The Tome of Secure Compilation

Fully-Abstract Compilation to Protected Modules Architectures

Marco Patrignani

iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium
first.last@cs.kuleuven.be

Supervisors: Dave Clarke & Frank Piessens

'ssup?

The Tome of Secure Compilation

Fully-Abstract Compilation to Protected Modules Architectures

'ssup?

The Tome of Secure Compilation

Fully-Abstract Compilation to Protected Modules Architectures

'ssup?

The Tome of Secure Compilation

Fully-Abstract Compilation to Protected Modules Architectures

'ssup?

The Tome of Secure Compilation

Fully-Abstract Compilation to Protected Modules Architectures

'ssup?

The Tome of Secure Compilation

Fully-Abstract Compilation to Protected Modules Architectures

1

'ssup?

The Tome of Secure Compilation 2

Fully-Abstract Compilation to Protected Modules Architectures 1

'ssup?

The Tome of Secure Compilation

2

...

1

Fully-Abstract Compilation to Protected Modules Architectures

Protected Modules Architectures

What is PMA?

Protected Modules Architectures

A security architecture

What is PMA?

Protected Modules Architectures

A security architecture

Isolation @ assembly

What is PMA?

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- memory space

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- memory space
- protected module (isolated)

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- memory space
- protected module (isolated)
- split in code and data

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

r/w

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

r/x

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

r/w/x

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

r/w/x

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

r/w/x

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

r/w/x

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

A+I: Untyped Assembly + PMA

```
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
■ 0x0b52    write r0 at 0x0b55
  0x0b53    write r0 at 0x0001
  0x0b54    call func. at 0x0002
  0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted
- unprotected code is restricted
- entry points for communication (■)

A+I: Untyped Assembly + PMA

```
                x
0x0001    call func. at 0xb53
0x0002    write r0 at 0x0b55
⋮
■ 0x0b52    write r0 at 0x0b55
  0x0b53    write r0 at 0x0001
  0x0b54    call func. at 0x0002
  ──────────────────────────────
  0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- memory space
- protected module (isolated)
- split in code and data
- protected code is unrestricted
- unprotected code is restricted
- entry points for communication (■)

Reasoning about A+I

```
0x0001    call func. at 0xb52
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- interest in the behaviour of the module

Reasoning about A+I

```
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
```

- interest in the behaviour of the module: this one

Reasoning about A+I

```
0x0001    call func. at 0xb52
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- interest in the behaviour of the module: this one
- need to consider the *rest*

Reasoning about A+I

```
0x0001    call func. at 0xb52
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return t
0xab02    ...
```

Problem
There is a lot of *the rest*

- interest in the behaviour of the module: this one
- need to consider the *rest*

Reasoning about A+I

```
0x0001    call func. at 0xb52
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x
0xab01    return t
0xab02    ...
```

Solution
Use *Trace semantics*

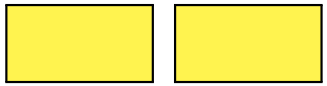
Problem
There is a lot of *the rest*

- interest in the behaviour of the module: this one
- need to consider the *rest*

What is trace semantics?

- A trace is a sequence of actions: 

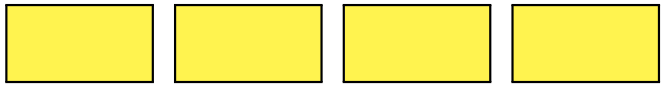
What is trace semantics?

- A trace is a sequence of actions: 

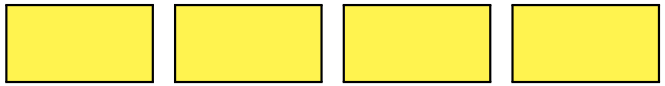
What is trace semantics?

- A trace is a sequence of actions: 

What is trace semantics?

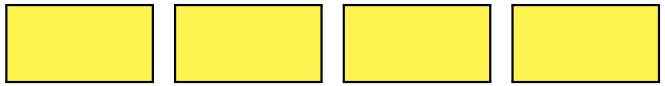
- A trace is a sequence of actions: 

What is trace semantics?

- A trace is a sequence of actions: 
- Actions describe an observable behaviour abstractly

 $= \begin{cases} \text{call a function} \\ \text{write something somewhere} \\ \ldots \end{cases}$

What is trace semantics?

- A trace is a sequence of actions: 
- Actions describe an observable behaviour abstractly

 $= \begin{cases} \text{call a function} \\ \text{write something somewhere} \\ \ldots \end{cases}$

- The behaviour of some code is a set of traces

Trace semantics for A+I

```
0x0001    call func. at 0xb52
0x0002    write r0 at 0x0b55
⋮
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
⋮
0xab00    jump to 0x0001
0xab01    return to 0x0b53
0xab02    ...
```

- disregard the rest

Trace semantics for A+I

```
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
```

- disregard the rest

Trace semantics for A+I

```
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
```

- disregard the rest
- abstract its behaviour *from the module perspective*:

Trace semantics for A+I

- disregard the rest
- abstract its behaviour *from the module perspective*:

  1. jump to an entry ■ (call/return)

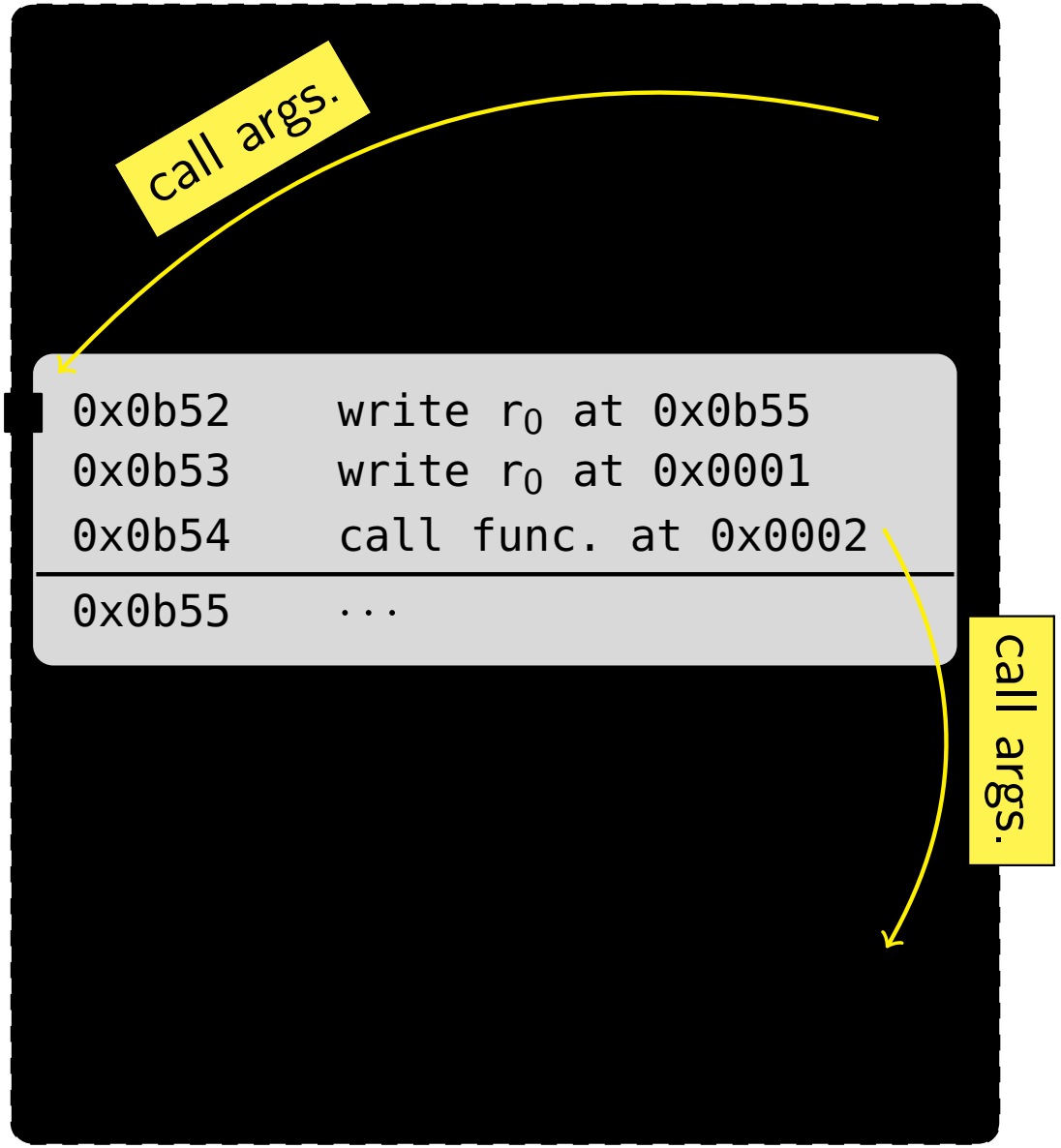
Trace semantics for A+I

- disregard the rest
- abstract its behaviour *from the module perspective*:

  1. jump to an entry ■ (call/return)

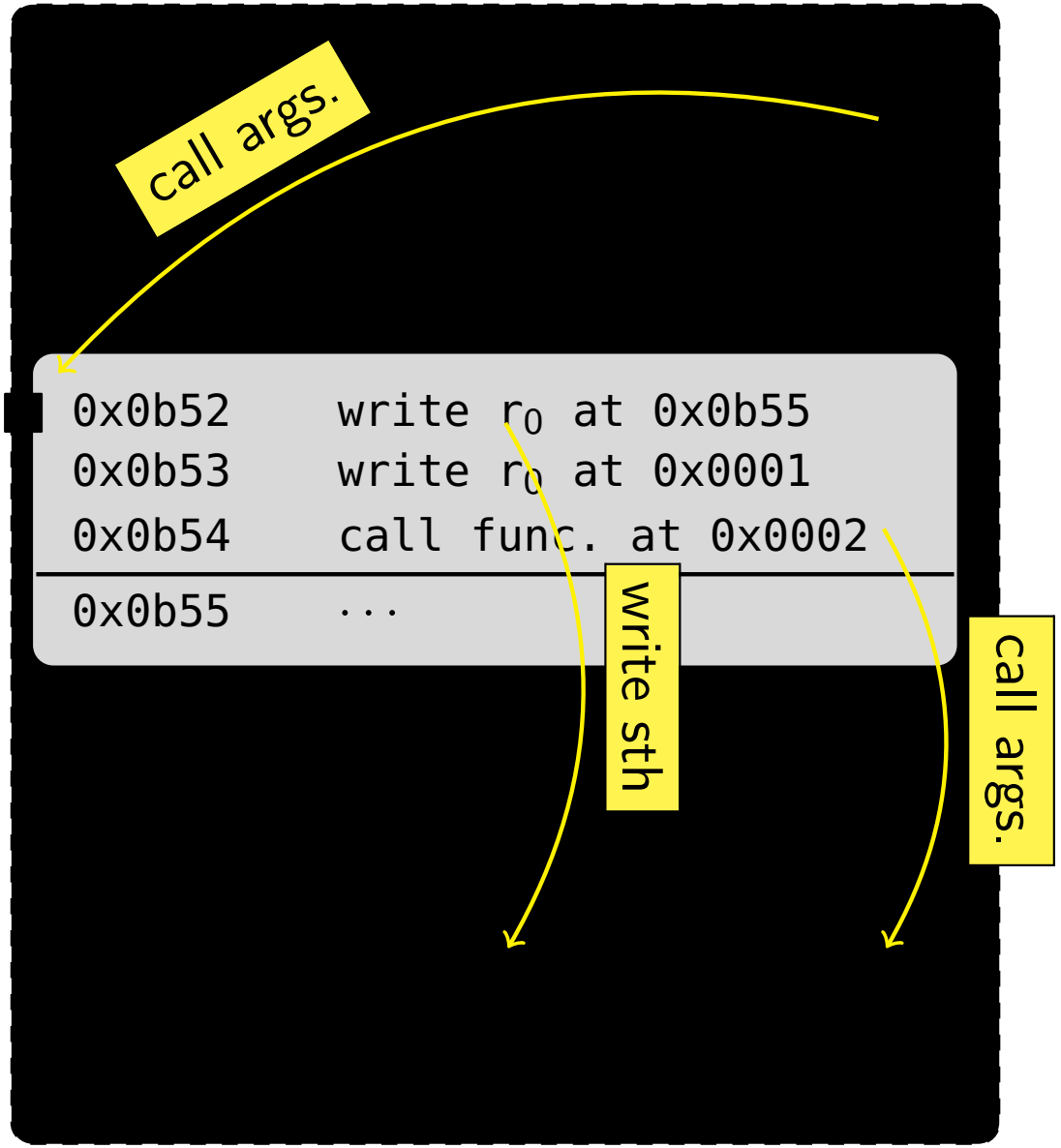
- abstract the module behaviour *from the rest perspective*:

Trace semantics for A+I



- disregard the rest
- abstract its behaviour *from the module perspective*:
  1. jump to an entry ■ (call/return)
- abstract the module behaviour *from the rest perspective*:
  1. call/return outside

Trace semantics for A+I



- disregard the rest
- abstract its behaviour *from the module perspective*:
  1. jump to an entry ■ (call/return)
- abstract the module behaviour *from the rest perspective*:
  1. call/return outside
  2. read/write outside

Trace semantics for this example

```
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
```

Trace of this example:

call 0x0b52 (args) · write 0x0001 (arg) · call 0x0002 (args)

Trace semantics for this example

```
0x0b52    write r0 at 0x0b55
0x0b53    write r0 at 0x0001
0x0b54    call func. at 0x0002
0x0b55    ...
```

Trace of this example:

call 0x0b52 (args)**?** · write 0x0001 (arg) · call 0x0002 (args)**!**

Correctness of the Trace semantics

- formalise the trace semantics

Correctness of the Trace semantics

$$\mathsf{TR} = \left\{ \alpha = \left\{ \begin{array}{c} \rightarrow \\ \texttt{call}\ a\ (\overline{v}) \\ \texttt{ret}\ v \\ \texttt{write}(a, v) \\ \texttt{read}(a, v) \\ \overset{\alpha}{\Longrightarrow} \end{array} \right\} \right\}$$

- formalise the trace semantics

Correctness of the Trace semantics

$$\mathsf{TR} = \left\{ \alpha = \left\{ \begin{array}{c} \rightarrow \\ \texttt{call}\ a\ (\overline{v}) \\ \texttt{ret}\ v \\ \texttt{write}(a, v) \\ \texttt{read}(a, v) \\ \overset{\alpha}{\Longrightarrow} \end{array} \right\} \right\}$$

- formalise the trace semantics
- define traces of a program

Correctness of the Trace semantics

$$\mathsf{TR} = \left\{ \alpha = \begin{Bmatrix} \rightarrow \\ \texttt{call}\ a\ (\overline{v}) \\ \texttt{ret}\ v \\ \texttt{write}(a, v) \\ \texttt{read}(a, v) \\ \overset{\alpha}{\Longrightarrow} \end{Bmatrix} \right\}$$

$$\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P) = \{\overline{\alpha} \mid P \overset{\overline{\alpha}}{\Longrightarrow} P'\}$$

- formalise the trace semantics
- define traces of a program

Correctness of the Trace semantics

$$\mathsf{TR} = \left\{ \alpha = \left\{ \begin{array}{c} \rightarrow \\ \texttt{call}\ a\ (\overline{v}) \\ \texttt{ret}\ v \\ \texttt{write}(a, v) \\ \texttt{read}(a, v) \\ \overset{\alpha}{\Longrightarrow} \end{array} \right\} \right\}$$

$$\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P) = \{\overline{\alpha} \mid P \overset{\overline{\alpha}}{\Longrightarrow} P'\}$$

- formalise the trace semantics
- define traces of a program
- prove the semantics to be as precise as what *the rest* captured

Correctness of the Trace semantics

$$\mathsf{TR} = \left\{ \alpha = \begin{Bmatrix} \rightarrow \\ \texttt{call}\ a\ (\overline{v}) \\ \texttt{ret}\ v \\ \texttt{write}(a, v) \\ \texttt{read}(a, v) \\ \overset{\alpha}{\Longrightarrow} \end{Bmatrix} \right\}$$

$$\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P) = \{\overline{\alpha} \mid P \overset{\overline{\alpha}}{\Longrightarrow} P'\}$$

- formalise the trace semantics
- define traces of a program
- prove the semantics to be as precise as what *the rest* captured

$$P_1 \simeq P_2 \iff \mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P_1) = \mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P_2)$$

Correctness of the Trace semantics

$$\mathsf{TR} = \left\{ \alpha = \left\{ \begin{array}{c} \rightarrow \\ \texttt{call}\ a\ (\overline{v}) \\ \texttt{ret}\ v \\ \texttt{write}(a, v) \\ \texttt{read}(a, v) \\ \overset{\alpha}{\Longrightarrow\!\!\!\!\rightarrow} \end{array} \right\} \right\}$$

$$\mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P) = \{\overline{\alpha} \mid P \overset{\overline{\alpha}}{\Longrightarrow\!\!\!\!\rightarrow} P'\}$$

$$P_1 \simeq P_2 \iff \mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P_1) = \mathsf{Traces}^{\mathsf{S}}_{\mathsf{A+I}}(P_2)$$

- formalise the trace semantics
- define traces of a program
- prove the semantics to be as precise as what *the rest* captured
- this defines "what the rest captures" (i.e., contextual equivalence)

Trace Semantics with Equivalence Classes

Equivalence classes!!

Secure Compilation

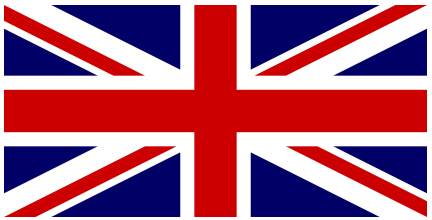
What is secure compilation?

Compilation

What is compilation?

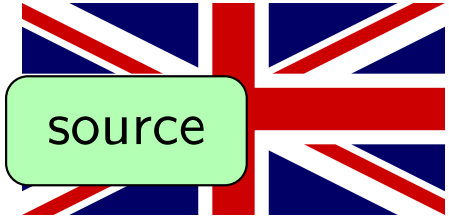
Compilation

What is compilation?



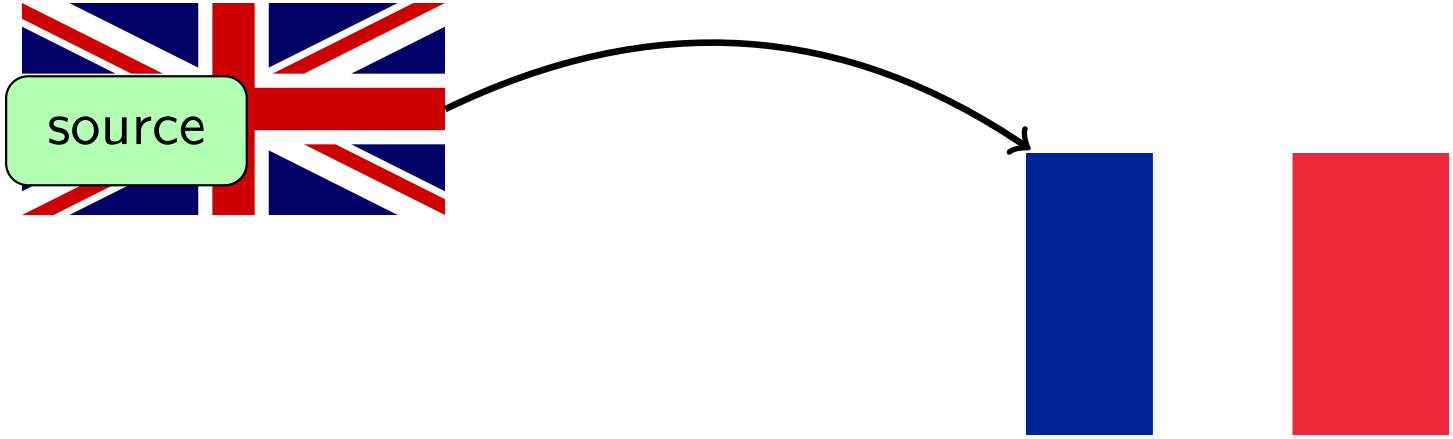
Compilation

What is compilation?



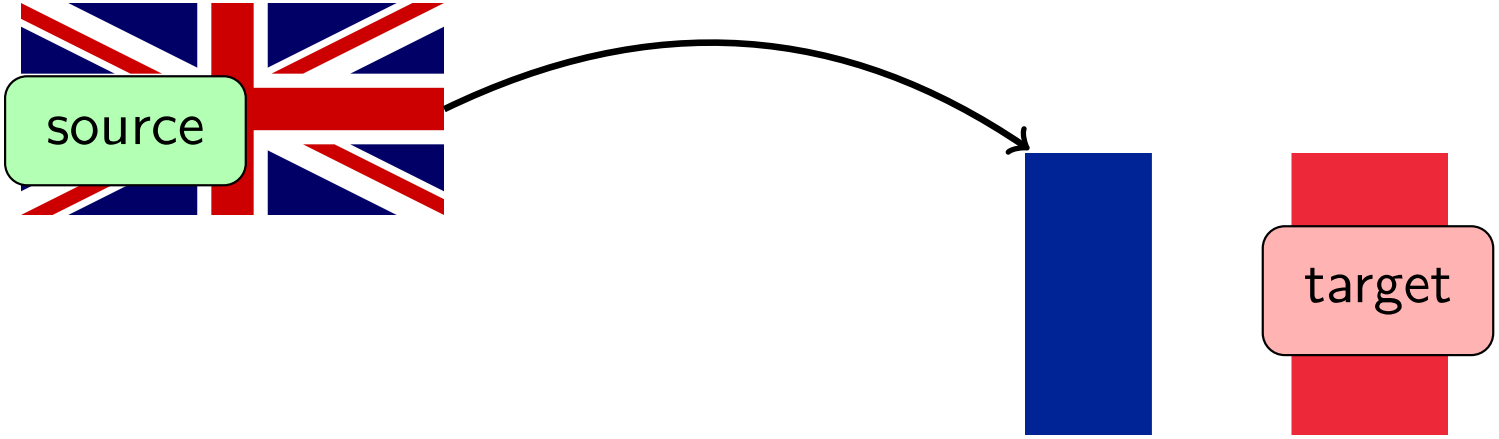
Compilation

What is compilation?



Compilation

What is compilation?

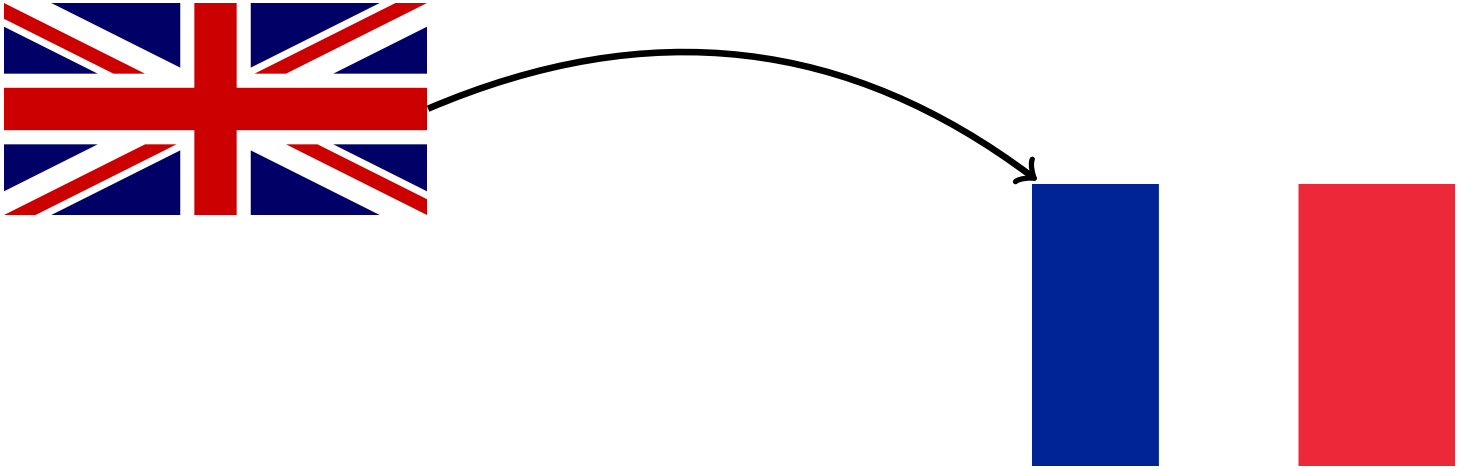


Correct Compilation

What is correct compilation?

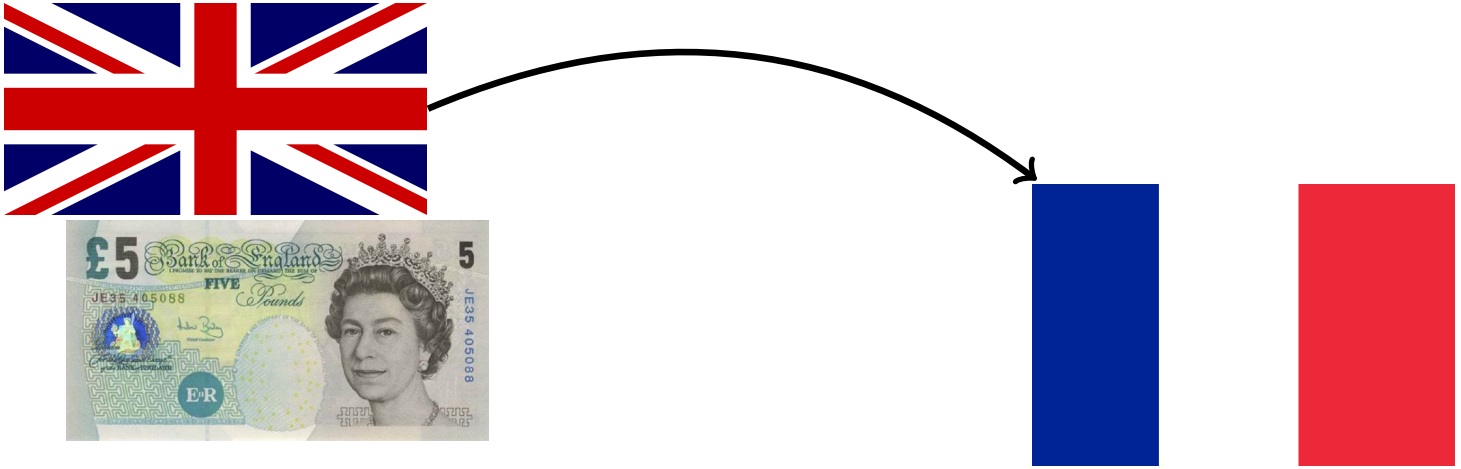
Correct Compilation

What is correct compilation?



Correct Compilation

What is correct compilation?



Correct Compilation

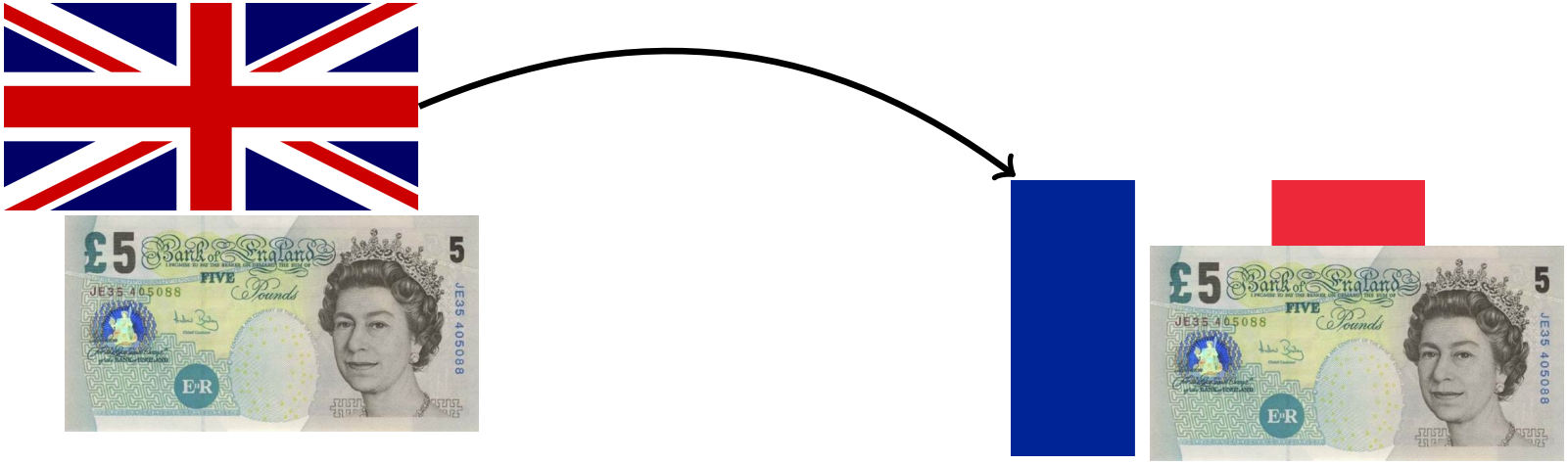
What is correct compilation?

