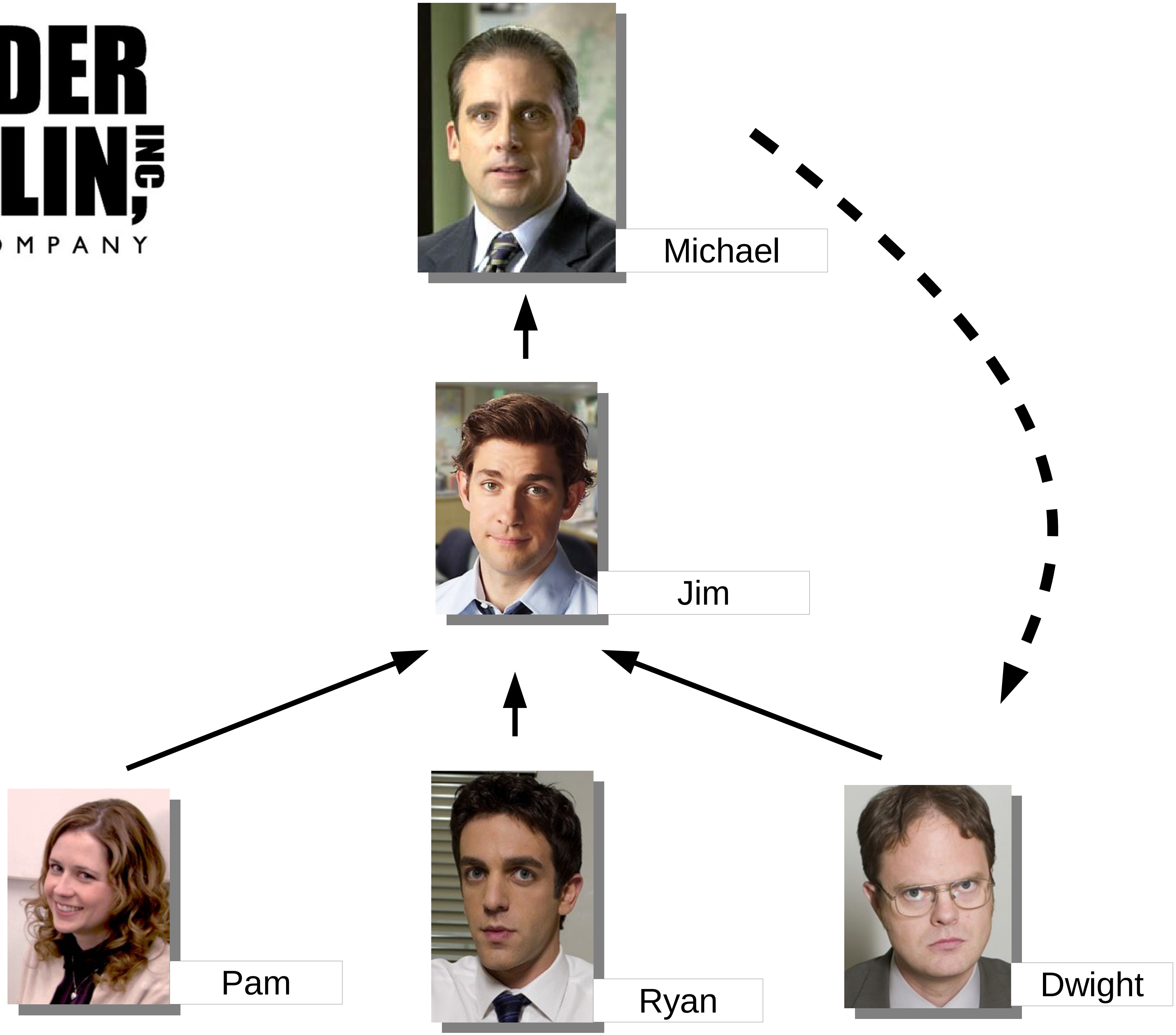
DUNDER MIFFLIN, INC
PAPER COMPANY



forget about

declassification

DUNDER MIFFLIN, INC
PAPER COMPANY



security policies are
dynamic

is existing literature **useless**?

add support for
dynamic policies to

On Flow-Sensitive Security Types

Sebastian Hunt
Department of Computing
School of Informatics, City University
London EC1V OHB, UK
seb@soi.city.ac.uk

David Sands
Department of Computer Science and Engineering,
Chalmers University of Technology
Göteborg, Sweden
dave@chalmers.se

with just a **small** modification

A Little Goes a Long Way

by Ashley Mills Monaghan

illustrations by Vivian Nguyen

principal
typing

one typing to **rule** them all

$\Gamma \vdash$

```
report := inputDwight
michaelData := report

report := inputJim
michaelData := report
```