Correct Compilation

What is correct compilation?

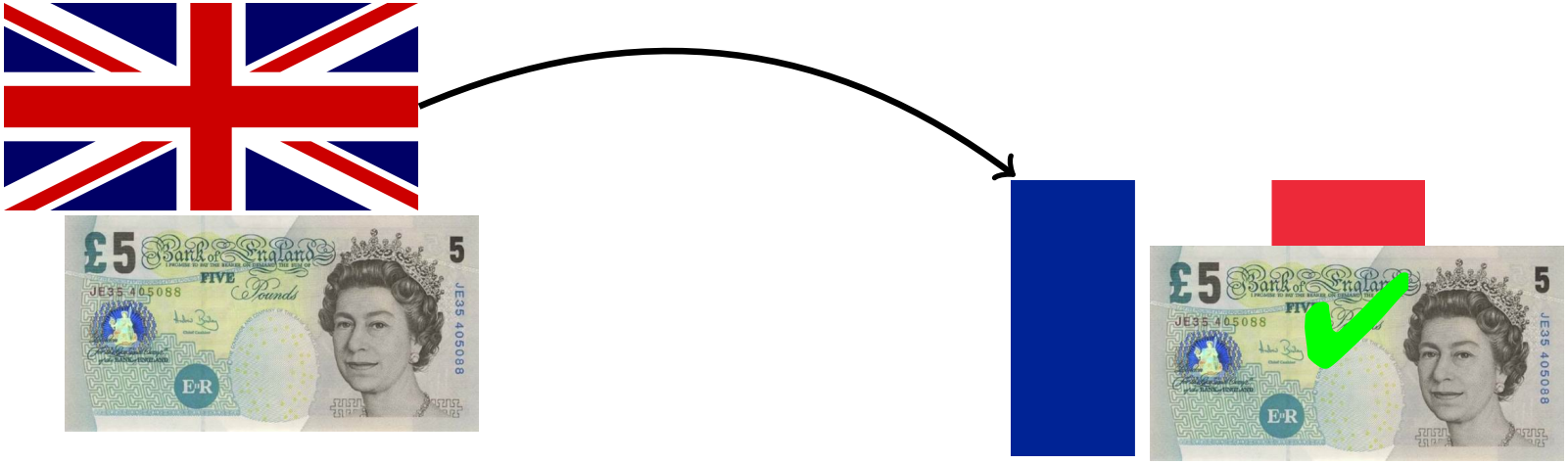
Correct Compilation

What is correct compilation?



Correct Compilation

What is correct compilation?

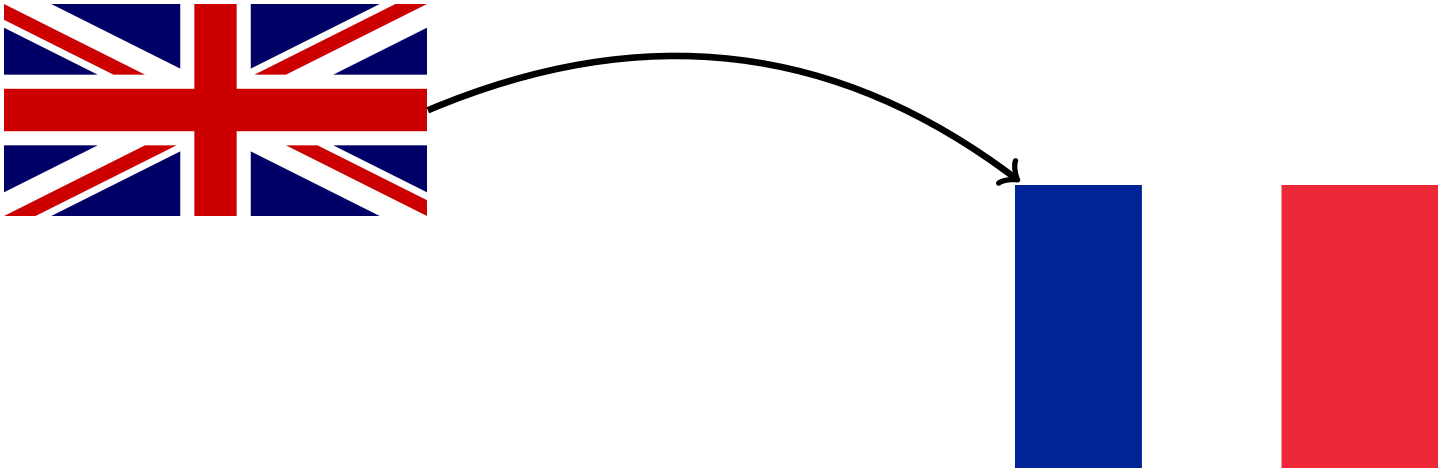


Secure Compilation

What is secure compilation?

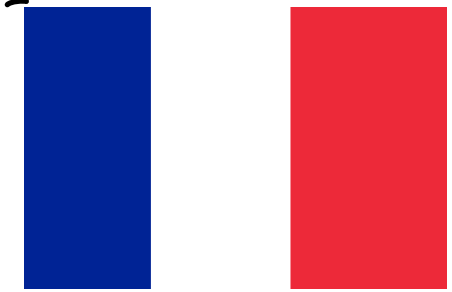
Secure Compilation

What is secure compilation?



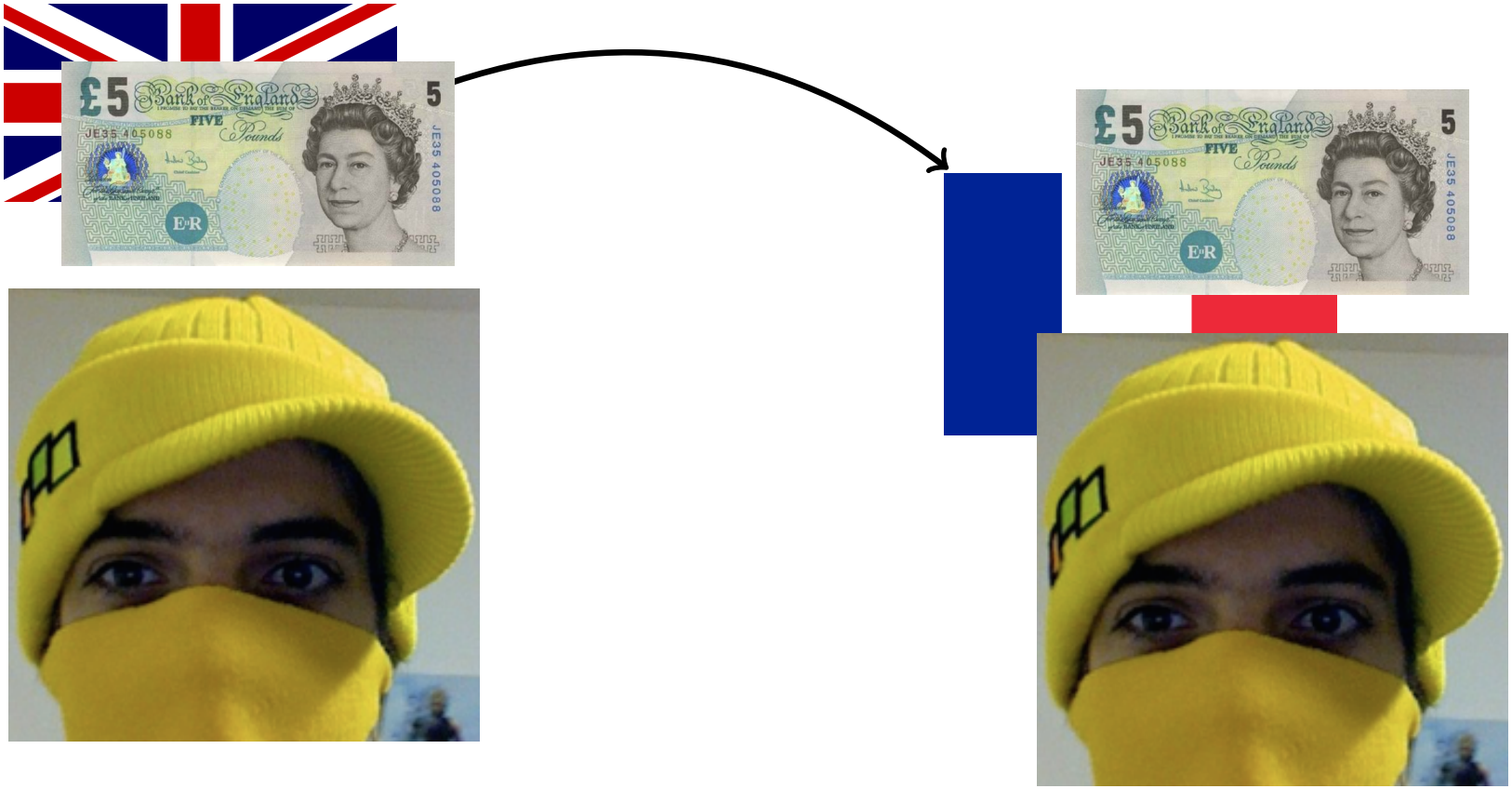
Secure Compilation

What is secure compilation?



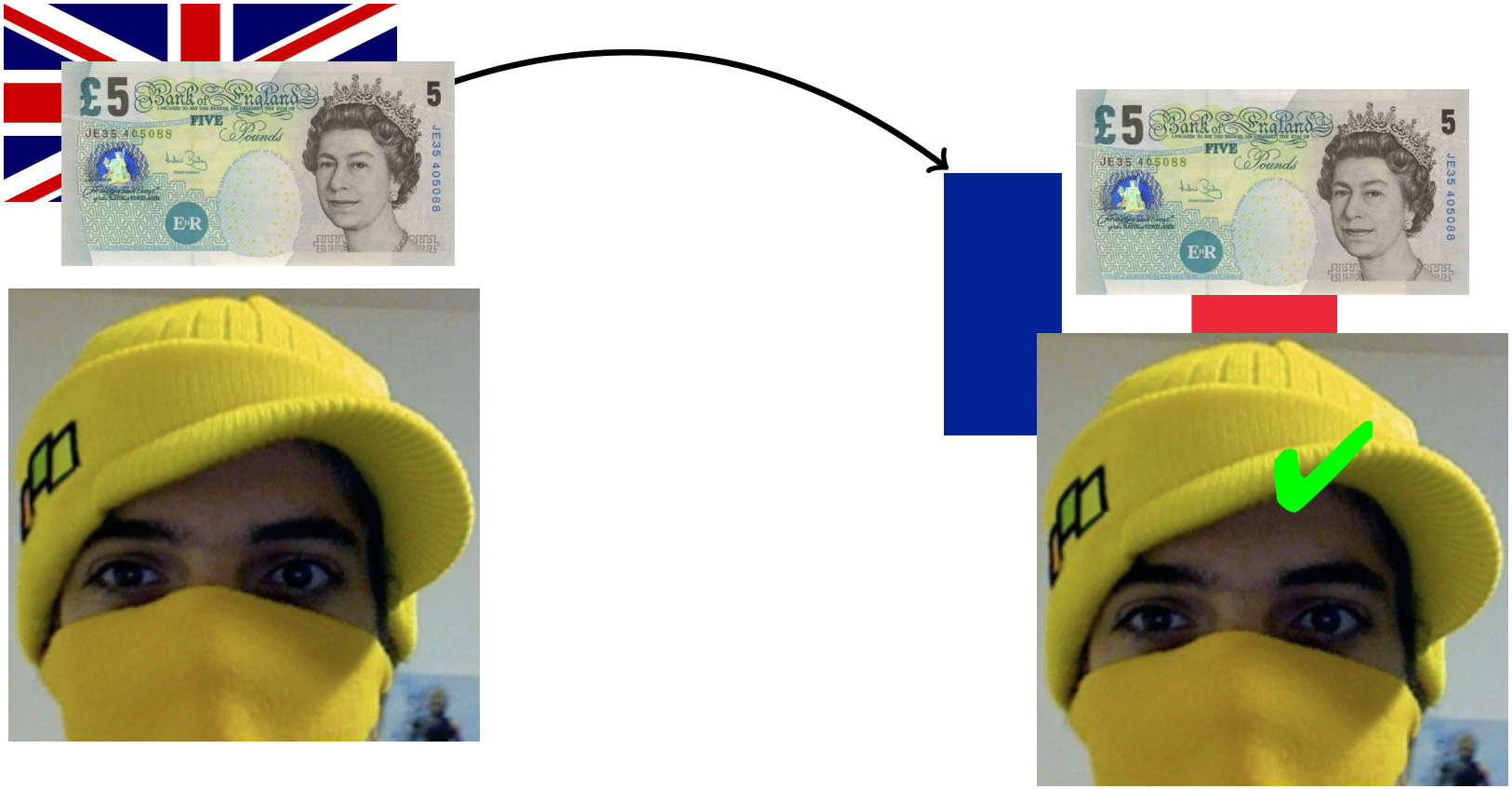
Secure Compilation

What is secure compilation?



Secure Compilation

What is secure compilation?



Secure Compilation

What is secure compilation?

Secure Compilation

What is secure compilation?

Secure Compilation

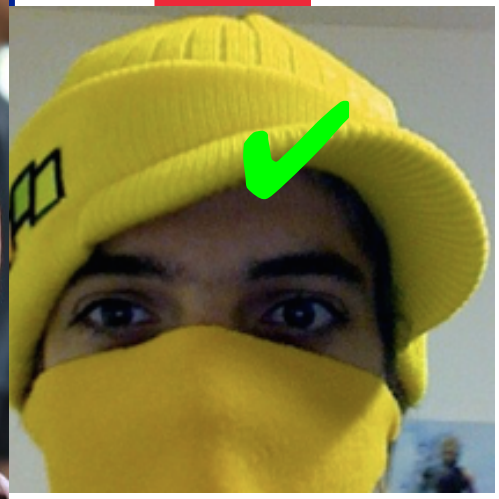
What is secure compilation?

Secure Compilation

What is secure compilation?

Source security properties

Hold @ target



Secure Compilation

What is secure compilation?

J+E: Java-like Language

- component-based
- private fields
- programming to an interface
- exceptions

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4    }
5    extern extAccount : Account;
6
7  package PE;
8    class AccountClass
9      implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account {
12       return new PE.AccountClass();
13     }
14
15     private counter : Int;
16   }
17   object extAccount : AccountClass;
```

J+E: Java-like Language

- component-based
- private fields
- programming to an interface
- exceptions
- **Q:** What is secure in this code?

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4    }
5    extern extAccount : Account;
6
7  package PE;
8    class AccountClass
9      implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account {
12       return new PE.AccountClass();
13     }
14
15     private counter : Int;
16   }
17   object extAccount : AccountClass;
```

J+E: Java-like Language

- component-based
- private fields
- programming to an interface
- exceptions
- **Q:** What is secure in this code?

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4    }
5    extern extAccount : Account;
6
7  package PE;
8    class AccountClass
9      implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account {
12       return new PE.AccountClass();
13     }
14
15     private counter : Int;
16   }
17   object extAccount : AccountClass;
```

J+E: Java-like Language

- component-based
- private fields
- programming to an interface
- exceptions
- **Q:** What is secure in this code?
- **Q:** How do we securely compile this code?

```
package PI;
  interface Account {
    public createAccount() : Foo;
  }
  extern extAccount : Account;

package PE;
  class AccountClass
    implements PI.Account {
    AccountClass() { counter = 0; }
    public createAccount() : Account {
      return new PE.AccountClass();
    }

    private counter : Int;
  }
  object extAccount : AccountClass;
```

J+E: Java-like Language

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4    }
5    extern extAccount : Account;
6
7  package PE;
8    class AccountClass
9      implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account {
12       return new PE.AccountClass();
13     }
14
15     private counter : Int;
16   }
17   object extAccount : AccountClass;
```

J+E: Java-like Language

Dynamic dispatch

---

v-tables

Secure stack

```
package PI;
  interface Account {
    public createAccount() : Foo;
  }
  extern extAccount : Account;

package PE;
  class AccountClass
    implements PI.Account {
    AccountClass() { counter = 0; }
    public createAccount() : Account {
      return new PE.AccountClass();
    }

    private counter : Int;
  }
  object extAccount : AccountClass;
```

J+E: Java-like Language

- proxy to `createAccount`

Dynamic dispatch

---

v-tables

Secure stack

```
package PI;
  interface Account {
    public createAccount() : Foo;
  }
  extern extAccount : Account;

package PE;
  class AccountClass
    implements PI.Account {
    AccountClass() { counter = 0; }
    public createAccount() : Account {
      return new PE.AccountClass();
    }

    private counter : Int;
  }
  object extAccount : AccountClass;
```

J+E: Java-like Language

- proxy to `createAccount`

  `createAccount` body

  constructor

  Dynamic dispatch

---

  v-tables

  Secure stack

  `extAccount`
  `counter`

```
1  package PI;
2    interface Account {
3      public createAccount() : Foo;
4    }
5    extern extAccount : Account;
6
7  package PE;
8    class AccountClass
9      implements PI.Account {
10     AccountClass() { counter = 0; }
11     public createAccount() : Account {
12       return new PE.AccountClass();
13     }
14
15     private counter : Int;
16   }
17   object extAccount : AccountClass;
```

PMA for Secure Compilation

Source level

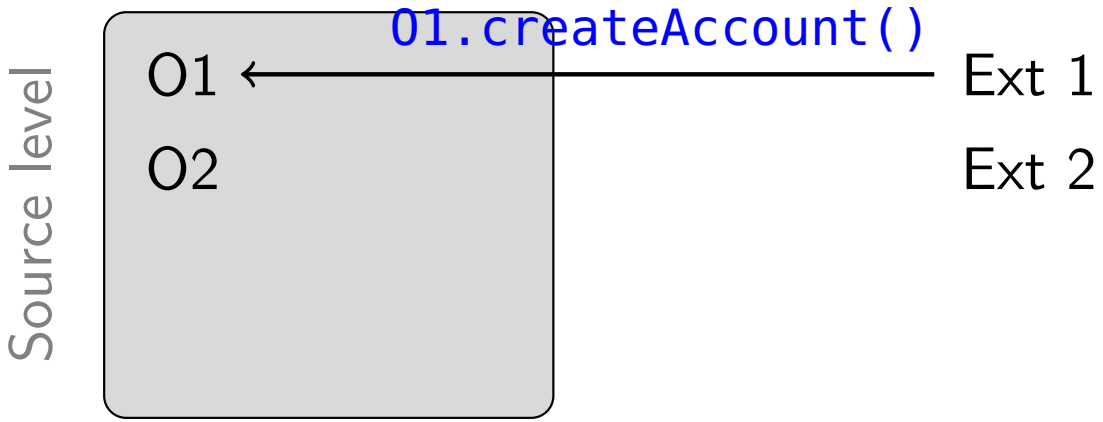
O1

O2

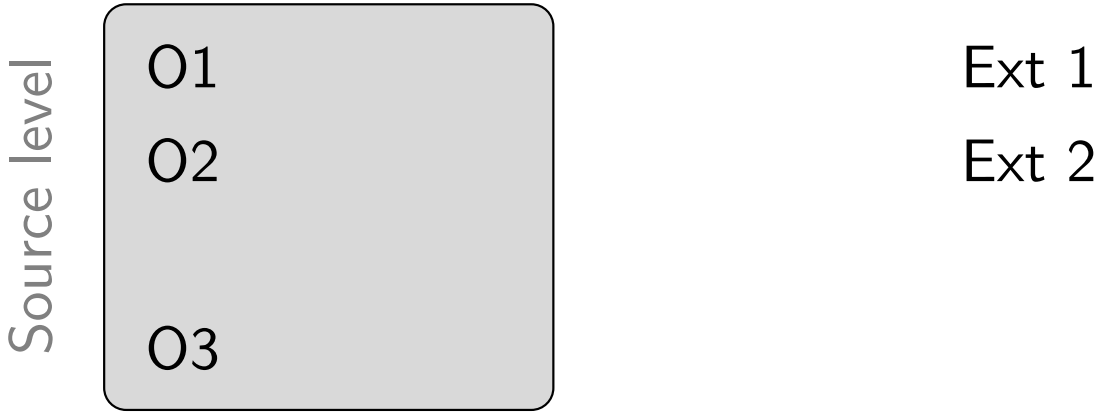
Ext 1

Ext 2

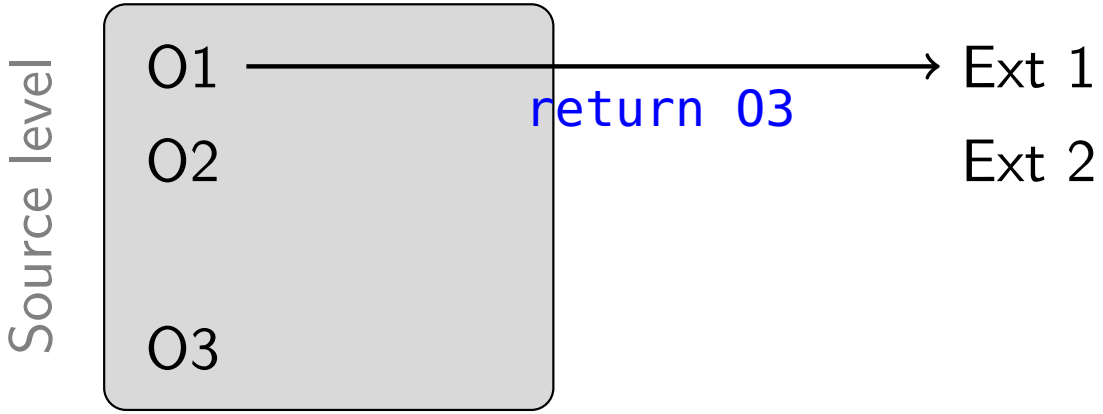
PMA for Secure Compilation



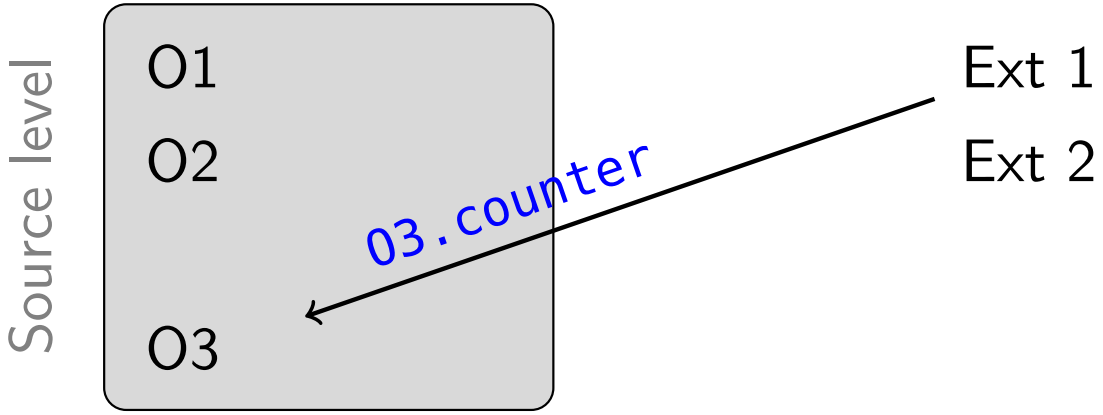
PMA for Secure Compilation



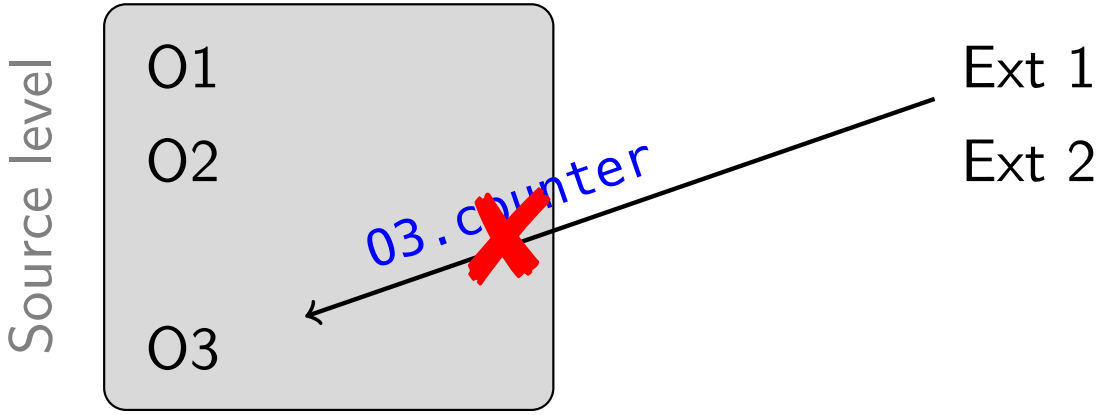
PMA for Secure Compilation



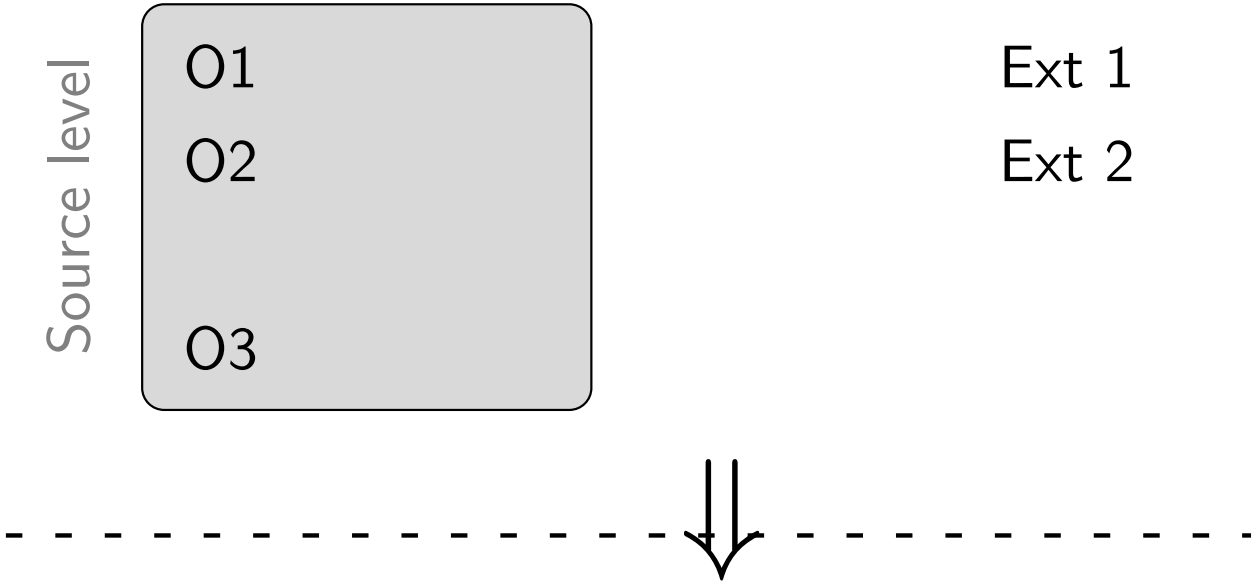
PMA for Secure Compilation



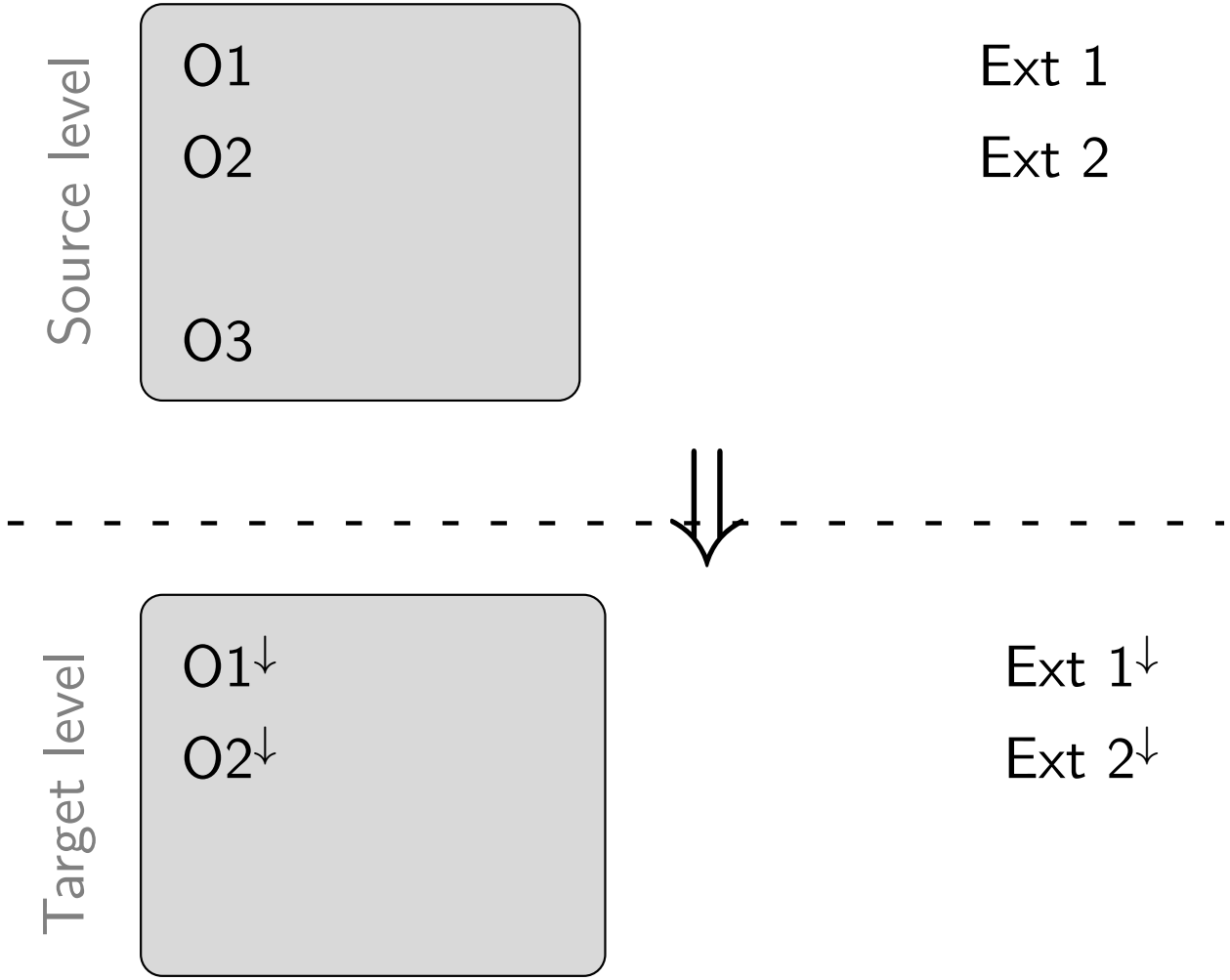
PMA for Secure Compilation



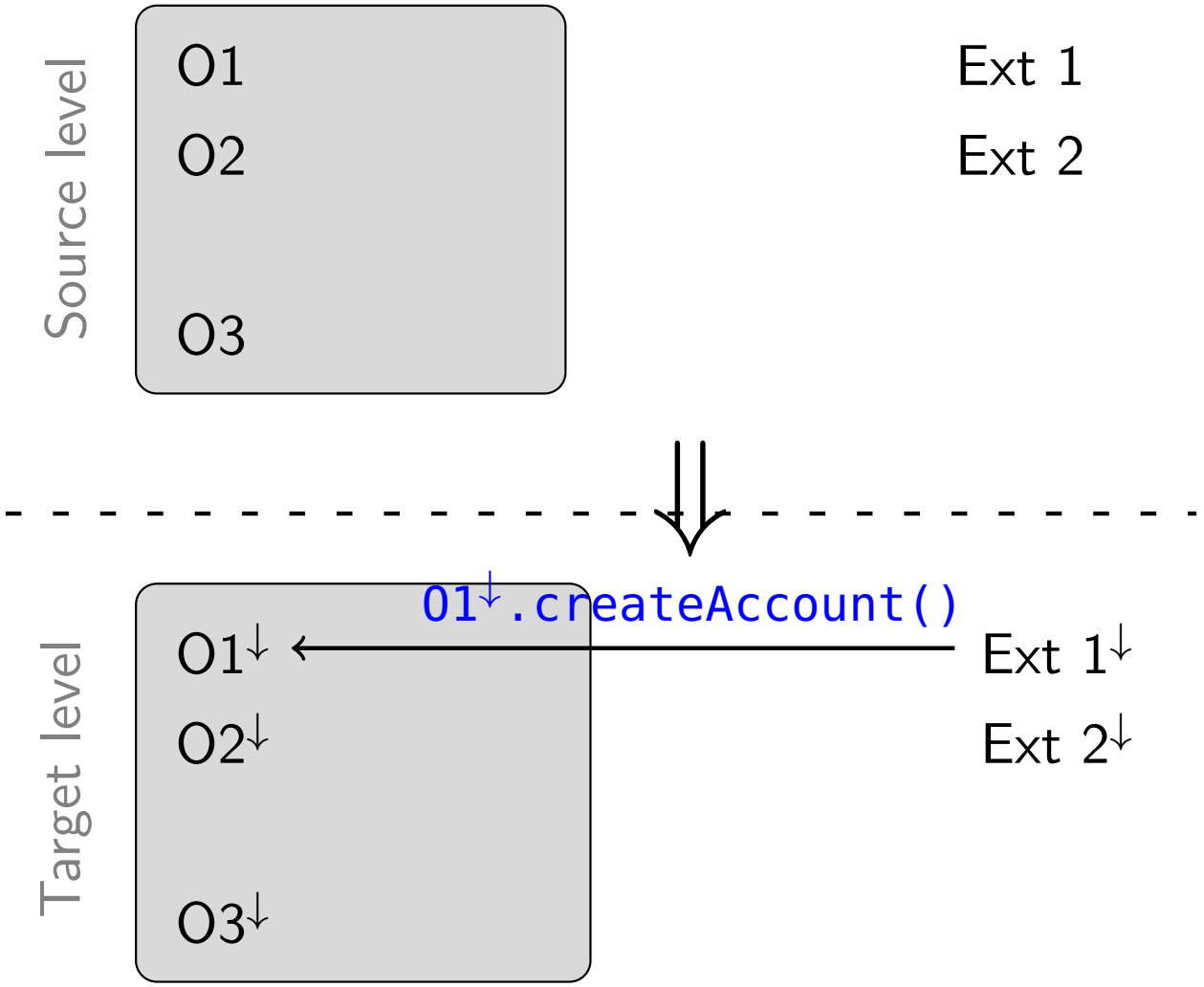
PMA for Secure Compilation



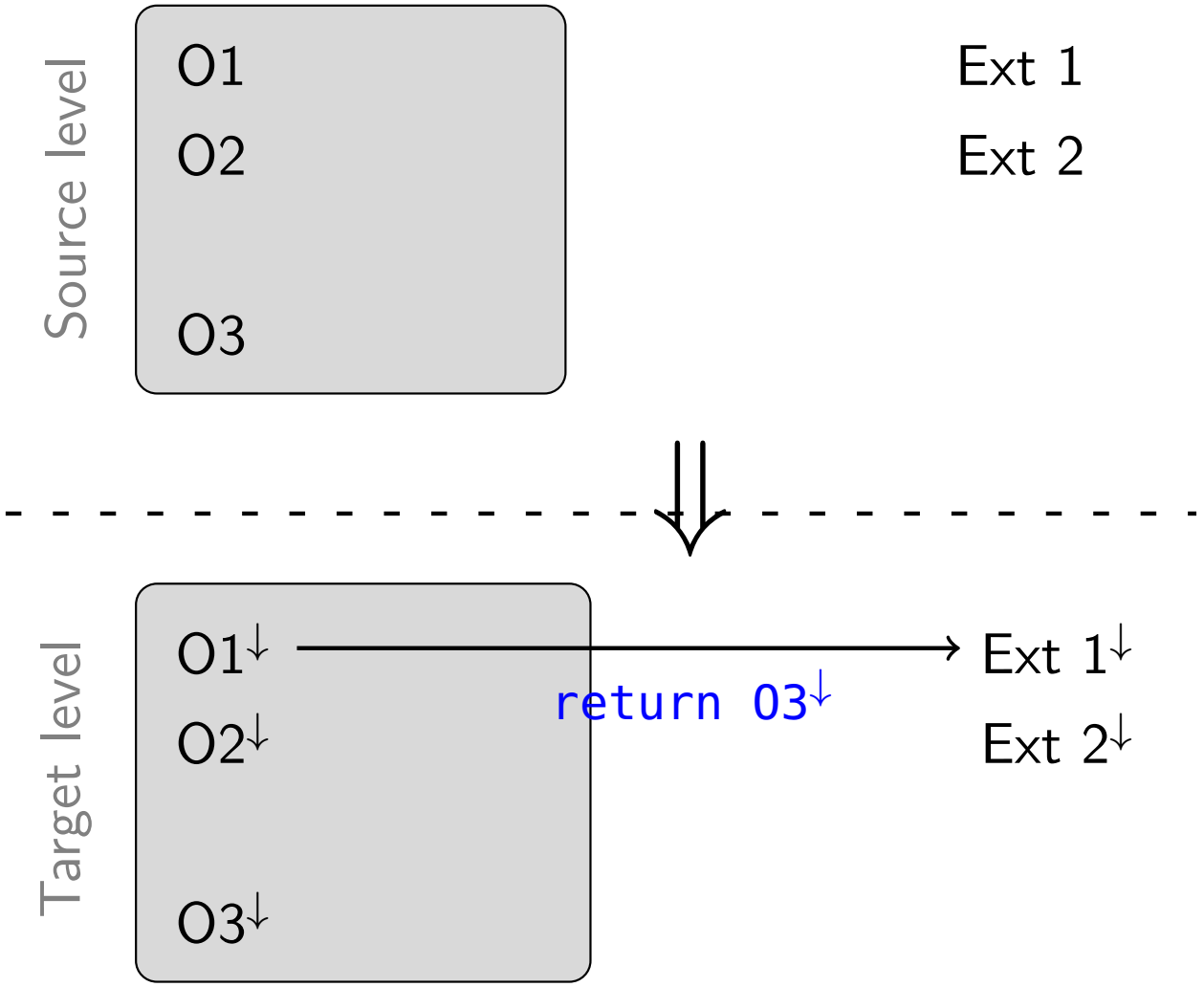
PMA for Secure Compilation



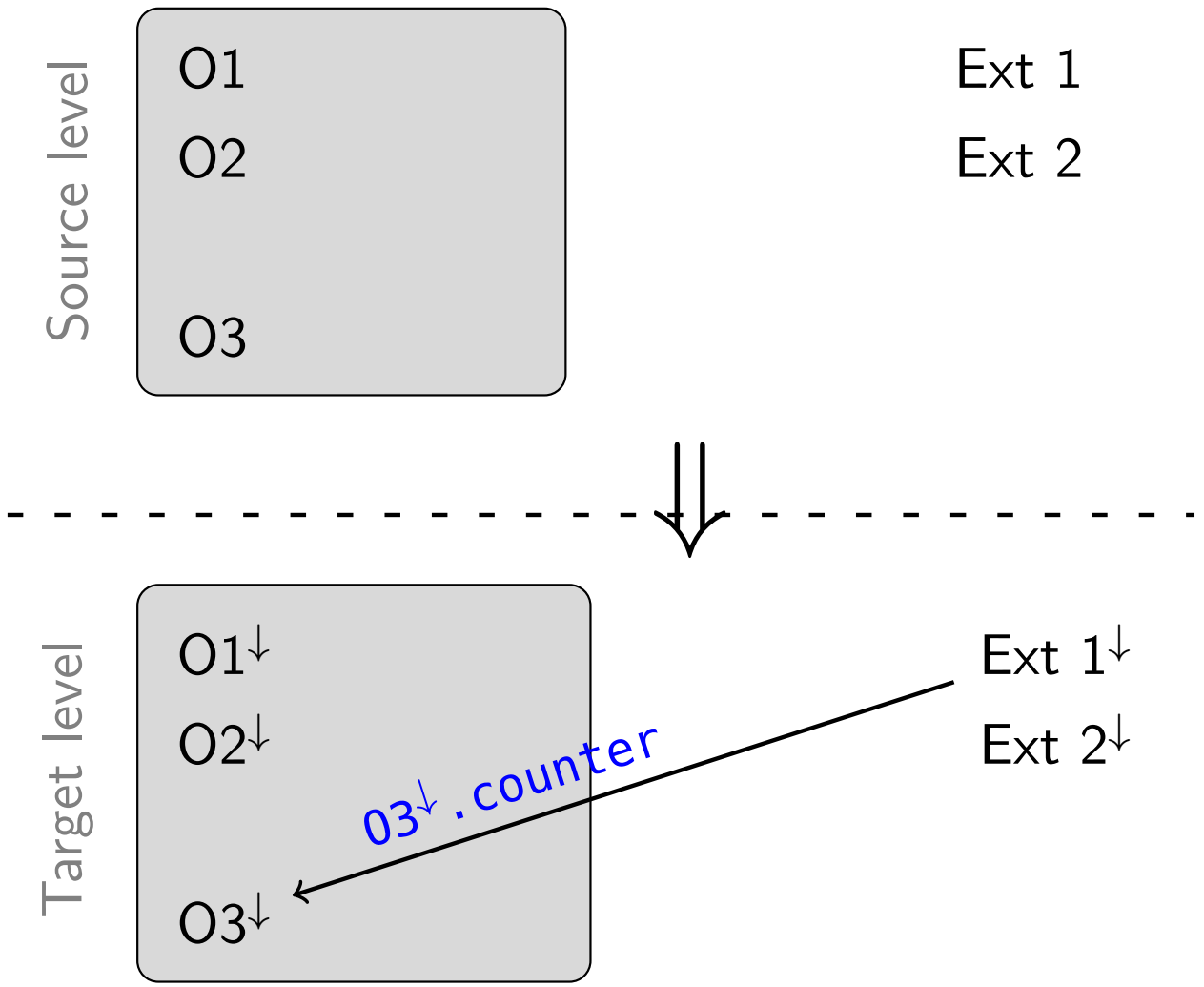
PMA for Secure Compilation



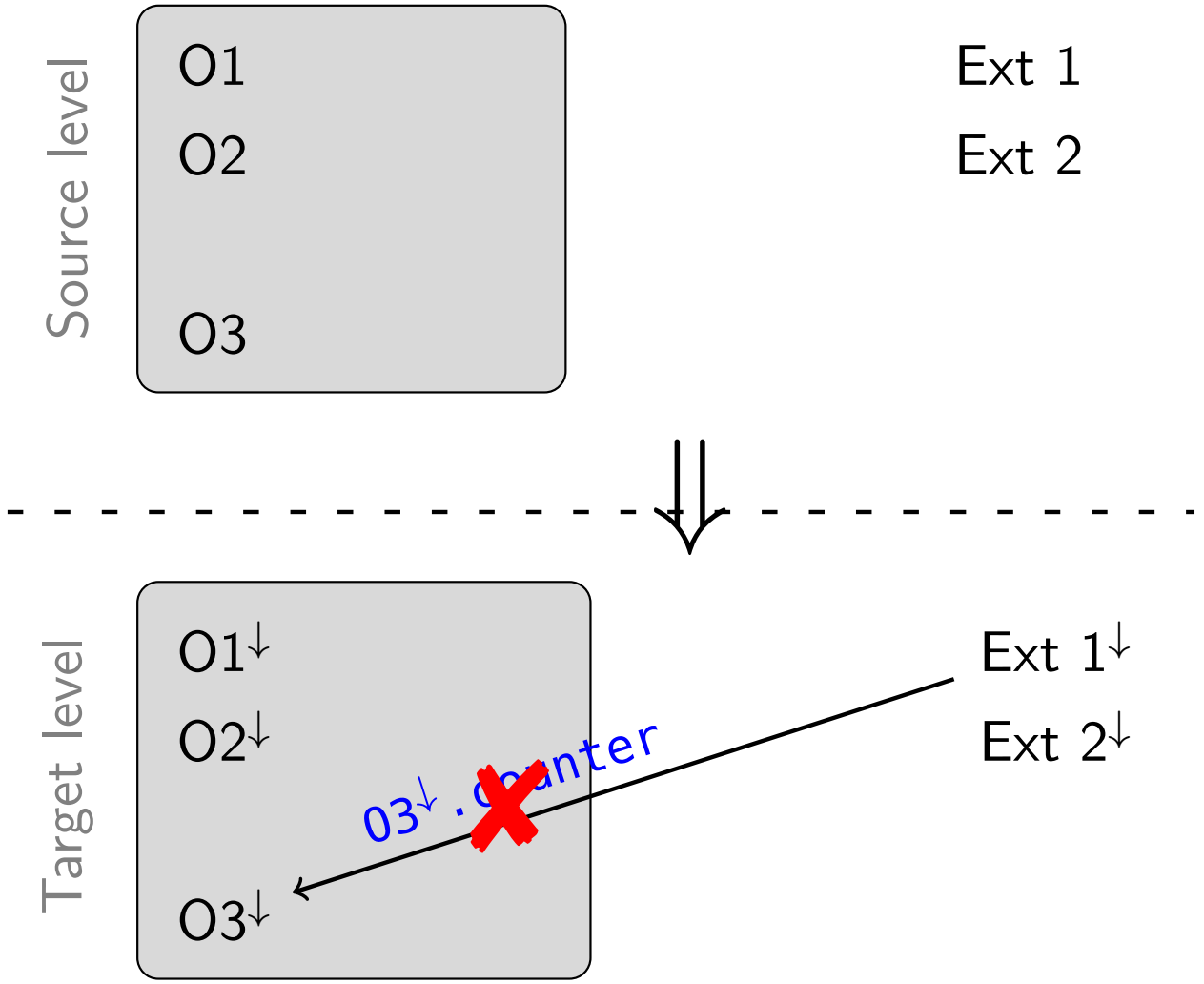
PMA for Secure Compilation



PMA for Secure Compilation

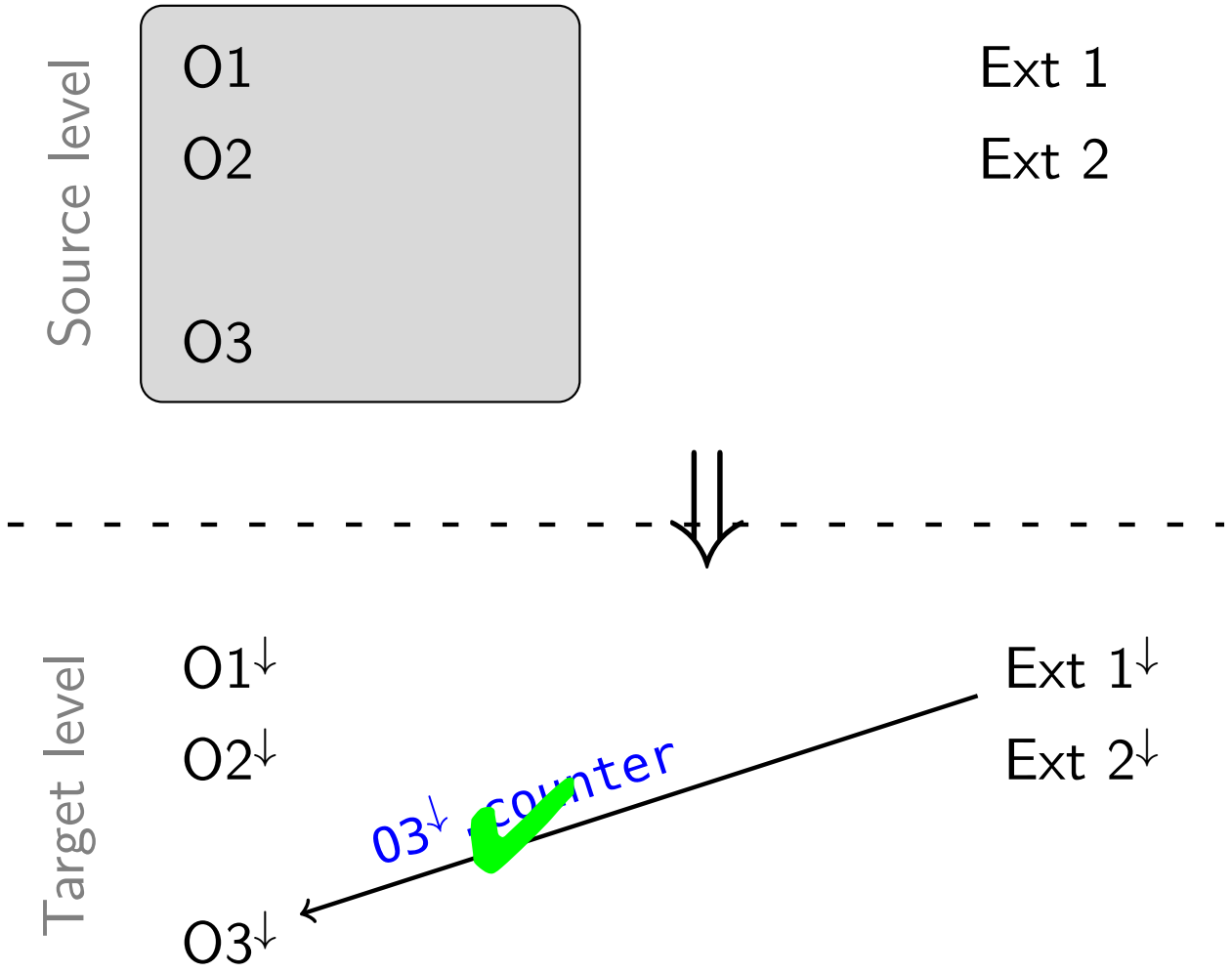


PMA for Secure Compilation



- Protect against low-level attackers

PMA for Secure Compilation



- Protect against low-level attackers
- Target code is vulnerable without PMA

Secure Compilation of Outcalls

Q: : Is that all?

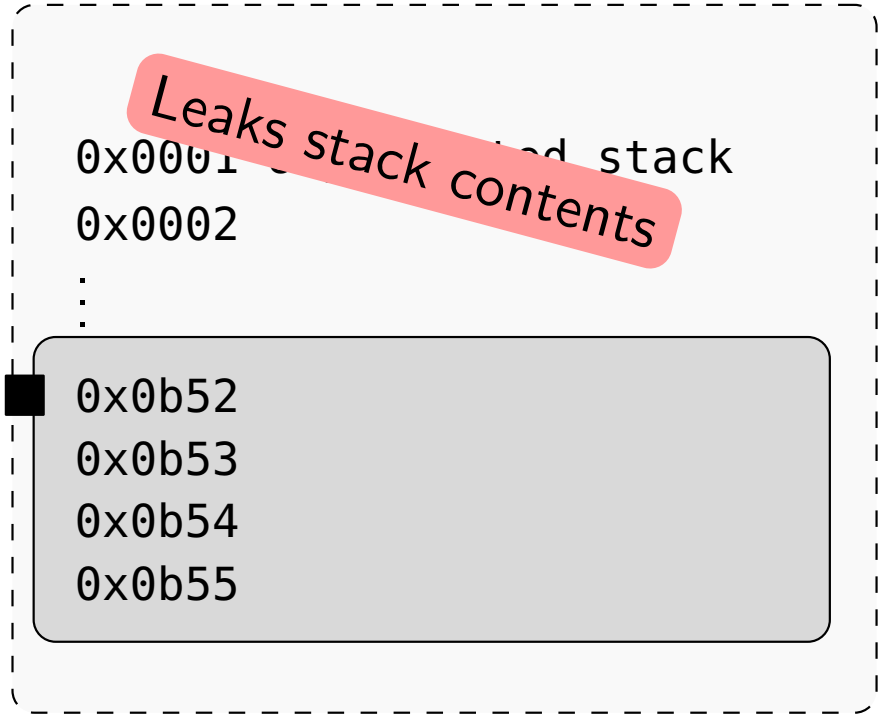
Secure Compilation of Outcalls

Q: : Is that all?

Secure Compilation of Outcalls

Q: : Is that all?

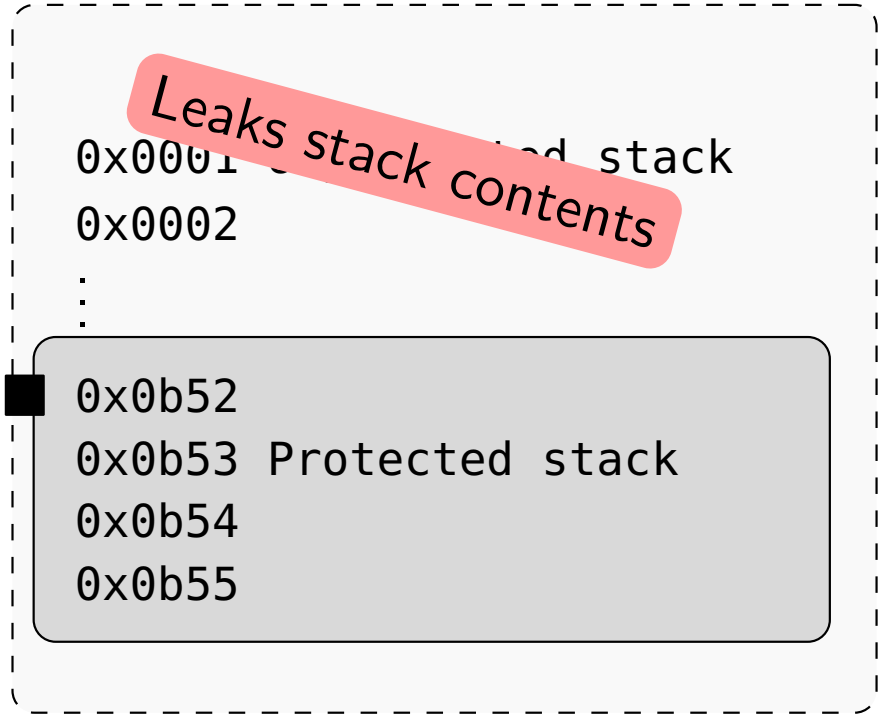
- protected stack



Secure Compilation of Outcalls

Q: : Is that all?

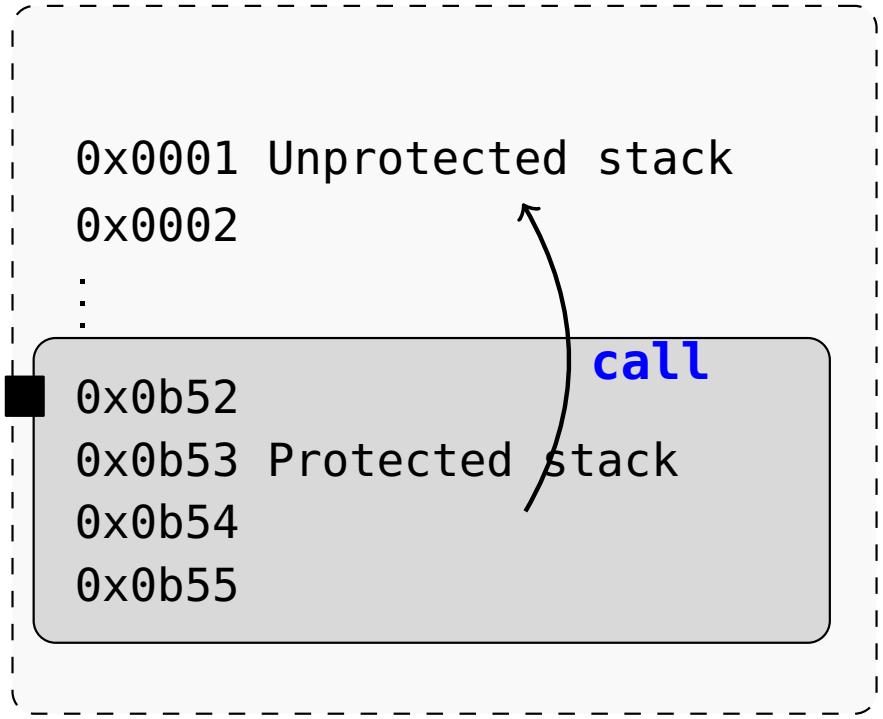
- protected stack



Secure Compilation of Outcalls

Q: : Is that all?

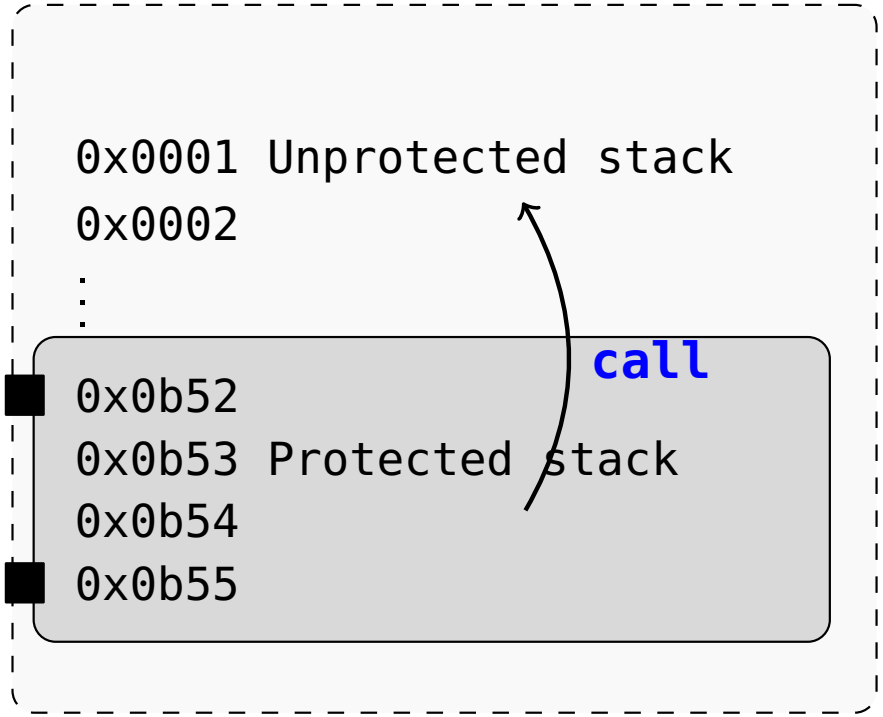
- protected stack
- returnback entry point



Secure Compilation of Outcalls

Q: : Is that all?