$\Gamma \vdash$

```
report := inputDwight
michaelData := report

report := inputJim
michaelData := report
```

dependencies
are all you need

$\Gamma \vdash$

```
report := inputDwight
michaelData := report

report := inputJim
michaelData := report
```

$$\Gamma(\texttt{inputDwight}) = \{\texttt{inputDwight}\}$$

$$\Gamma(\texttt{inputJim}) = \{\texttt{inputJim}\}$$

$$\Gamma(\texttt{report}) = \Gamma(\texttt{michaelData}) = \{\texttt{inputJim}\}$$

`inputDwight` → secret

`inputJim` → public

`report` → public

```
report := inputDwight
michaelData := report


report := inputJim
michaelData := report
```

$$\Gamma(\texttt{inputDwight}) = \{\texttt{inputDwight}\}$$

$$\Gamma(\texttt{inputJim}) = \{\texttt{inputJim}\}$$

$$\Gamma(\texttt{report}) = \Gamma(\texttt{michaelData}) = \{\texttt{inputJim}\}$$

`inputDwight` $\longrightarrow$ secret

`inputJim` $\longrightarrow$ secret

`report` $\longrightarrow$ public

```
report := inputDwight
michaelData := report

report := inputJim
michaelData := report
```

$$\Gamma(\texttt{inputDwight}) = \{\texttt{inputDwight}\}$$

$$\Gamma(\texttt{inputJim}) = \{\texttt{inputJim}\}$$

$$\Gamma(\texttt{report}) = \Gamma(\texttt{michaelData}) = \{\texttt{inputJim}\}$$

one typing to rule them all

security policies are

dynamic

policy 1 →

policy 2 →

```
report := inputDwight
michaelData := report


report := inputJim
michaelData := report
```

still the same **dependencies**

only **last** policy relevant?

policy 1 →

policy 2 →

```
report := inputDwight
out report to Michael

report := inputJim
out report to Michael
```

modification 1:

add **outputs**

policy 1 →

policy 2 →

```
report := inputDwight
out report to Michael


report := inputJim
out report to Michael
```

dependencies

differ **per** output

policy 1 →

policy 2 →

```
report := inputDwight
out report to Michael @ p

report := inputJim
out report to Michael @ q
```

modification 2:

maintain **dependencies** per **output**

policy 1 →

policy 2 →

```
report := inputDwight
out report to Michael @ p

report := inputJim
out report to Michael @ q
```

$$\Gamma(\texttt{Michael @ p}) = \{\texttt{inputDwight}\}$$

$$\Gamma(\texttt{Michael @ q}) = \{\texttt{inputJim}\}$$

(policies irrelevant for typing)

```
report := inputDwight
out report to Michael @ p

report := inputJim
out report to Michael @ q
```

$$\Gamma(\texttt{Michael @ p}) = \{\texttt{inputDwight}\}$$

$$\Gamma(\texttt{Michael @ q}) = \{\texttt{inputJim}\}$$

A Little Goes a Long Way

by Ashley Mills Monaghan

illustrations by Vivian Nguyen

only adding **one** typing rule

TS-OUTPUT

$$\vdash \{\texttt{out}\ e\ \texttt{on}\ a\ @\ p\}\Gamma_{id}[a_p \mapsto fv(e) \cup \{\texttt{pc}, a, a_p\}; a \mapsto \{\texttt{pc}, a\}]$$

dependencies

are ***still*** all you need

how does this enforce dynamic policies?

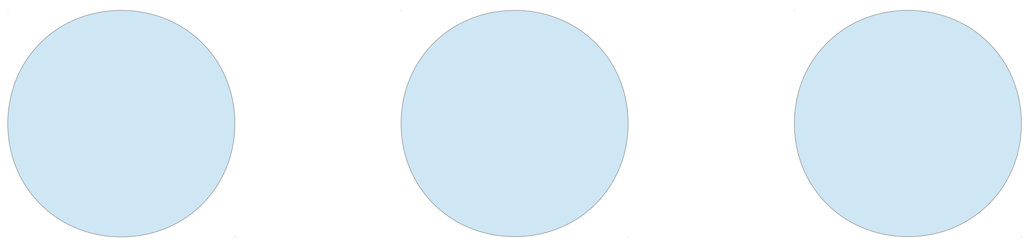
$$\Gamma\left(\texttt{Michael @ p}\right) = \left\{\texttt{inputDwight}\right\}$$

$$\Gamma\left(\texttt{Michael @ q}\right) = \left\{\texttt{inputJim}\right\}$$

let's first **define**
dynamic policies

policy changes **synchronously** with program execution

execution points determine current policy ...