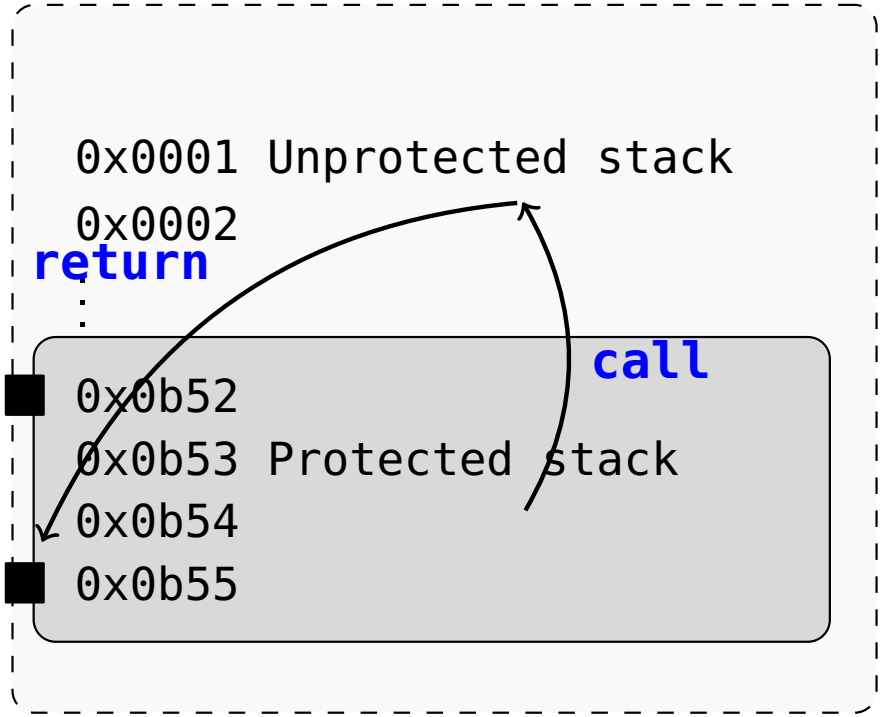
- protected stack
- returnback entry point



Secure Compilation of Outcalls

Q: : Is that all?

- protected stack
- returnback entry point



Secure Compilation of Outcalls

Q: : Is that all?

- protected stack
- returnback entry point
- reset flags and registers

Secure Compilation of Outcalls

Q: : Is that all?

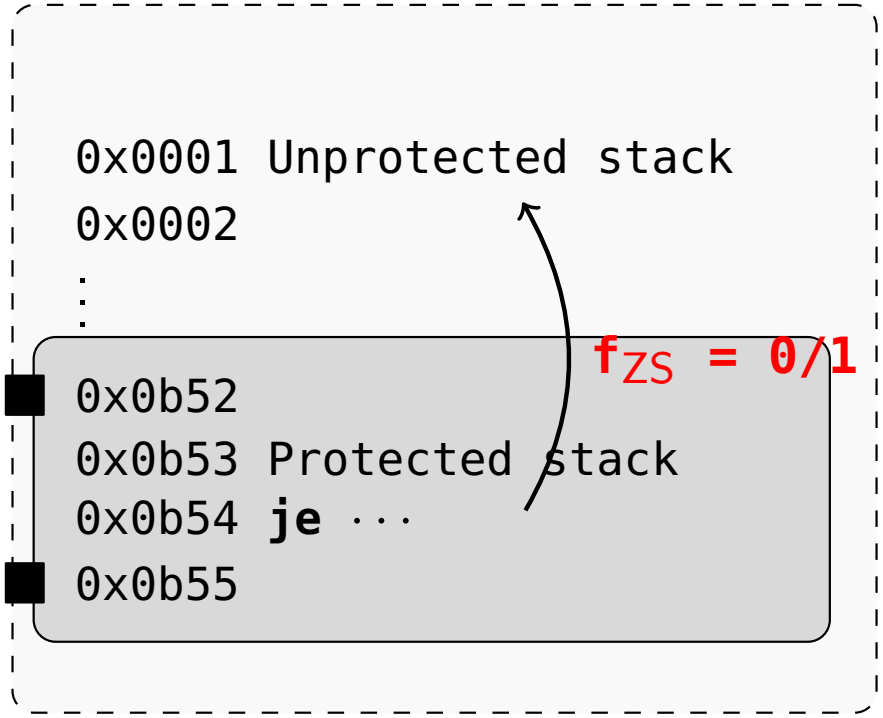
- protected stack
- returnback entry point
- reset flags and registers

```
0x0001 Unprotected stack
0x0002
⋮
0x0b52
0x0b53 Protected stack
0x0b54 je ···
0x0b55
```

Secure Compilation of Outcalls

Q: : Is that all?

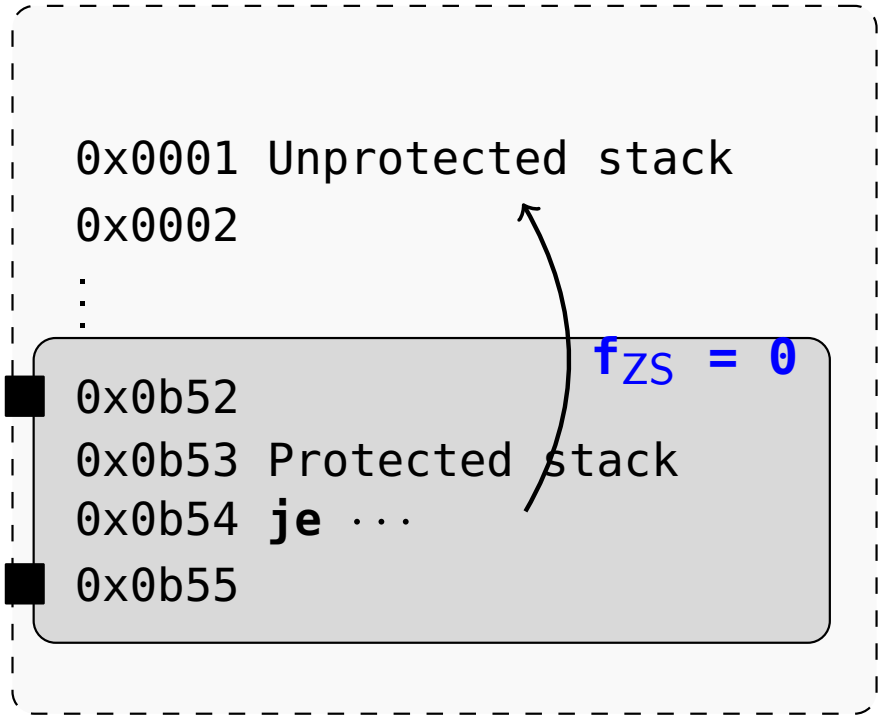
- protected stack
- returnback entry point
- reset flags and registers



Secure Compilation of Outcalls

Q: : Is that all?

- protected stack
- returnback entry point
- reset flags and registers



Secure Compilation of Outcalls

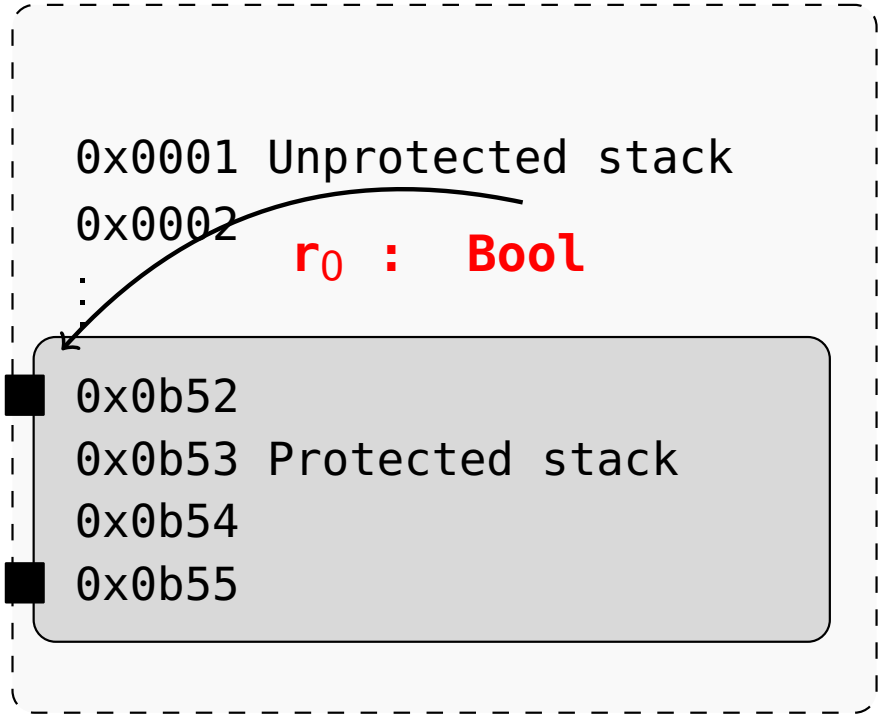
Q: : Is that all?

- protected stack
- returnback entry point
- reset flags and registers
- ground-typed values check

Secure Compilation of Outcalls

Q: : Is that all?

- protected stack
- returnback entry point
- reset flags and registers
- ground-typed values check

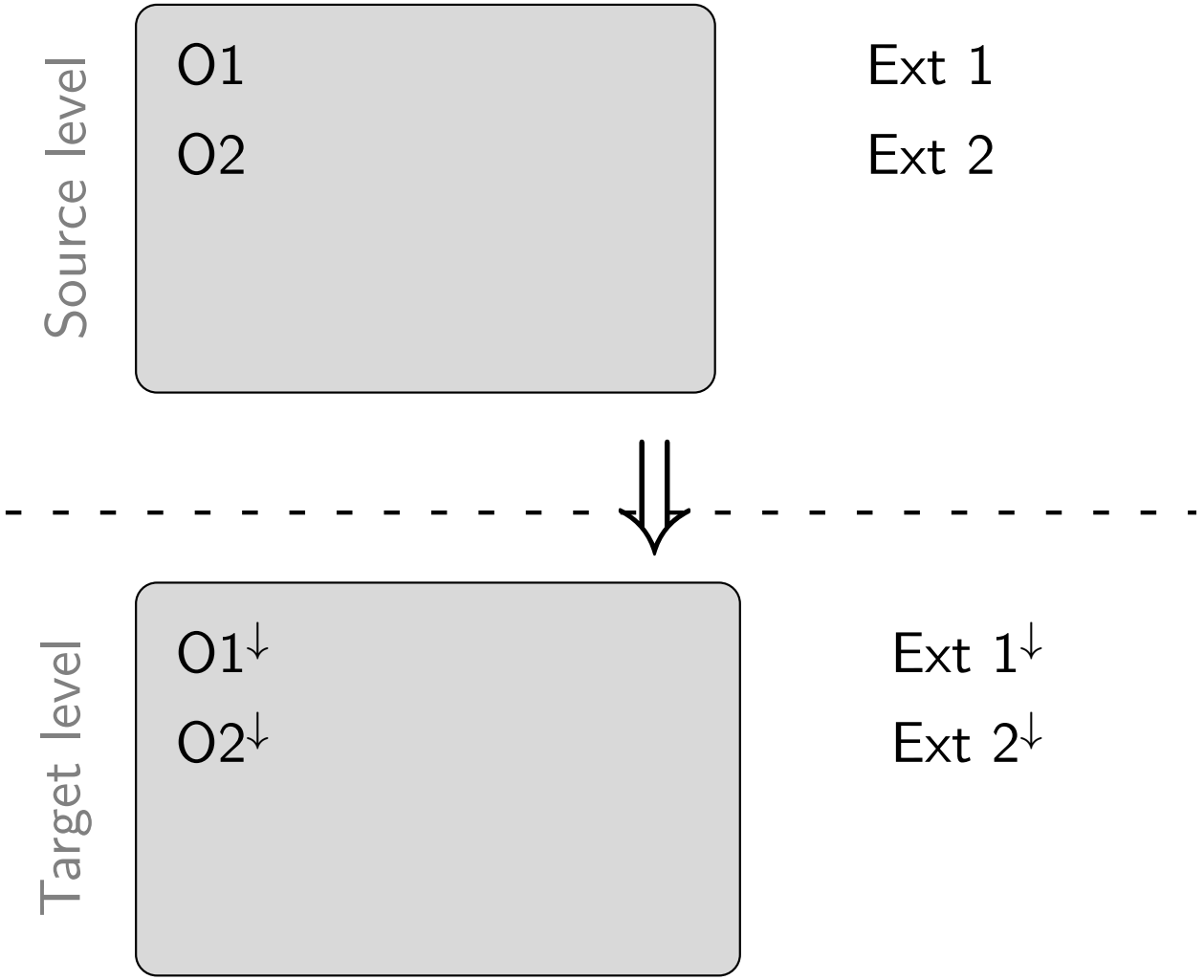


Secure Compilation of Outcalls

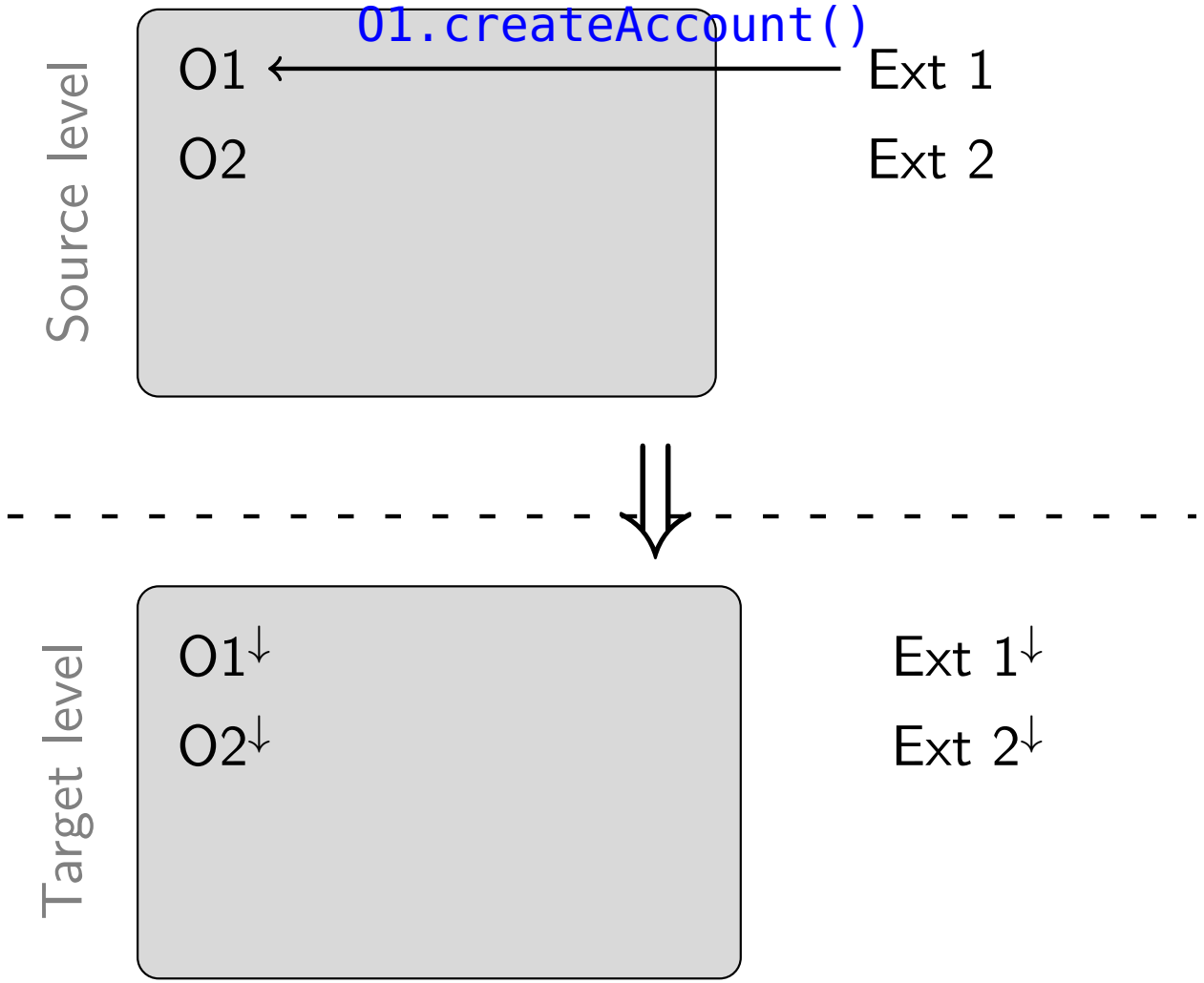
Q: : Is that all?

- protected stack
- returnback entry point
- reset flags and registers
- ground-typed values check