... approximated by **program points**

assume **approximation** of policy per program point

```
report := inputDwight
out report to Michael @ p

report := inputJim
out report to Michael @ q
```

assume **approximation**

of policy per ~~program~~ point

output

approx. p →

approx. q →

```
report := inputDwight
out report to Michael @ p

report := inputJim
out report to Michael @ q
```

enforcement:

check if dependencies conform with approximations

take-home messages
of this talk

security policies are

dynamic

extending existing work
is possible

dependencies

are all you need

Paper, code and more pictures on

slio.bitbucket.org

Dynamic Enforcement of Dynamic Policies

Pablo Buiras and Bart van Delft

This poster is supported by COINS funding

Contribution

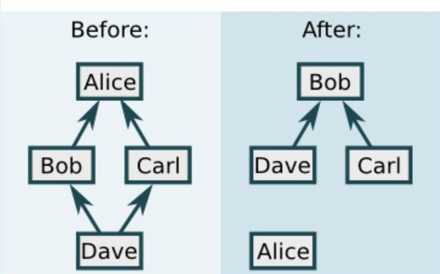
Information flow research aims to detect and prevent information flows disallowed in a system. Although security policies are inherently dynamic, most approaches only enforce static policies.

This poster presents an extension to support dynamic policies in LIO, a policy enforcement library for Haskell.

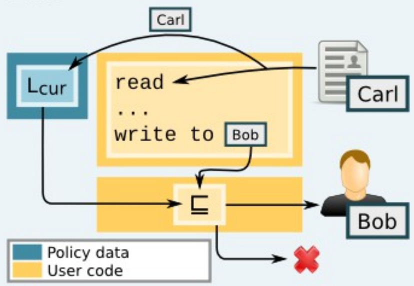
Dynamic Policies

Most enforcement mechanisms only enforce **static** security policies, such as the example company policy in the 'Before' picture. Here Carl's information can flow to his boss Alice, but not to Bob or Dave.

In practice, however, policies are much more **dynamic** and change *while* the system is running. For example, Alice might get fired, leading to Bob and Dave being promoted as shown in the 'After' picture. Now Carl's information can flow to Bob, but no longer to Alice.



LIO



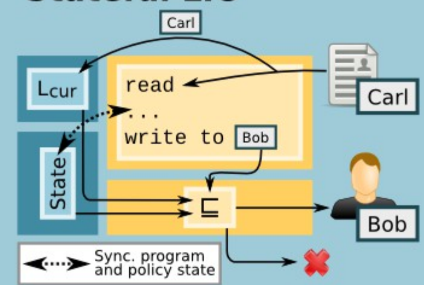
The Haskell library LIO dynamically enforces information flow control[1]. That is, it checks for violations of a (static) security policy while the program is running.

All input/output points are labelled with a security level (hence the name Labelled IO). The LIO library replaces default I/O operations and maintains in the **current label** an upper bound on the information currently in scope. As static policies are typically defined as *lattices*, such an upper bound always exists. Before a side-effect happens, such as writing to a file, LIO verifies that the information in scope is allowed to flow to an output with that label.

LIO is parametrised: user code is able to specify the set of security labels and the static ordering (⊑) between them.

CHALMERS
UNIVERSITY OF TECHNOLOGY

Stateful LIO

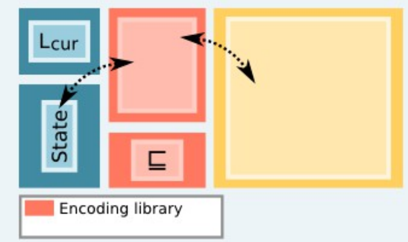


We propose that the LIO library additionally maintains a **policy state**. User code may specify what kind of information is stored in the state and use operations provided by LIO to read or modify the state.

The state is provided as an additional argument to ⊑ and can thus influence the ordering between labels. Therefore, there may be no upper bound to store in the current label, so we represent it as a set of labels.

An additional check is introduced to verify new information flows arising from state change.

Encodings



Rather than defining ⊑ and policy state by itself, user code can (be required to) use a library encoding a particular **policy language**. We have successfully implemented several policy languages, including DLM[2] and Paralocks[3].

Future Work

We have proven our extension secure for sequential LIO. The next challenge is to support **concurrent** LIO as well.

References

[1] *Flexible Dynamic Information Flow Control in Haskell*, Stefan et al., Haskell '11, p. 95-106, 2011.
[2] *Protecting privacy using the decentralized label model*, Myers, TOSEM, p. 410-442, 2000.
[3] *Paralocks - role-based information flow control and beyond*, Broberg and Sands, POPL '10, p. 431-444, 2010.

Example

We encode the state as the set of allowed flows:

$$\mathtt{lbl}_1 \sqsubseteq_S \mathtt{lbl}_2 \quad = \quad (\mathtt{lbl}_1, \mathtt{lbl}_2) \in S$$

Starting with the policy state as in the 'Before' figure, `exampleProgram` is secure. unless we remove the call to `fireAlice`, then LIO prevents the write to Bob.

```
fireAlice = do
  s <- getState
  let s'  = s + {(carl,bob)}
    - {(carl,alice),(bob,alice)}
  in setState s'

exampleProgram = do
  fireAlice
  data <- read dataCarl
  writeTo Bob data
```

thank you!

www.cse.chalmers.se/~vandeba