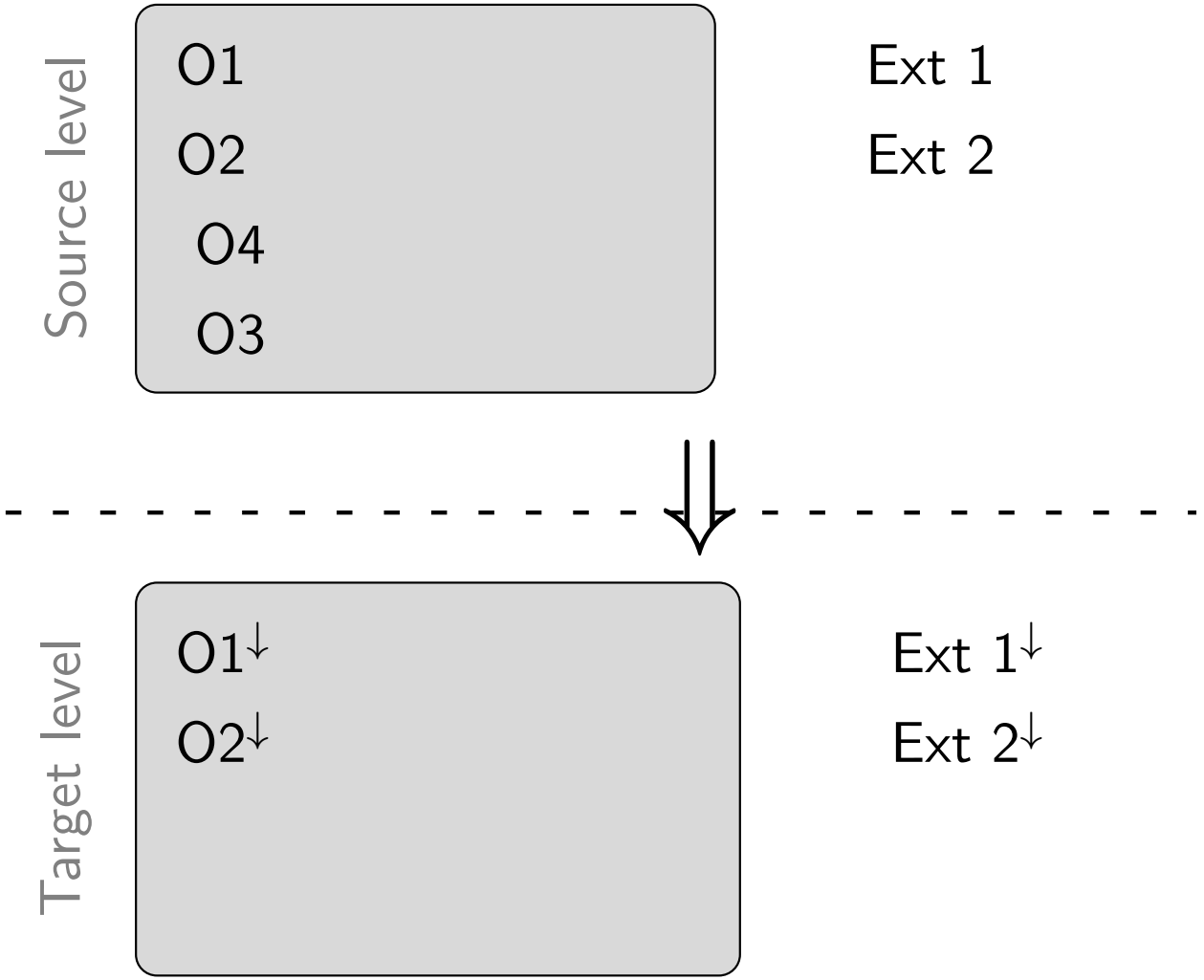
Dynamic Memory Allocation



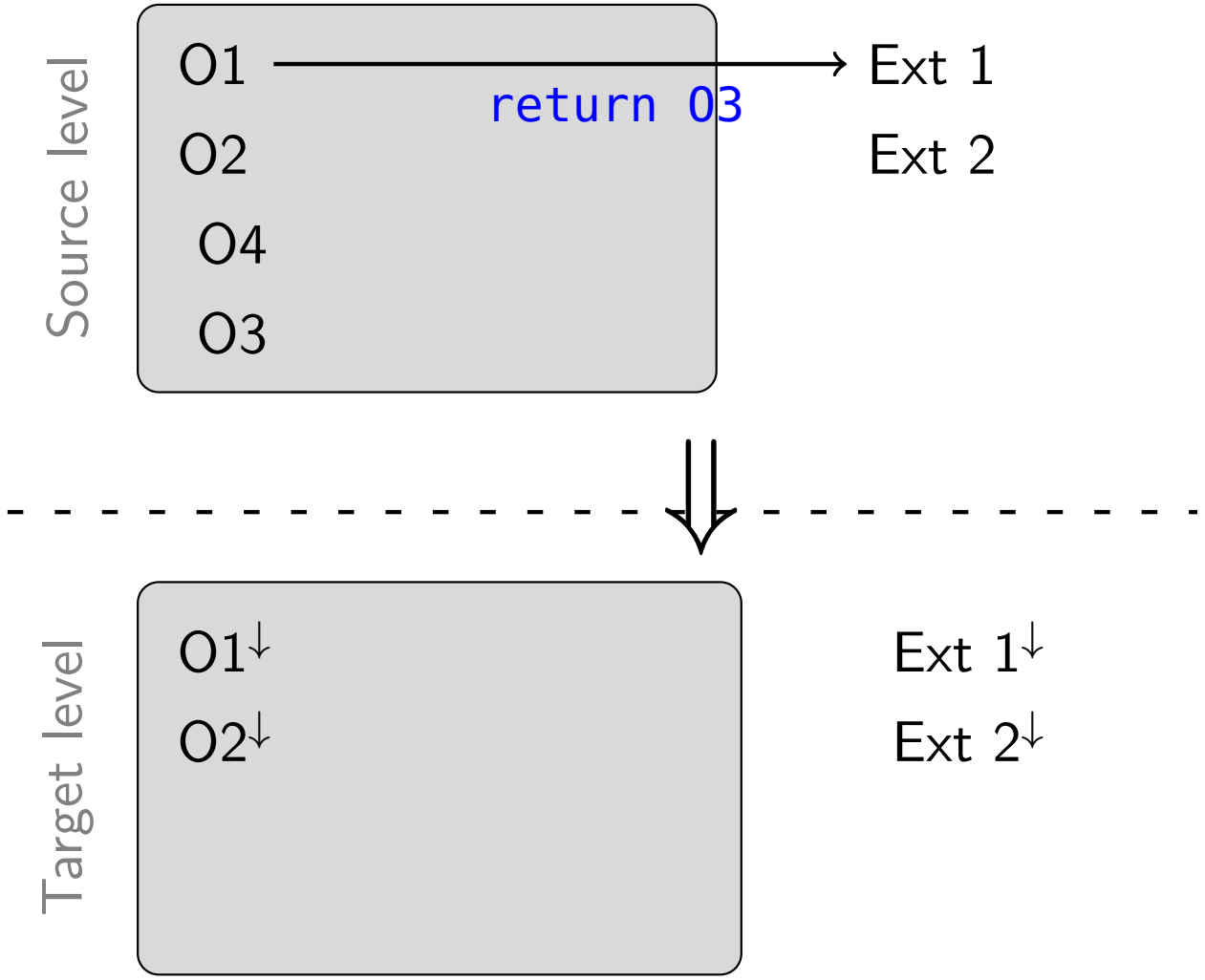
Dynamic Memory Allocation



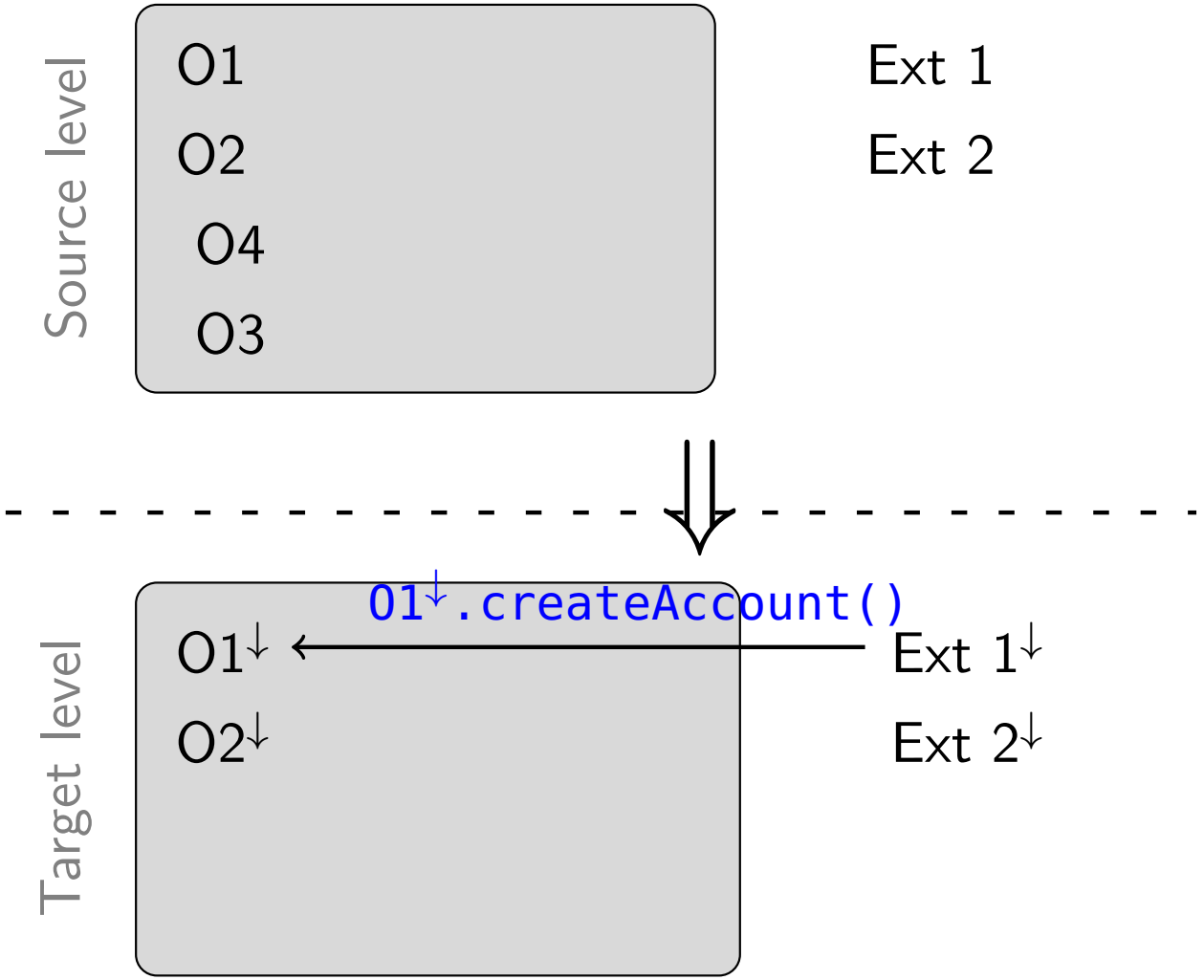
Dynamic Memory Allocation



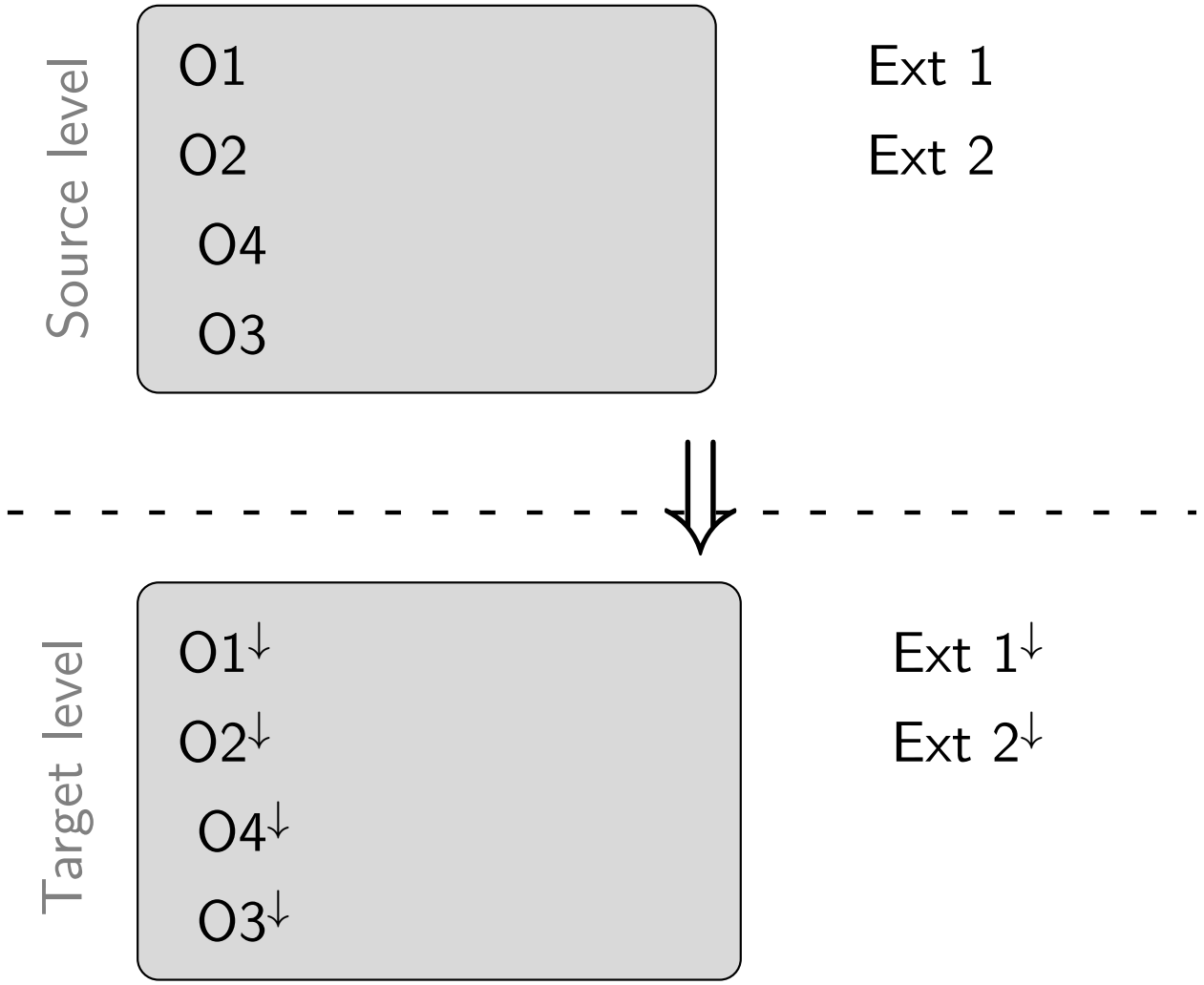
Dynamic Memory Allocation



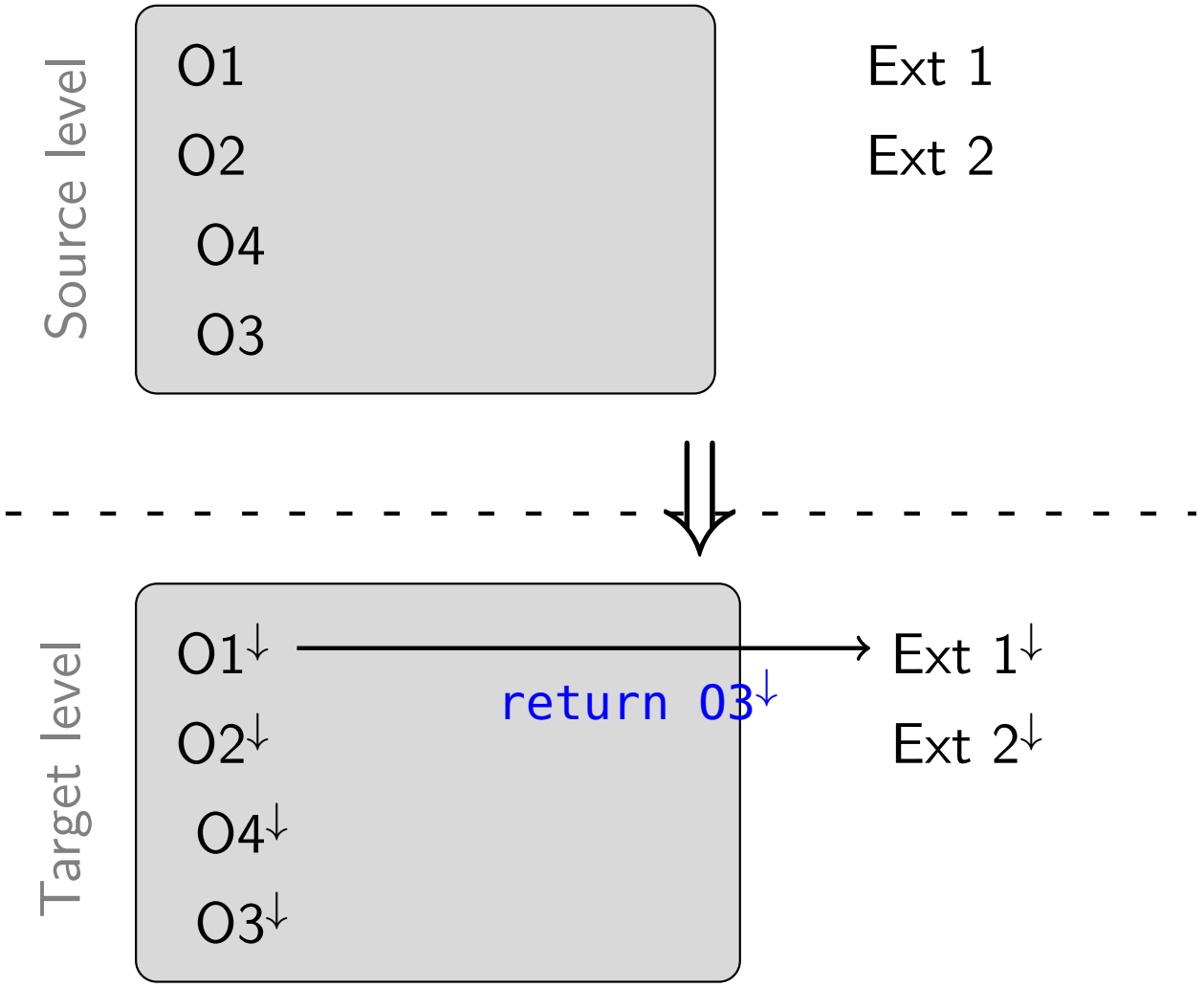
Dynamic Memory Allocation



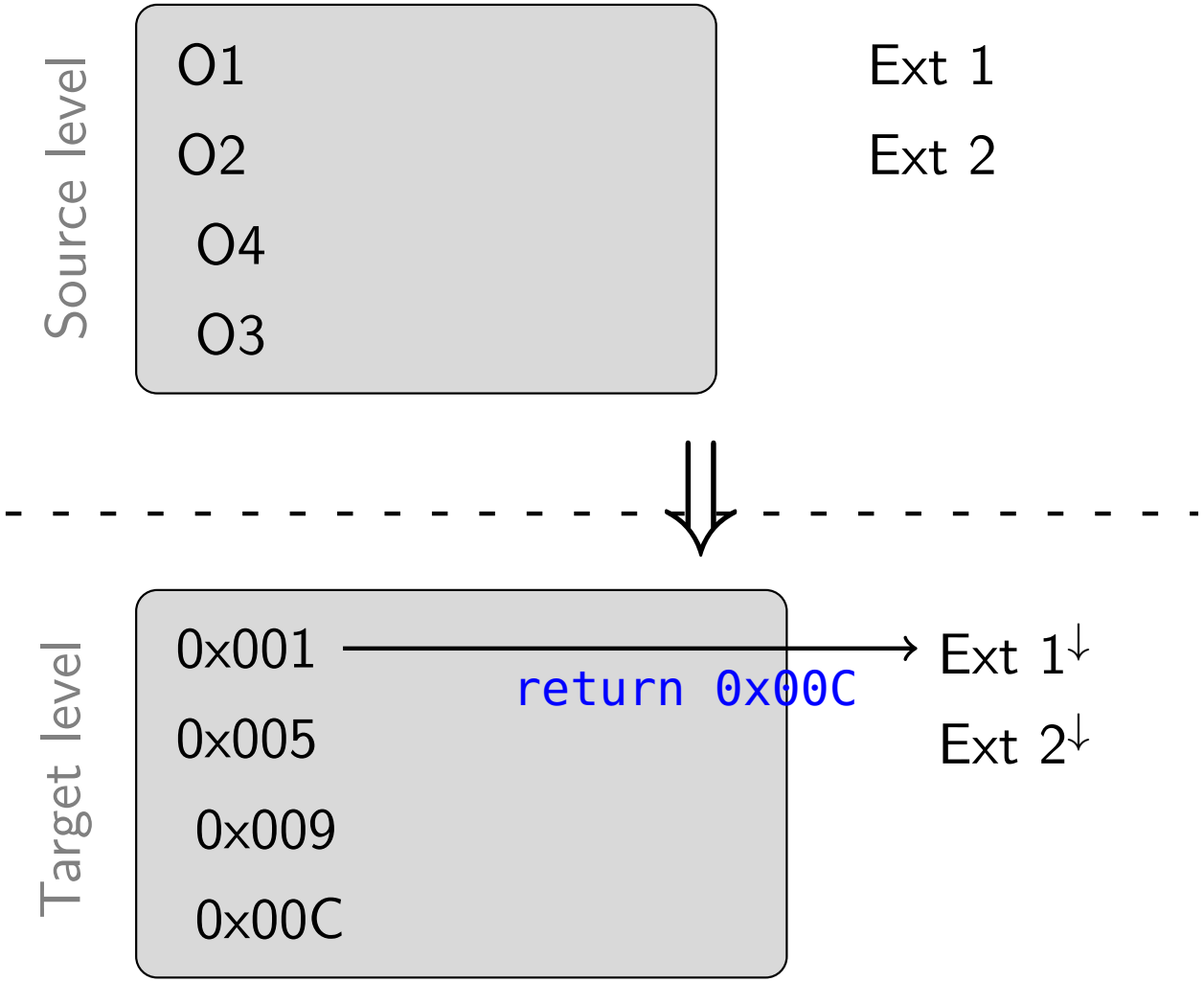
Dynamic Memory Allocation



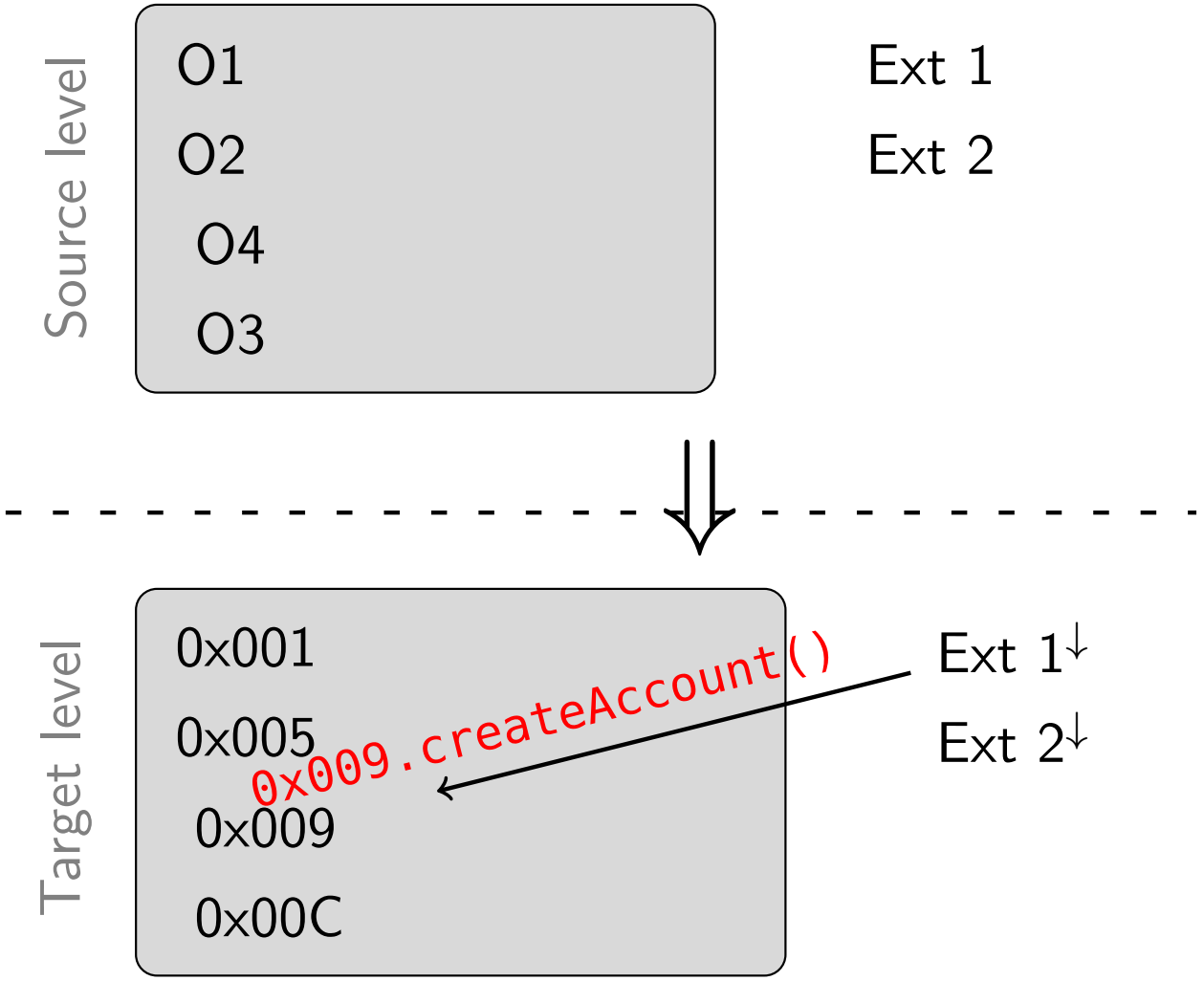
Dynamic Memory Allocation



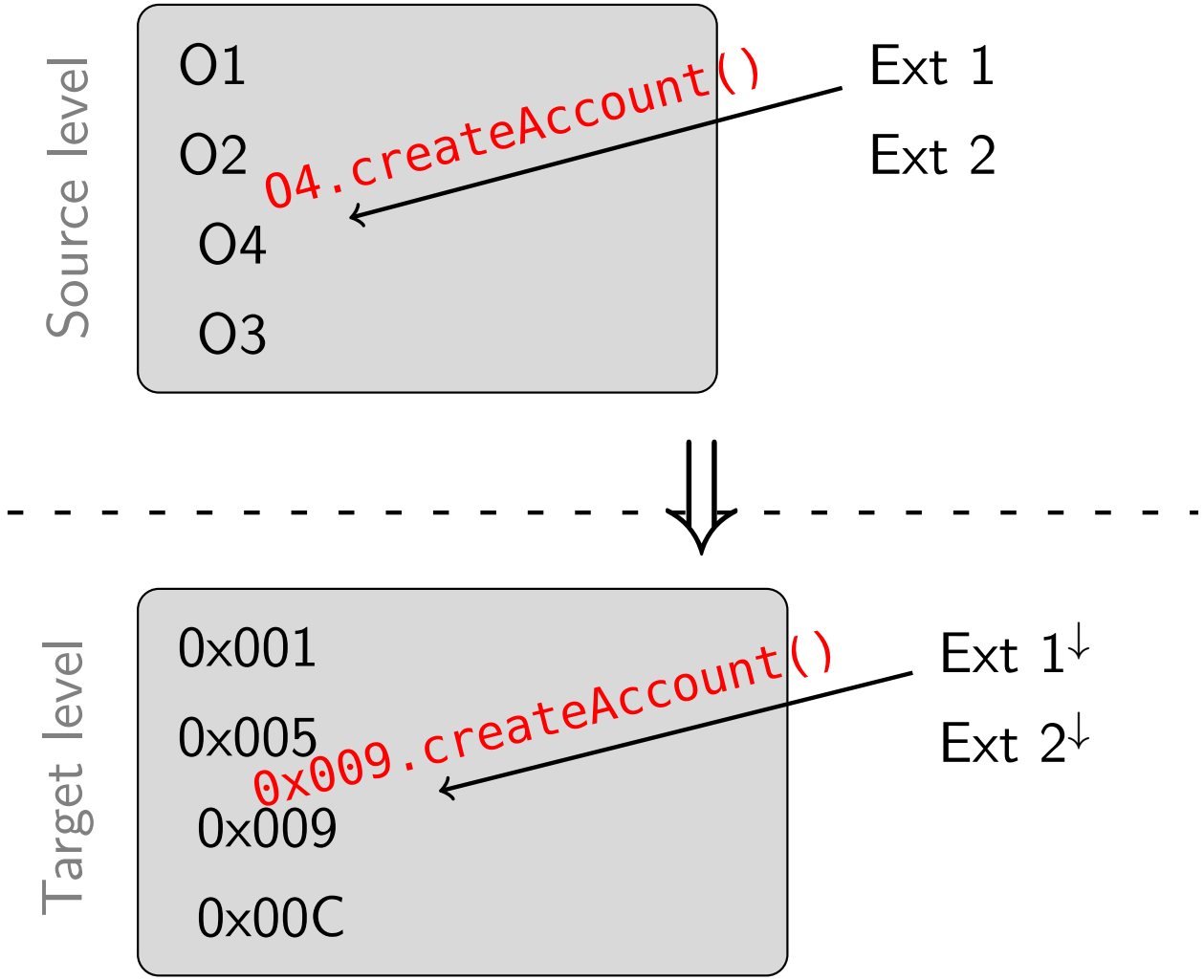
Dynamic Memory Allocation



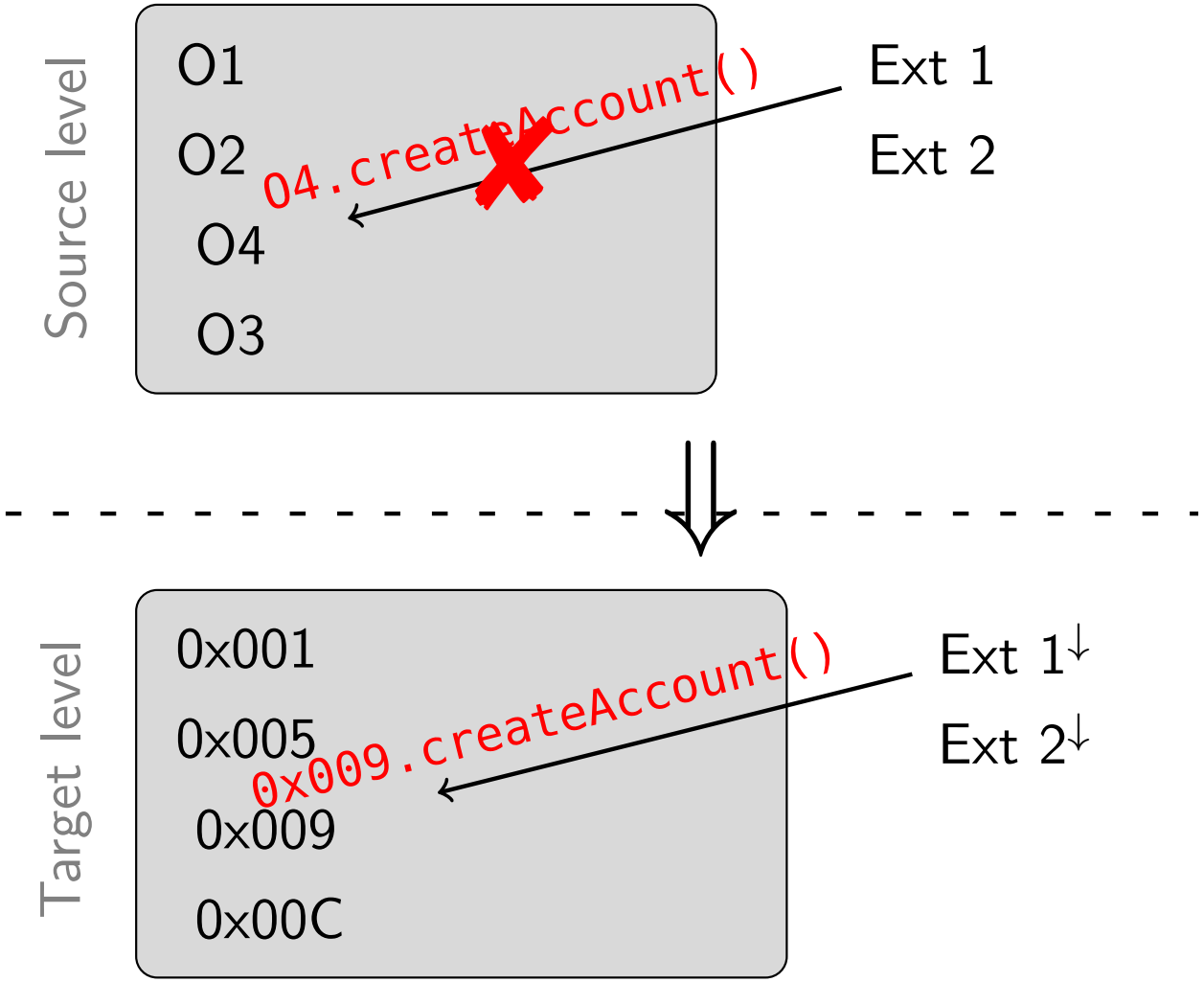
Dynamic Memory Allocation



Dynamic Memory Allocation

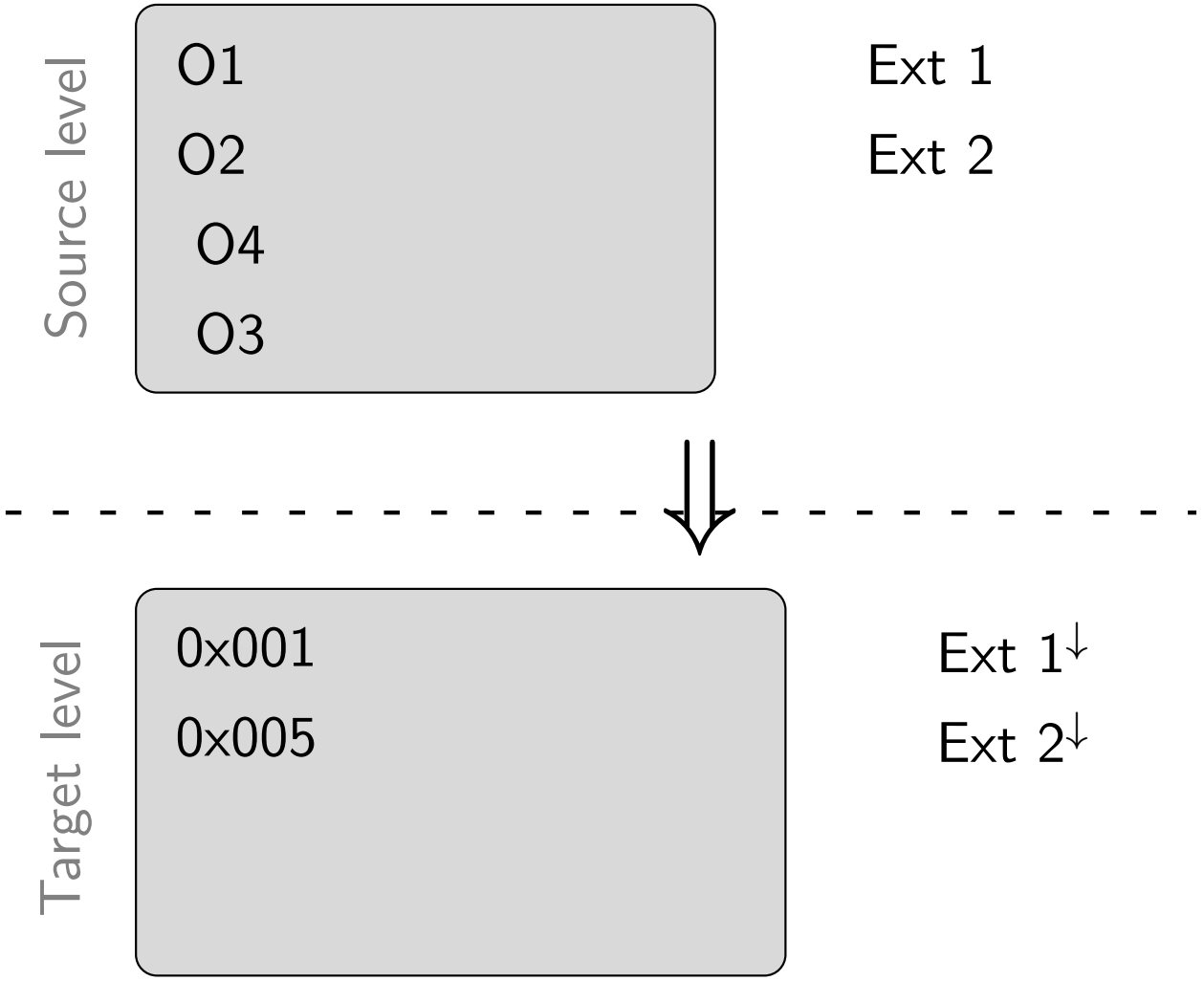


Dynamic Memory Allocation



- Object id guessing

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers

Dynamic Memory Allocation

Source level

O1
O2
O4
O3

Ext 1
Ext 2

$\Downarrow$

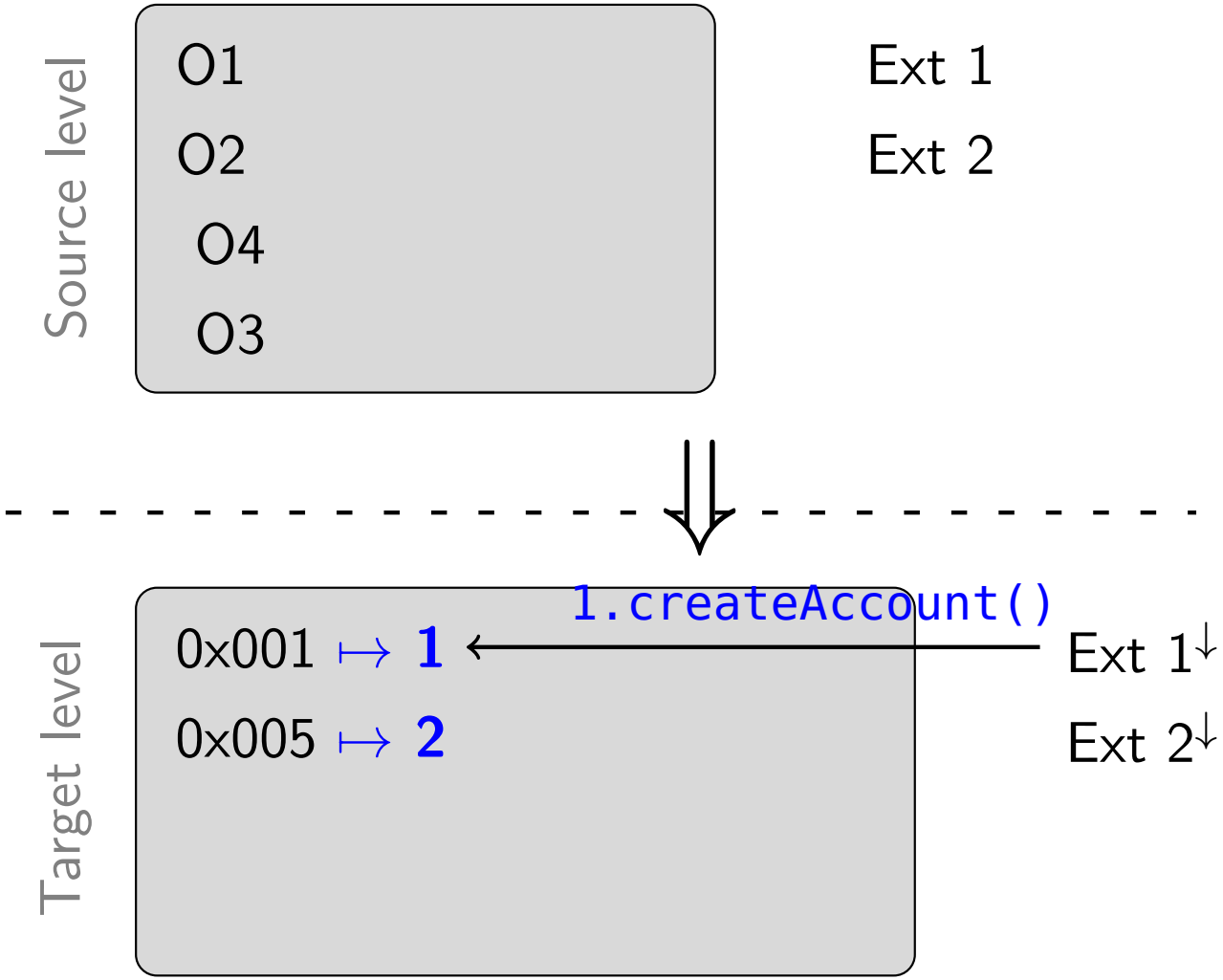
Target level

0x001 $\mapsto$ **1**
0x005 $\mapsto$ **2**

Ext $1^{\downarrow}$
Ext $2^{\downarrow}$

- Object id guessing
- map Oid to natural numbers

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers

Dynamic Memory Allocation

Source level

O1
O2
O4
O3

Ext 1
Ext 2

$\Downarrow$

Target level

0x001 $\mapsto$ **1**
0x005 $\mapsto$ **2**
0x009
0x00C

Ext $1^{\downarrow}$
Ext $2^{\downarrow}$

- Object id guessing
- map Oid to natural numbers

Dynamic Memory Allocation

Source level

O1
O2
O4
O3

Ext 1
Ext 2

$\Downarrow$

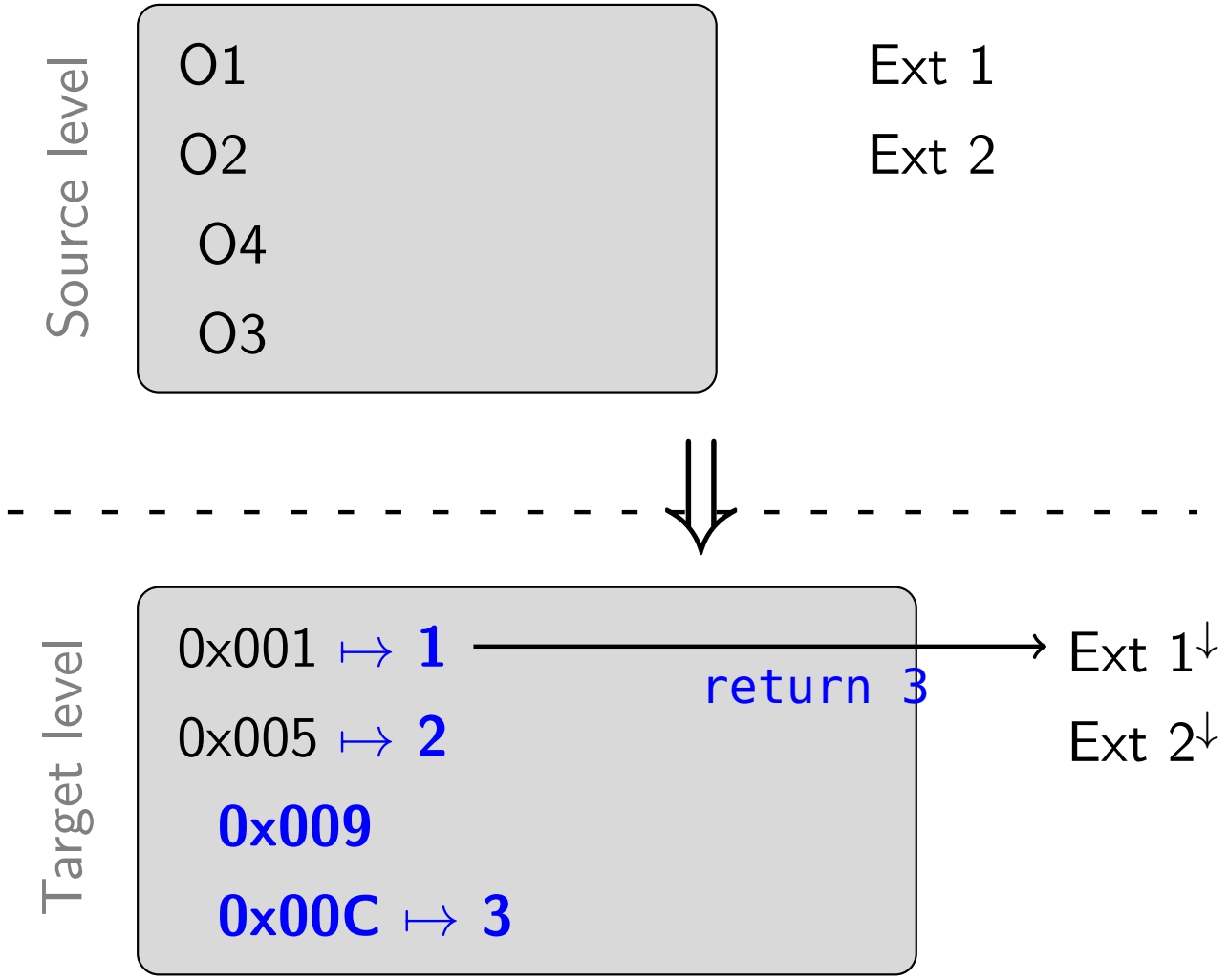
Target level

0x001 $\mapsto$ **1**
0x005 $\mapsto$ **2**
0x009
0x00C $\mapsto$ **3**

Ext $1^{\downarrow}$
Ext $2^{\downarrow}$

- Object id guessing
- map Oid to natural numbers
- add Oid to map

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers
- add Oid to map

Dynamic Memory Allocation

Source level

O1
O2
O4
O3

Ext 1
Ext 2

$\Downarrow$

Target level

0x001 $\mapsto$ **1**
0x005 $\mapsto$ **2**
0x009
0x00C $\mapsto$ **3**

Ext $1^{\downarrow}$
Ext $2^{\downarrow}$

- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received

Dynamic Memory Allocation

Source level

O1 : Account
O2 : Pair
O4
O3

Ext 1
Ext 2

$\Downarrow$

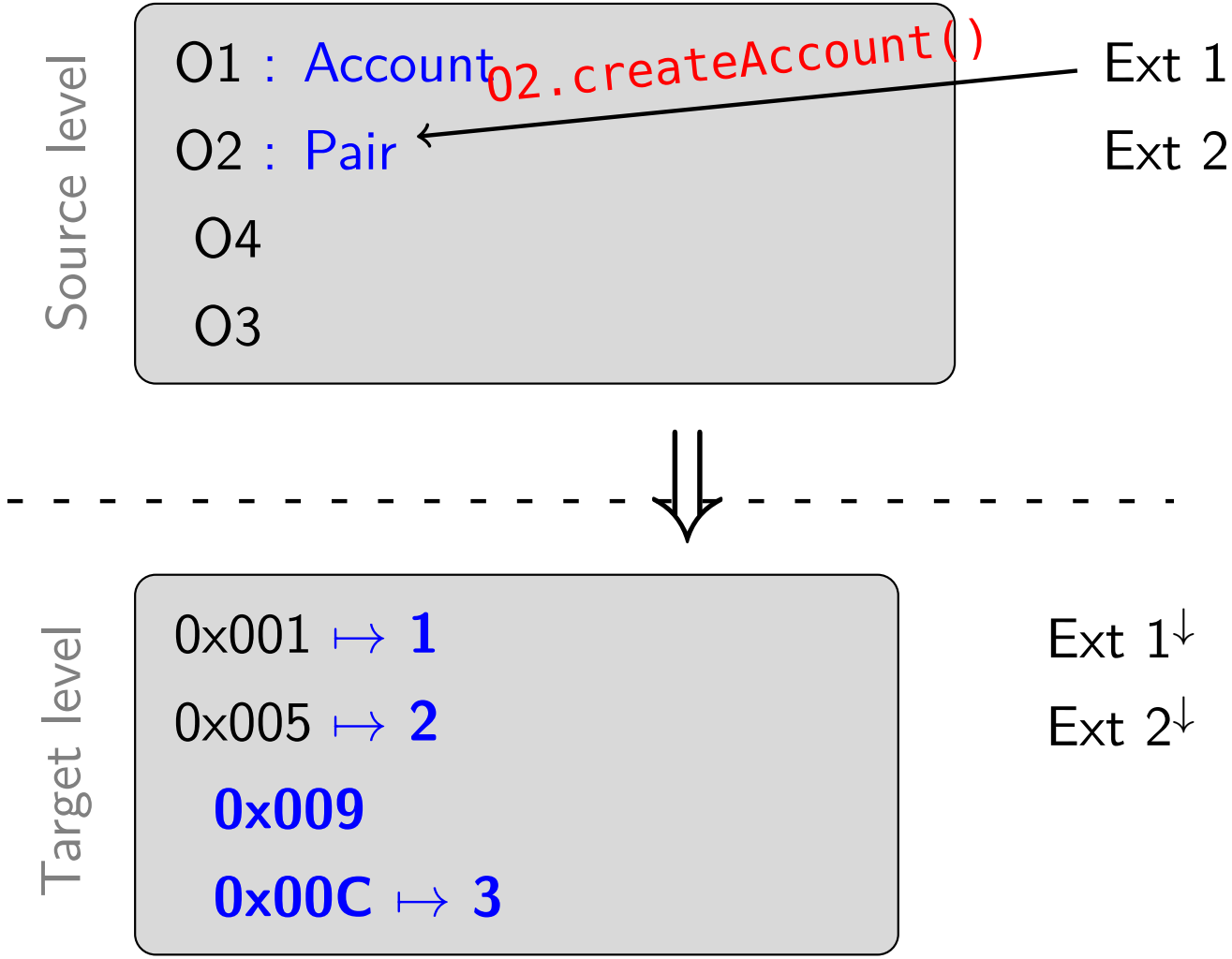
Target level

0x001 $\mapsto$ **1**
0x005 $\mapsto$ **2**
0x009
0x00C $\mapsto$ **3**

Ext $1^{\downarrow}$
Ext $2^{\downarrow}$

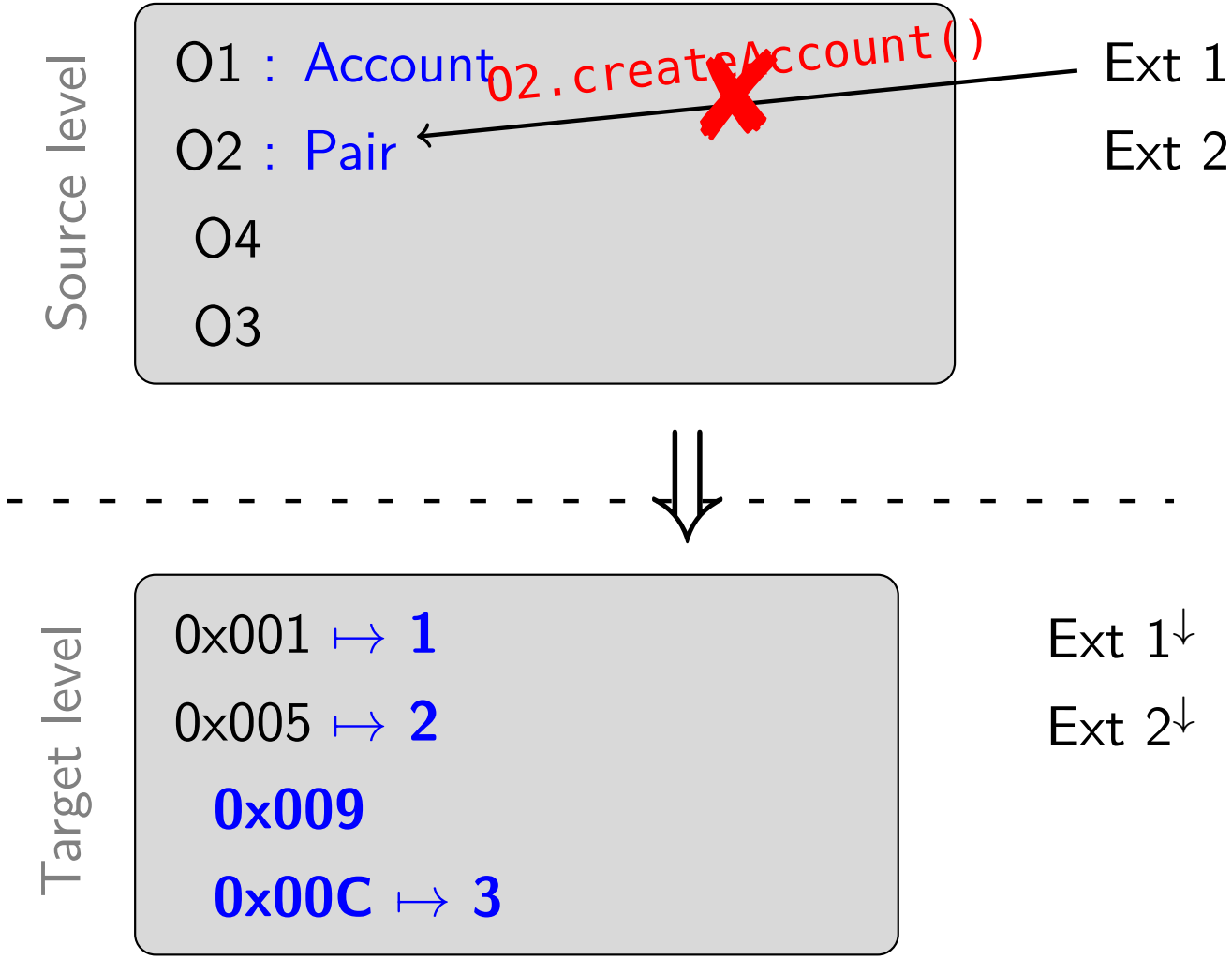
- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received

Dynamic Memory Allocation



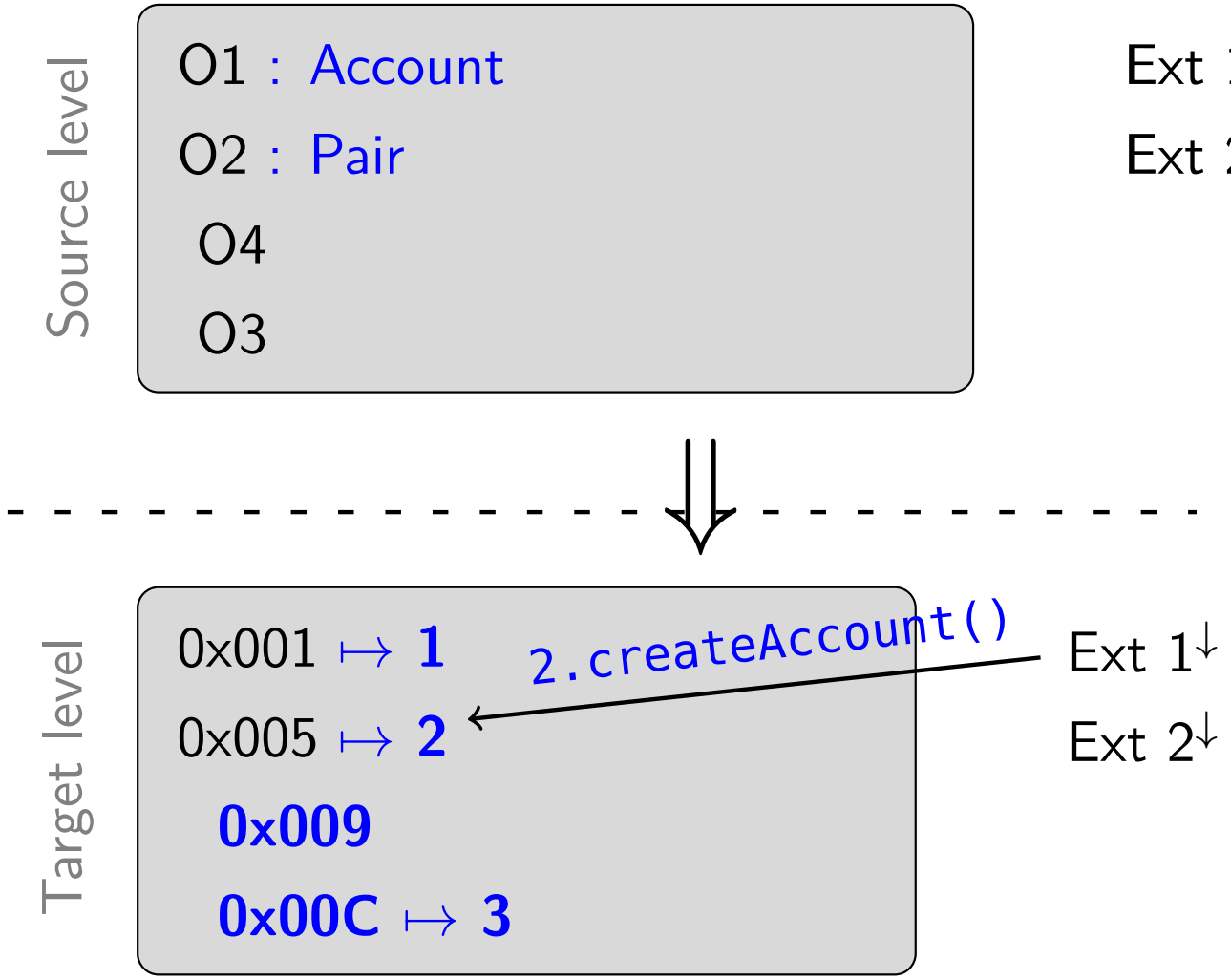
- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received

Dynamic Memory Allocation

Source level

O1 : Account
O2 : Pair
O4
O3

Ext 1
Ext 2

$\Downarrow$

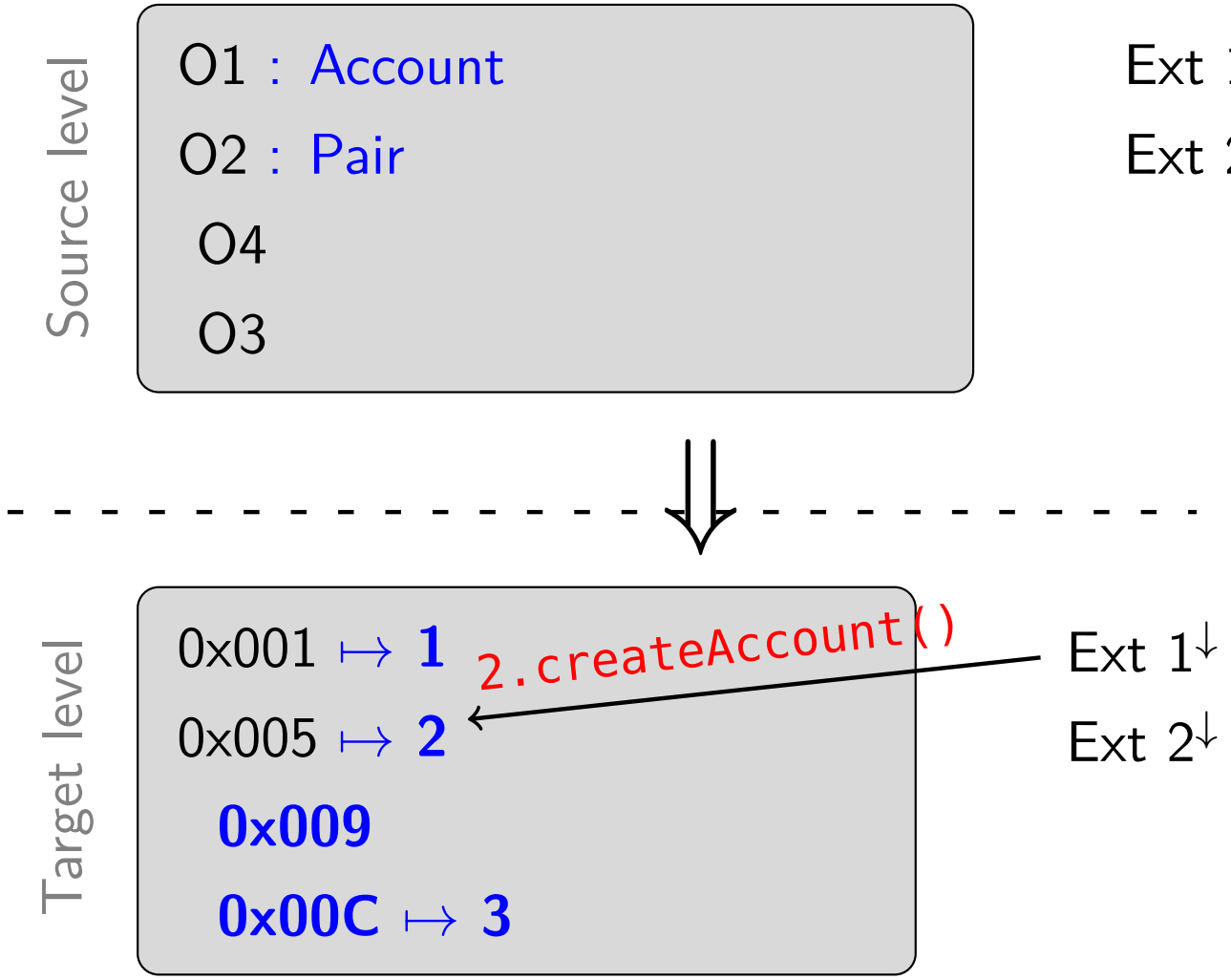
Target level

0x001 $\mapsto$ **1**
0x005 $\mapsto$ **2**
0x009
0x00C $\mapsto$ **3**

Ext $1^{\downarrow}$
Ext $2^{\downarrow}$

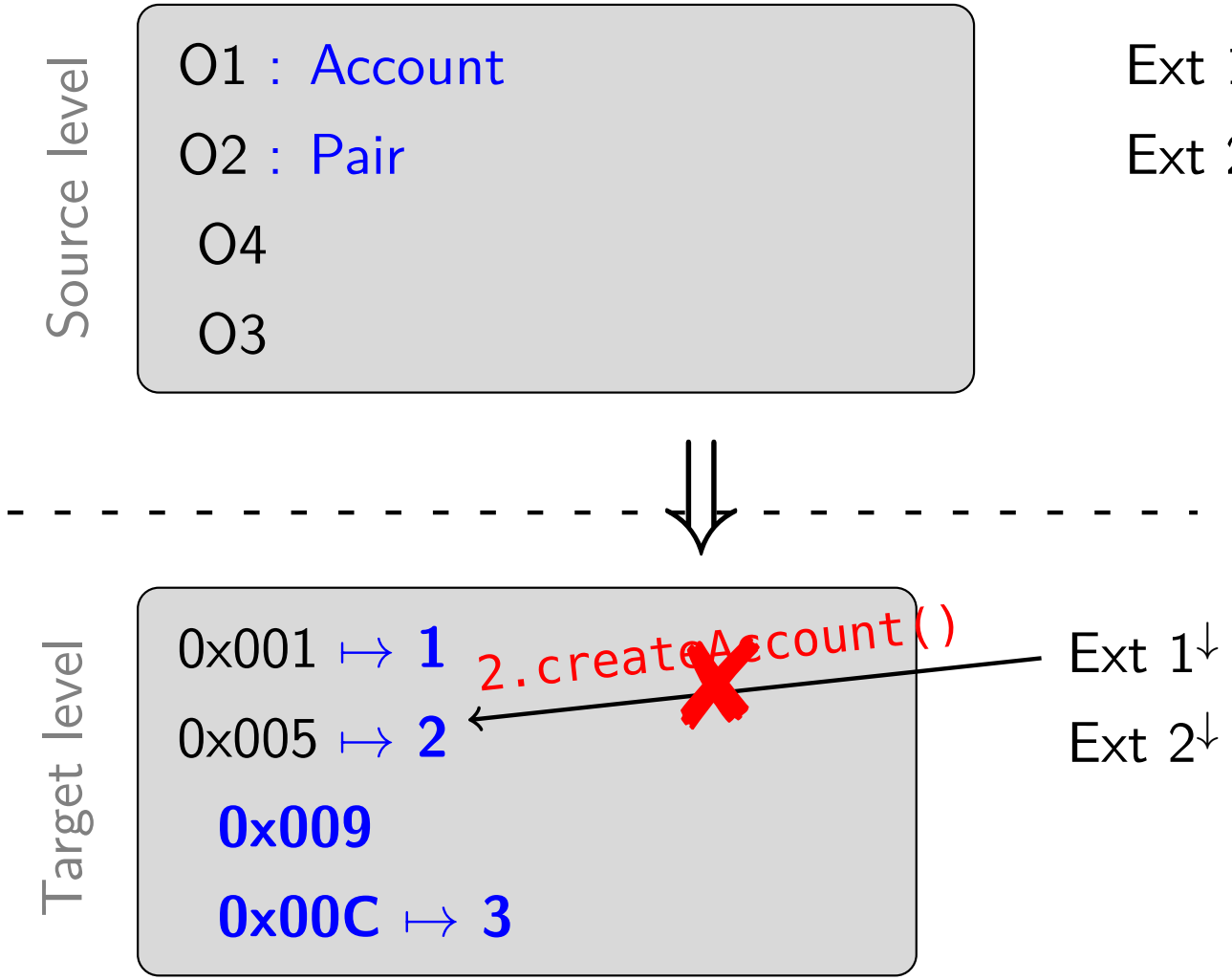
- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received
- dynamic typecheck for: current object

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received
- dynamic typecheck for: current object arguments

Dynamic Memory Allocation



- Object id guessing
- map Oid to natural numbers
- add Oid to map
- lookup ($\mathbf{O}(1)$) when number is received
- dynamic typecheck for: current object arguments

Why is this secure?

Source level

```
class ...
 secret = 1
```

Target level

Why is this secure?

Source level

```
class ...
 secret = 1
```

- -

Target level

```
class ...
 secret = 1
```

Why is this secure?

Source level

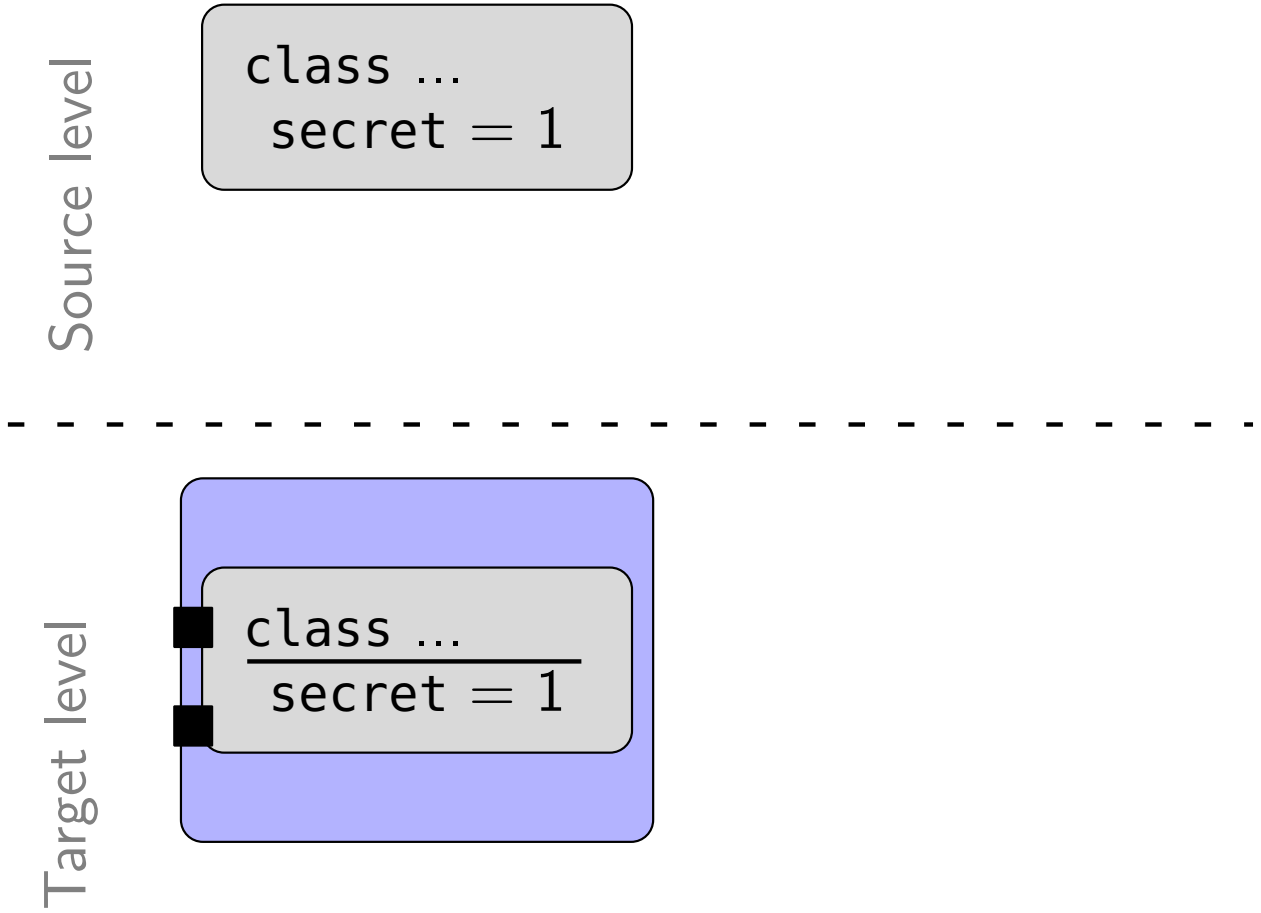
```
class ...
 secret = 1
```

Target level

```
class ...
 secret = 1
```

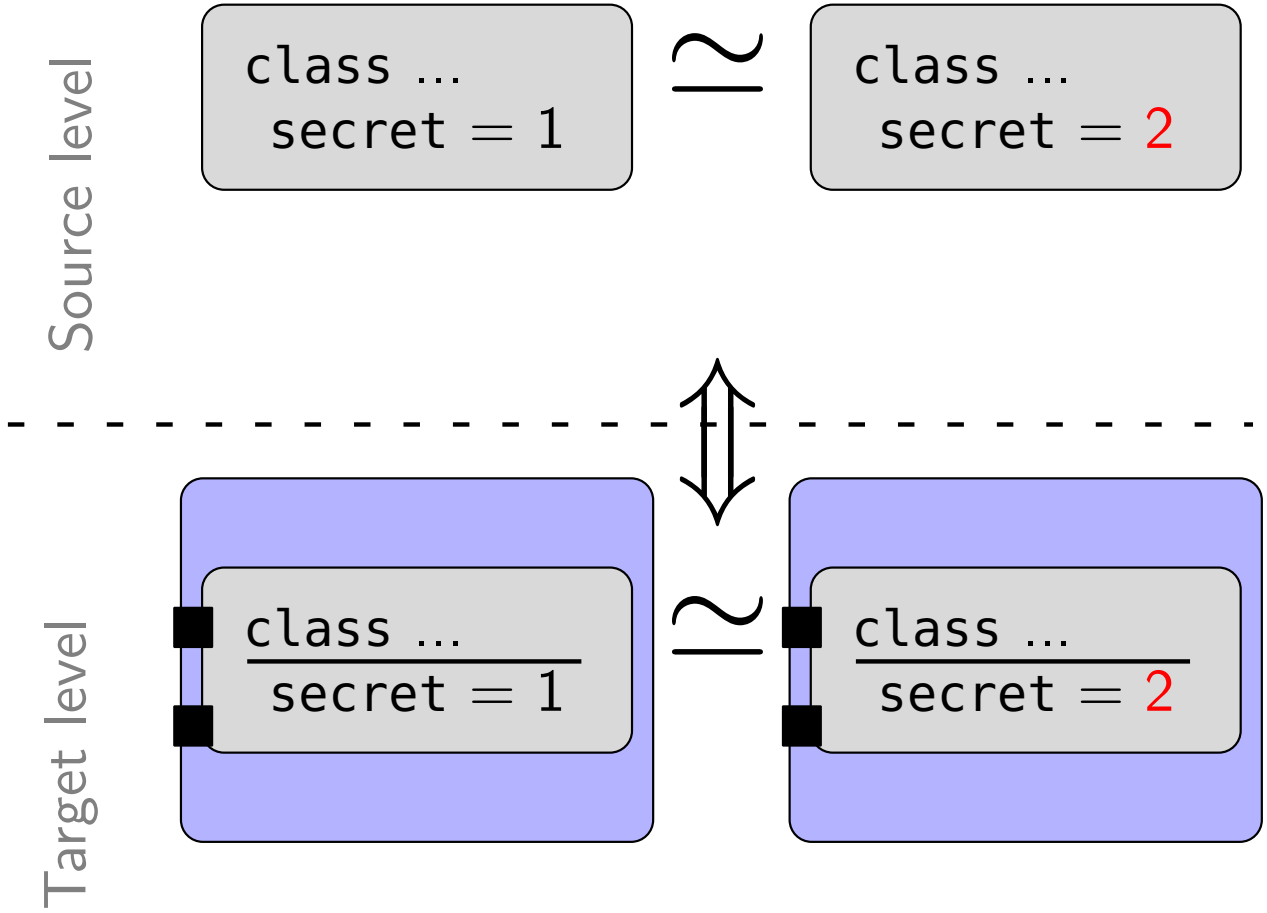
- we only add checks

Why is this secure?



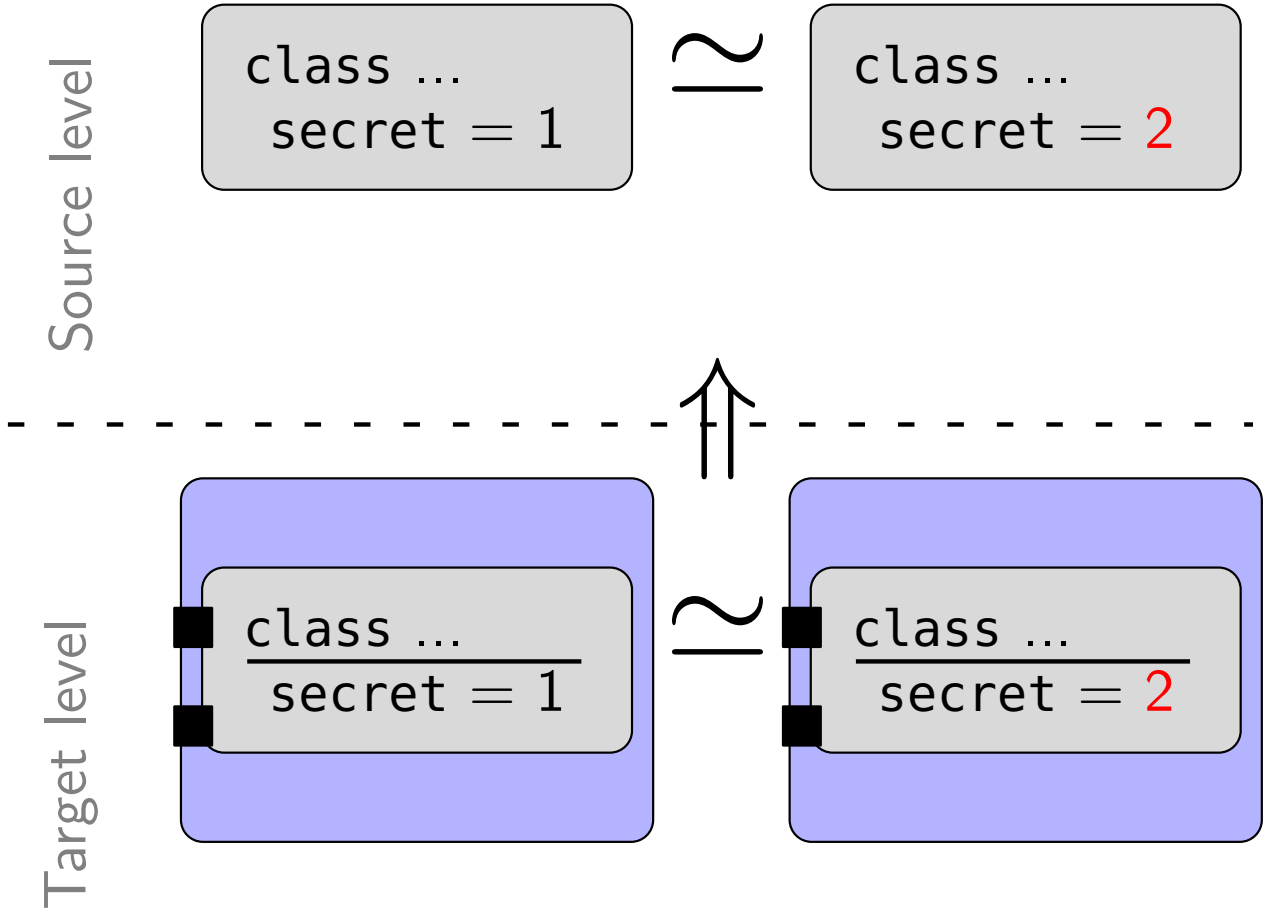
- we only add checks

Why is this secure?



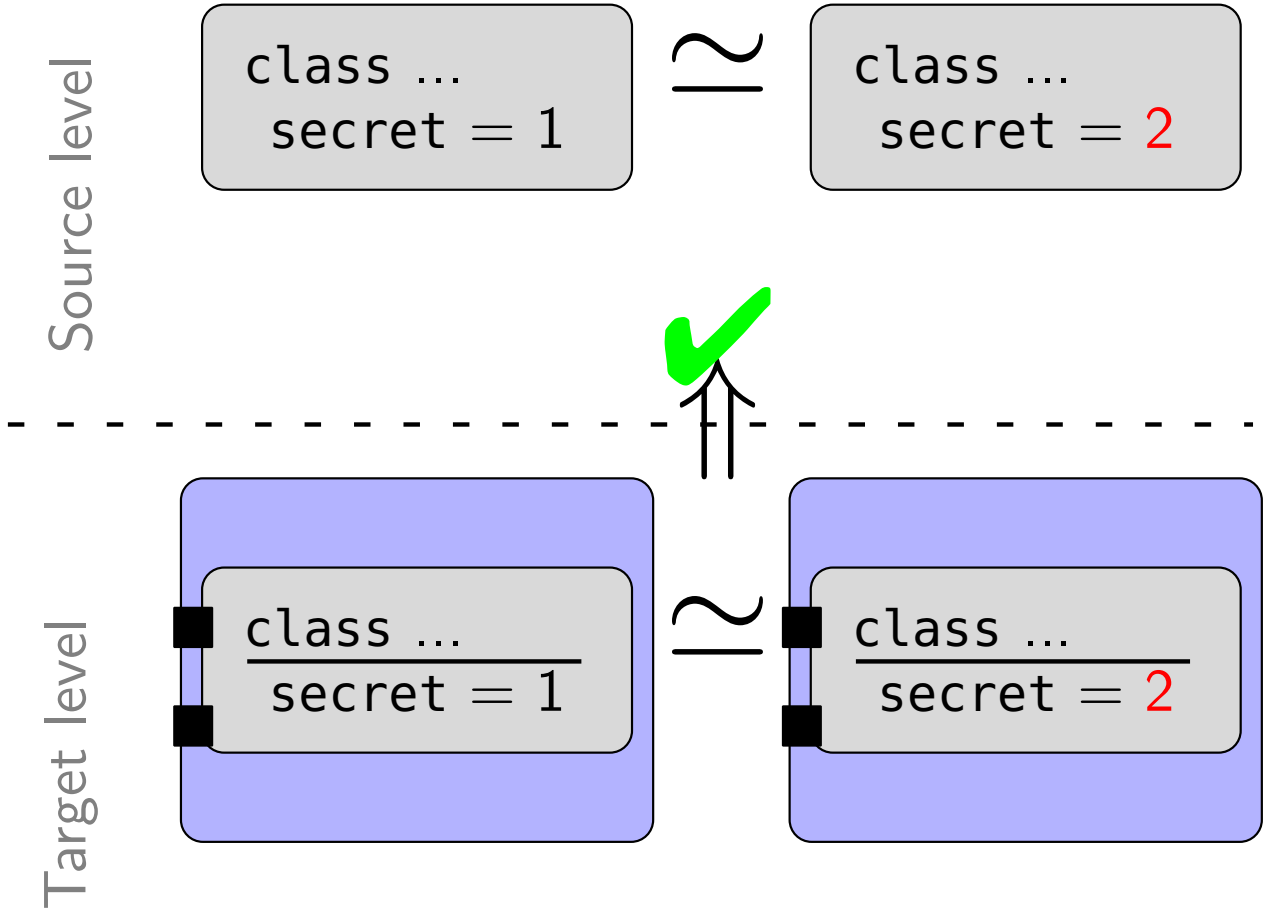
- we only add checks
- we need to prove full abstraction

Why is this secure?



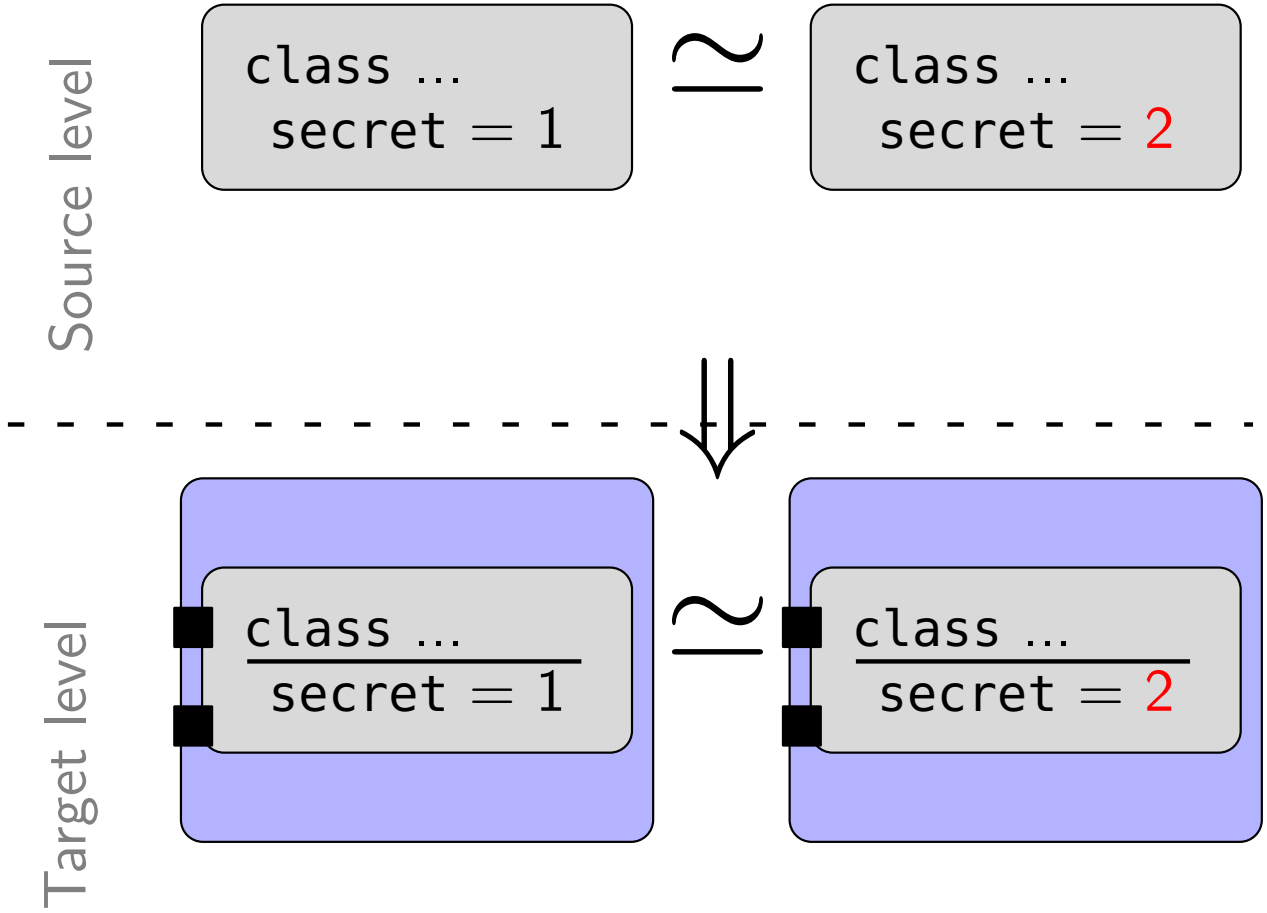
- we only add checks
- we need to prove full abstraction
- this is trivial

Why is this secure?



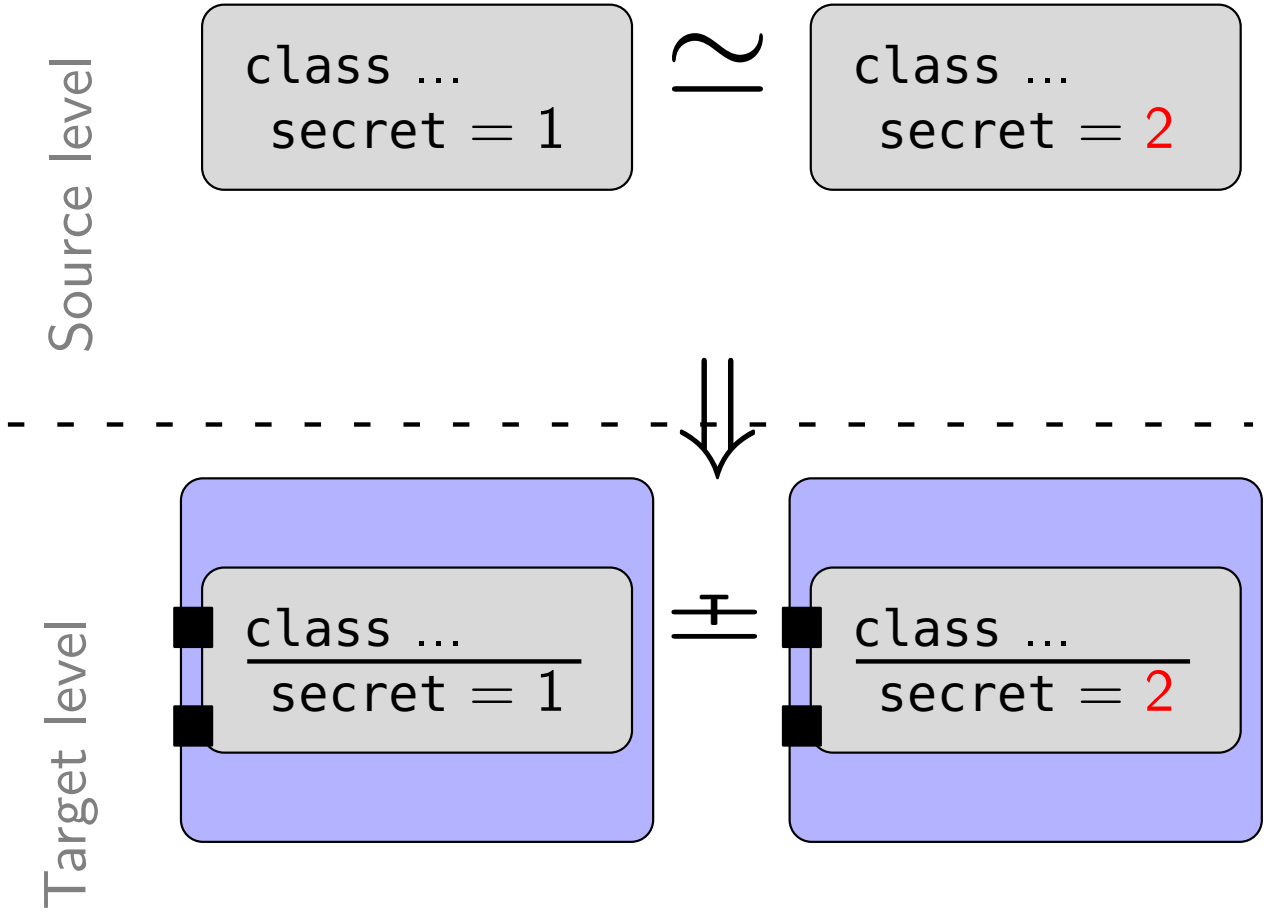
- we only add checks
- we need to prove full abstraction
- this is trivial

Why is this secure?



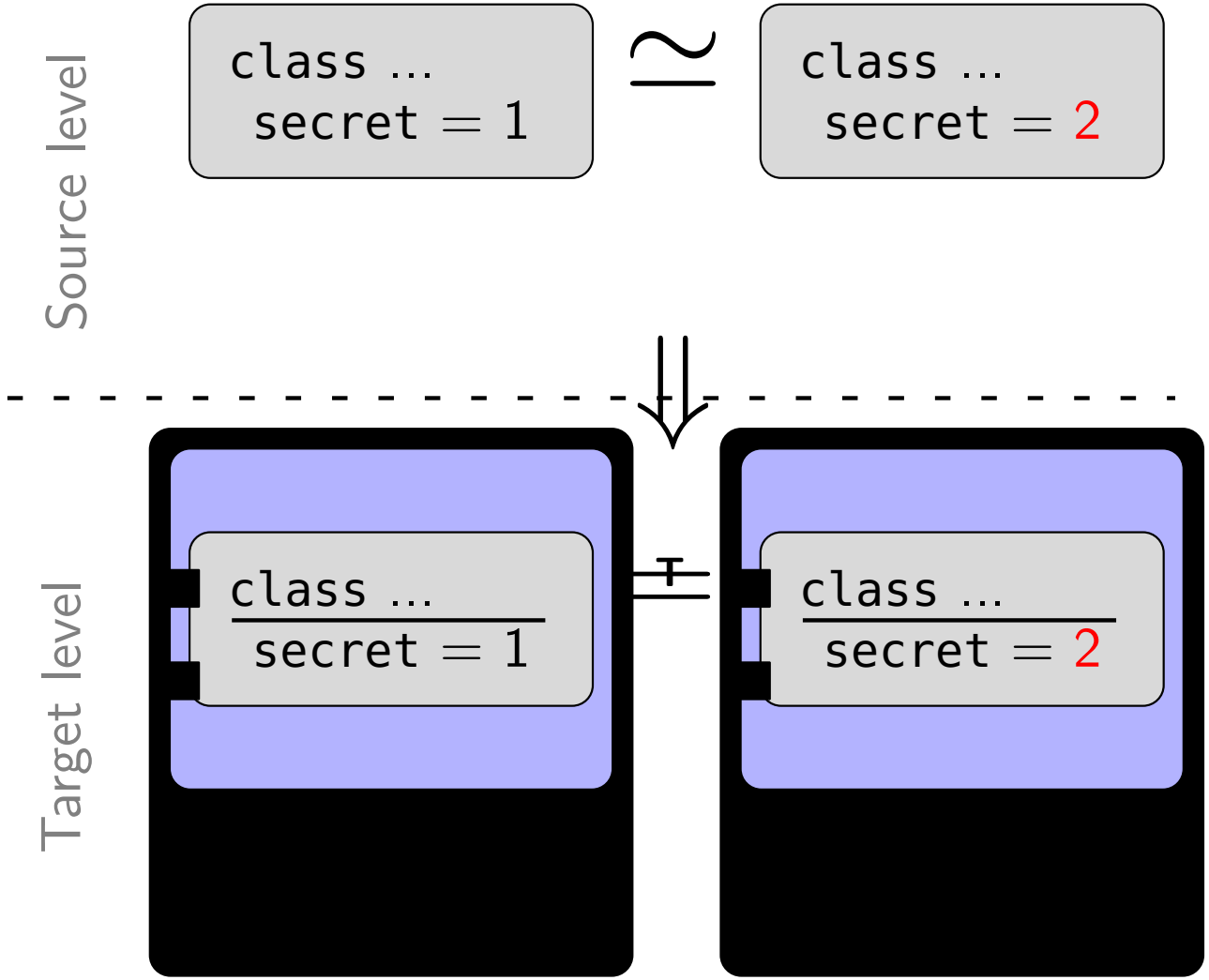
- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**

Why is this secure?



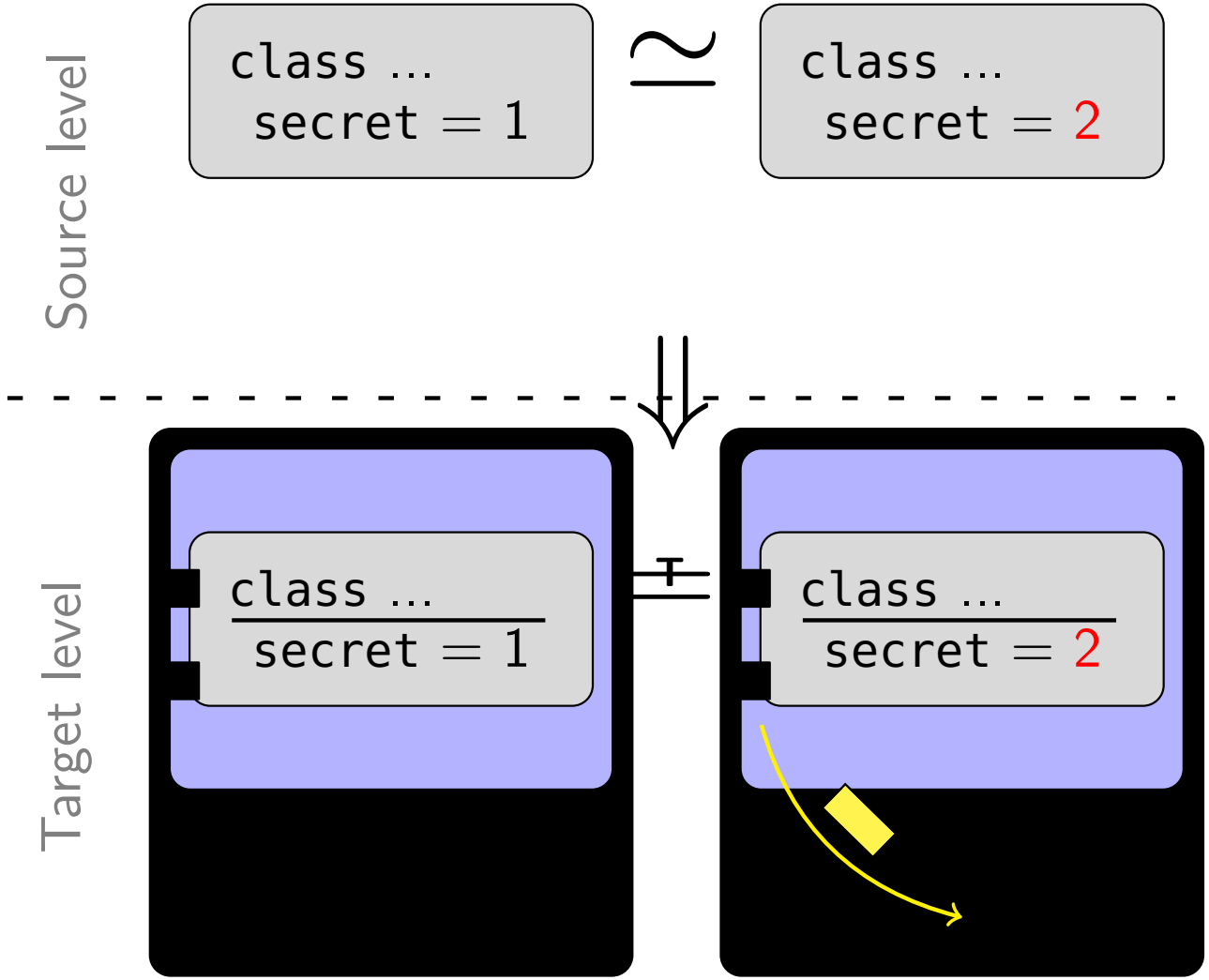
- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics

Why is this secure?



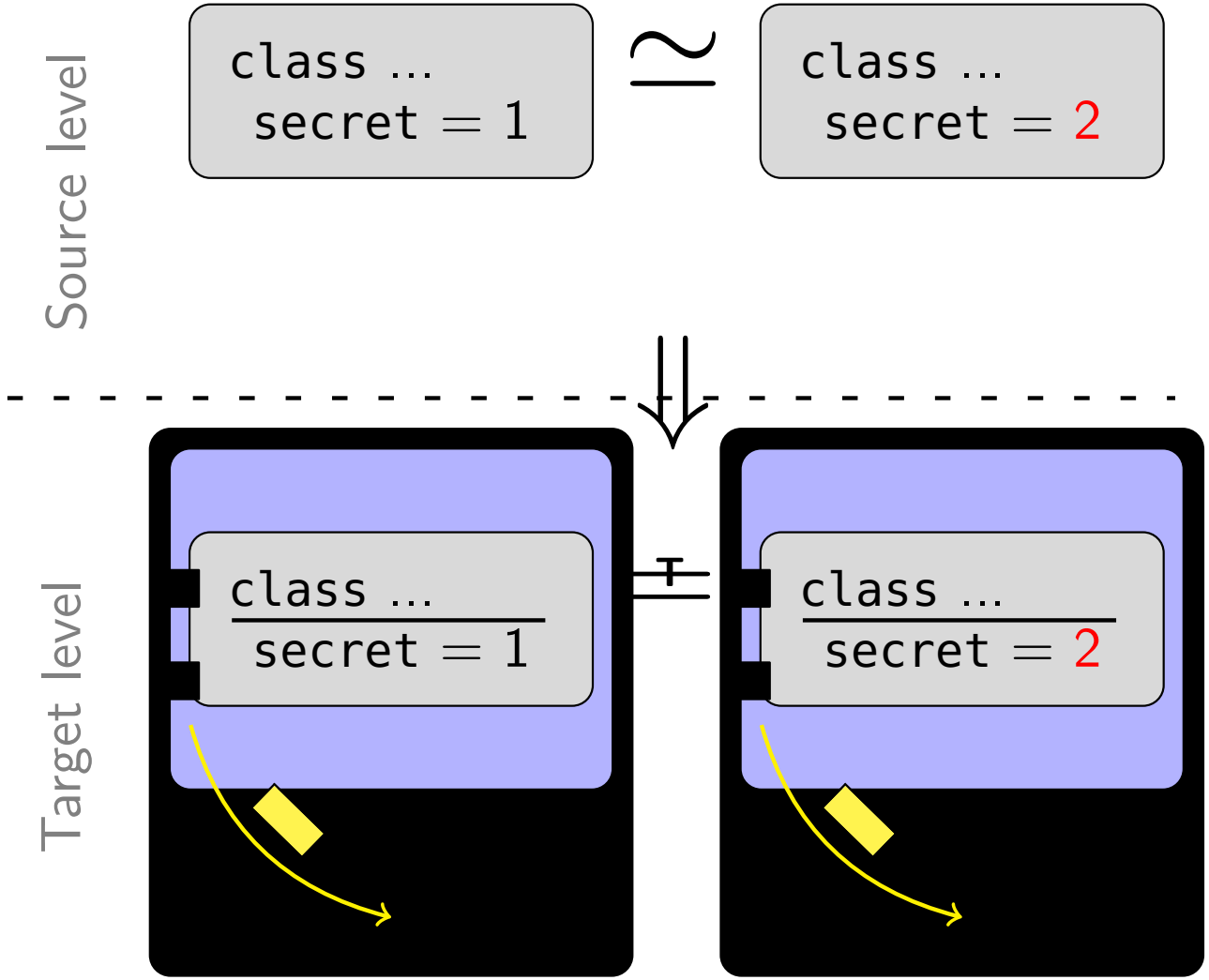
- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics

Why is this secure?



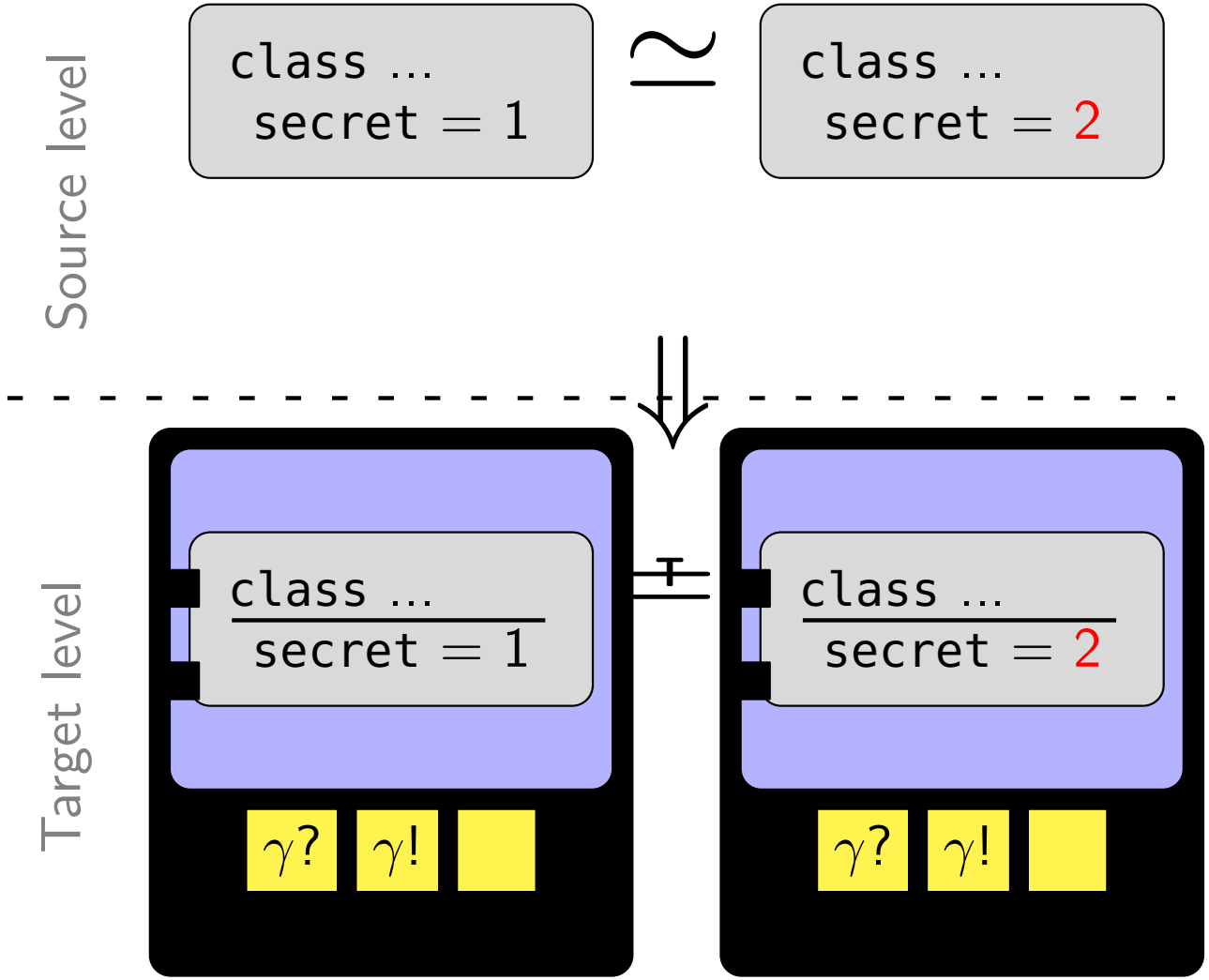
- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics

Why is this secure?



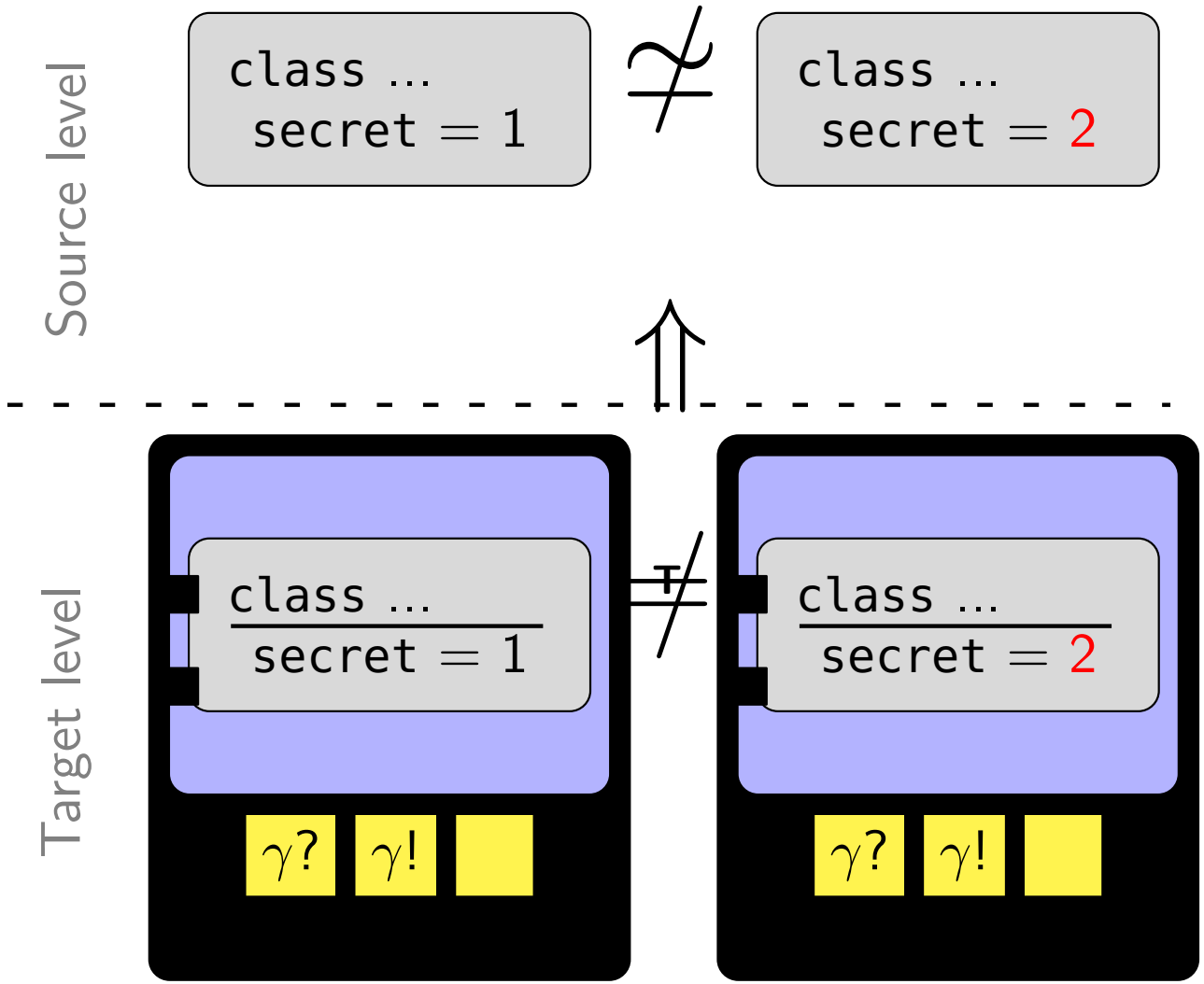
- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics

Why is this secure?



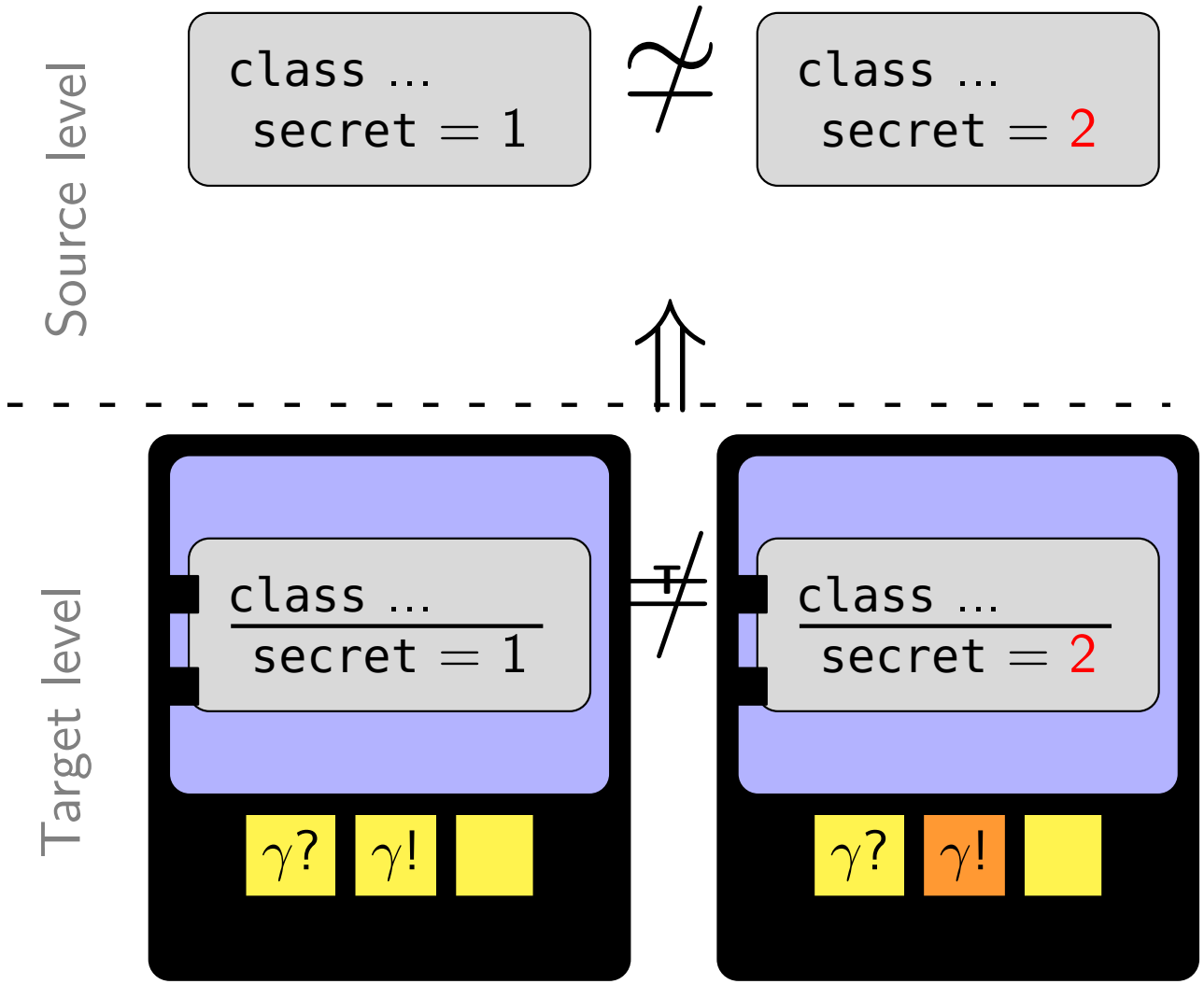
- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics
- and prove the contrapositive

Why is this secure?



- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics
- and prove the contrapositive

Why is this secure?

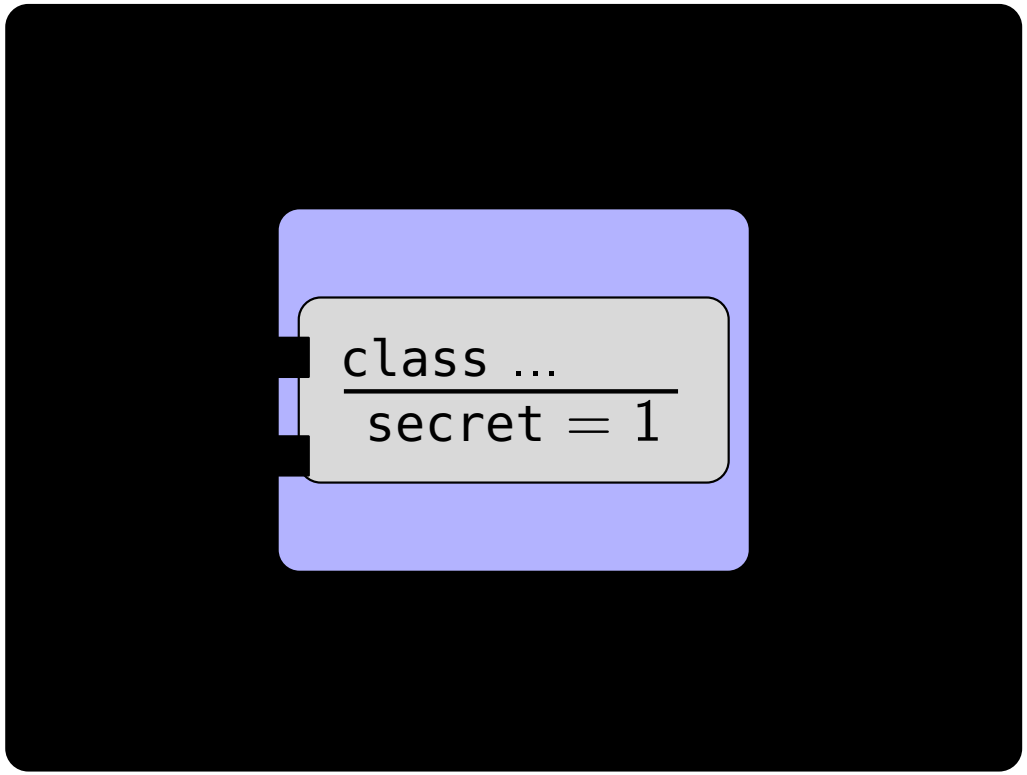


- we only add checks
- we need to prove full abstraction
- this is trivial
- this is **NOT**
- but we use trace semantics
- and prove the contrapositive

Why can we do this?

Source

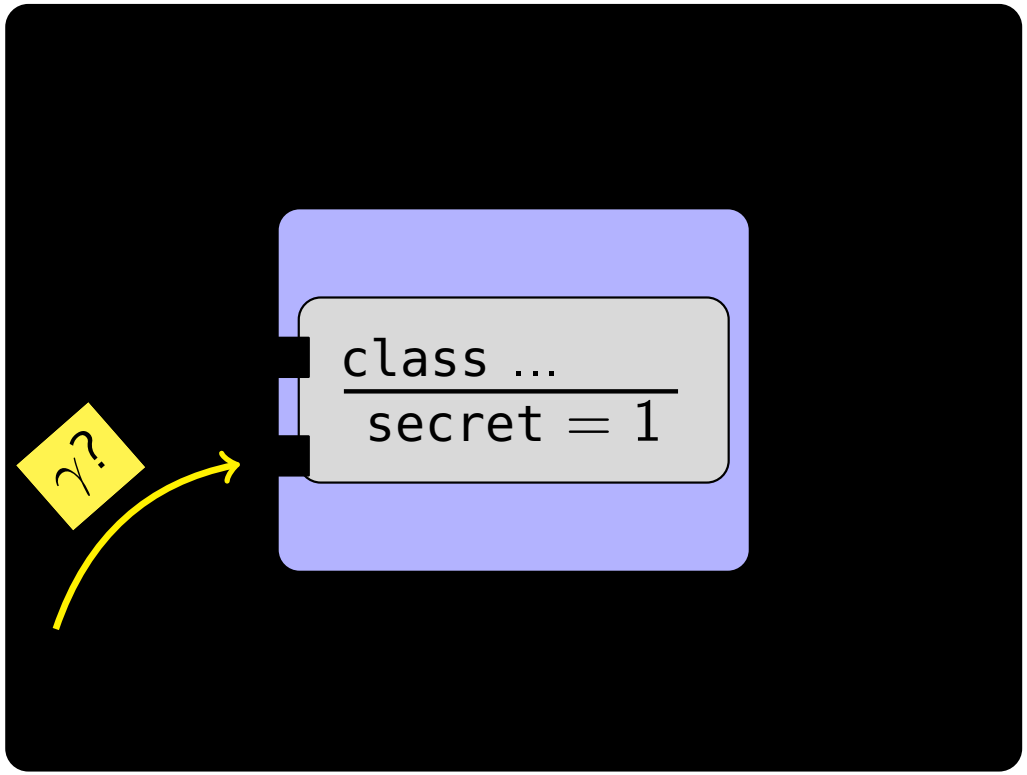
Target level



Why can we do this?

Source

Target level



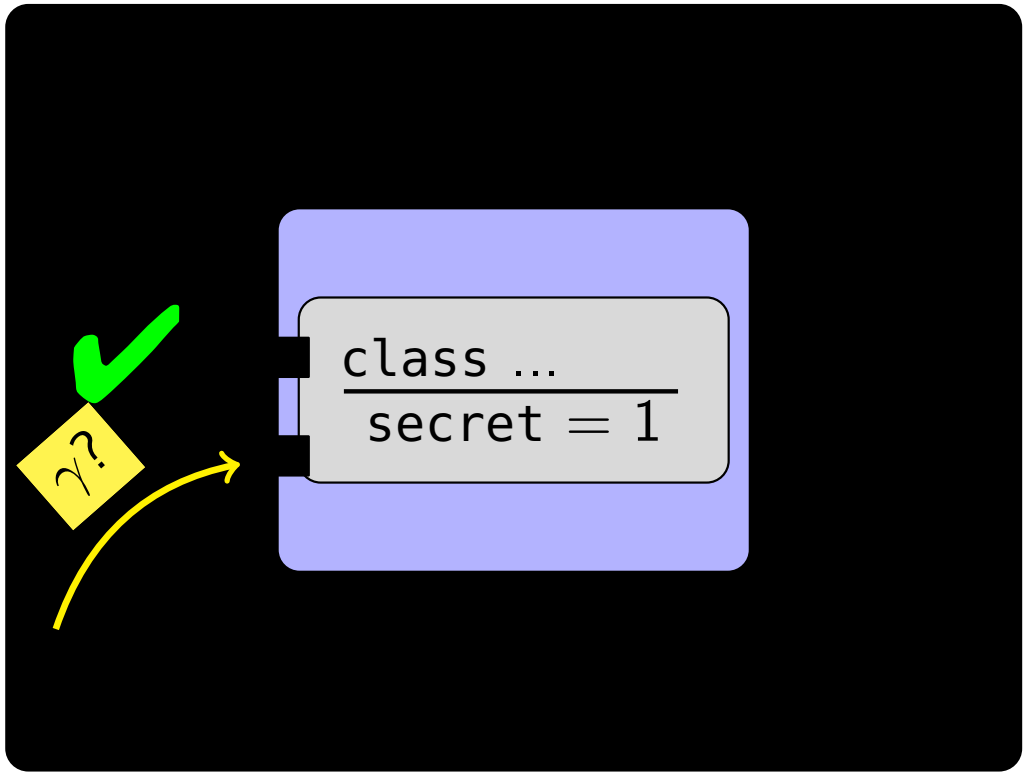
- analyse incoming traces

Why can we do this?

Source

Program $P \approx \gamma$?

Target level

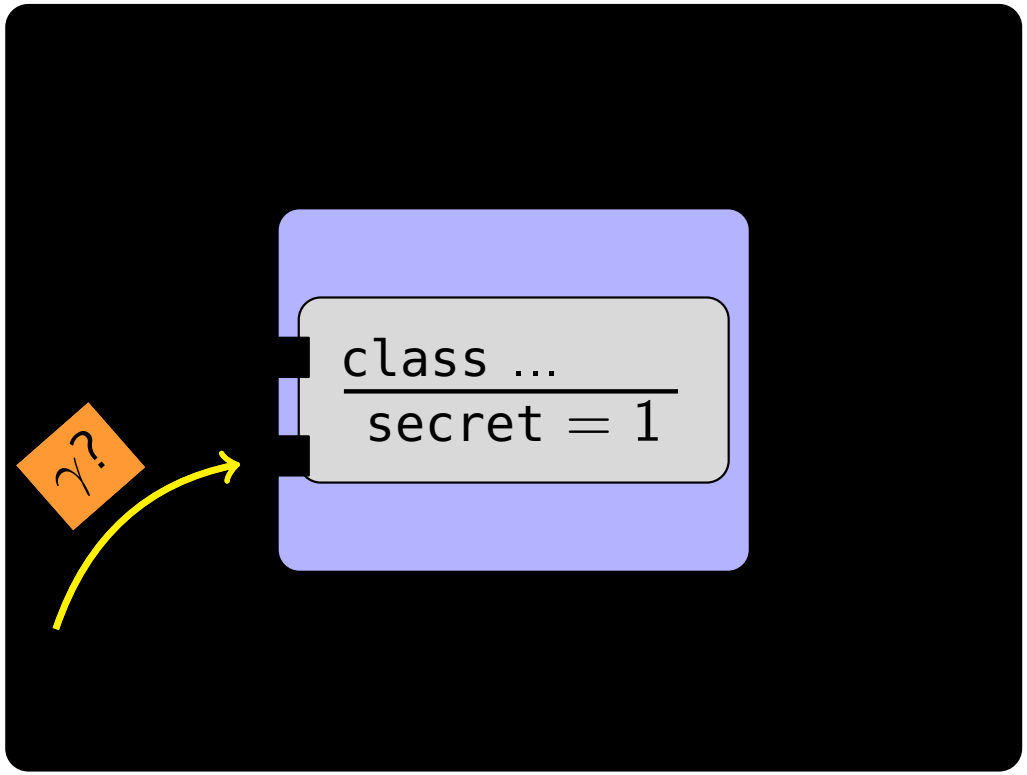


- analyse incoming traces
- let the *good* ones is

Why can we do this?

Source

Target level



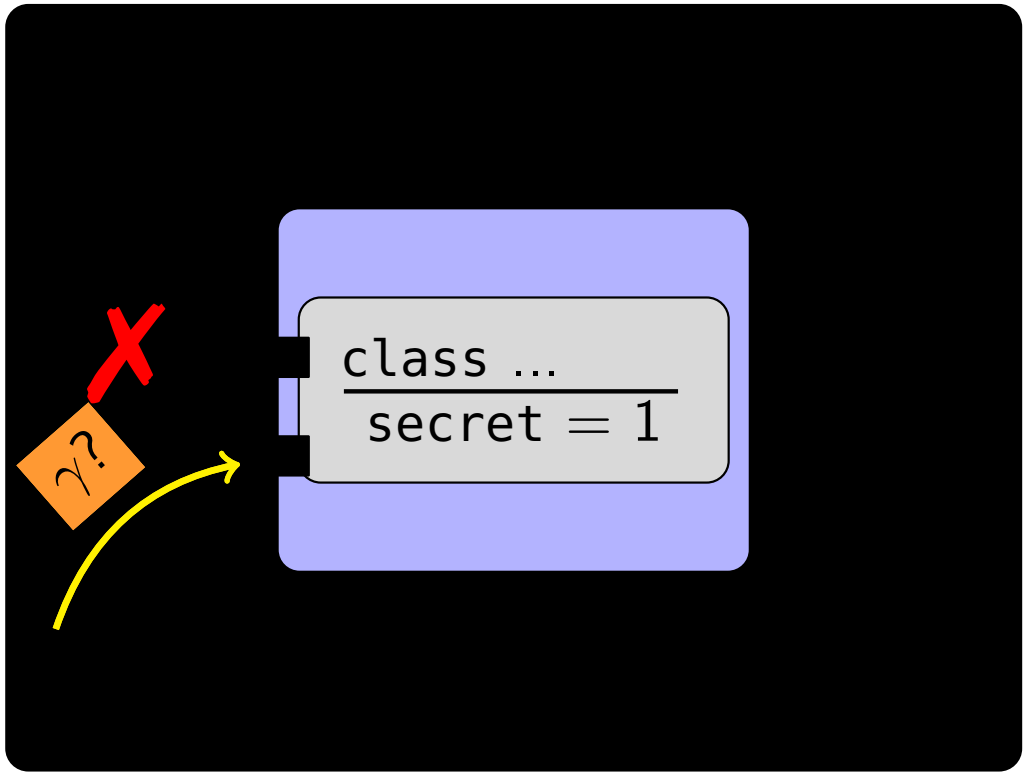
- analyse incoming traces
- let the *good* ones is

Why can we do this?

Source

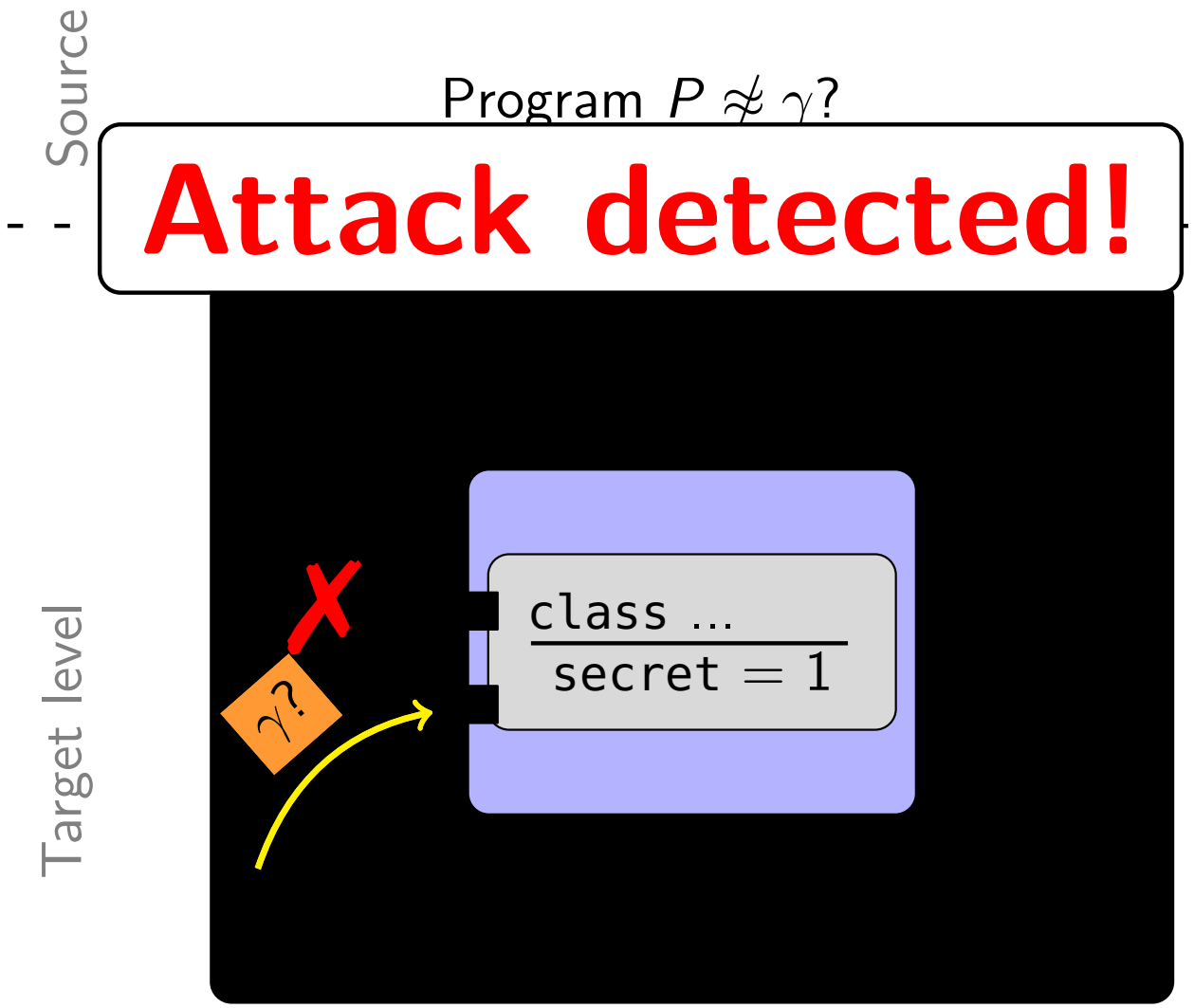
Program $P \not\approx \gamma$?

Target level



- analyse incoming traces
- let the *good* ones is
- block the *bad* ones

Why can we do this?

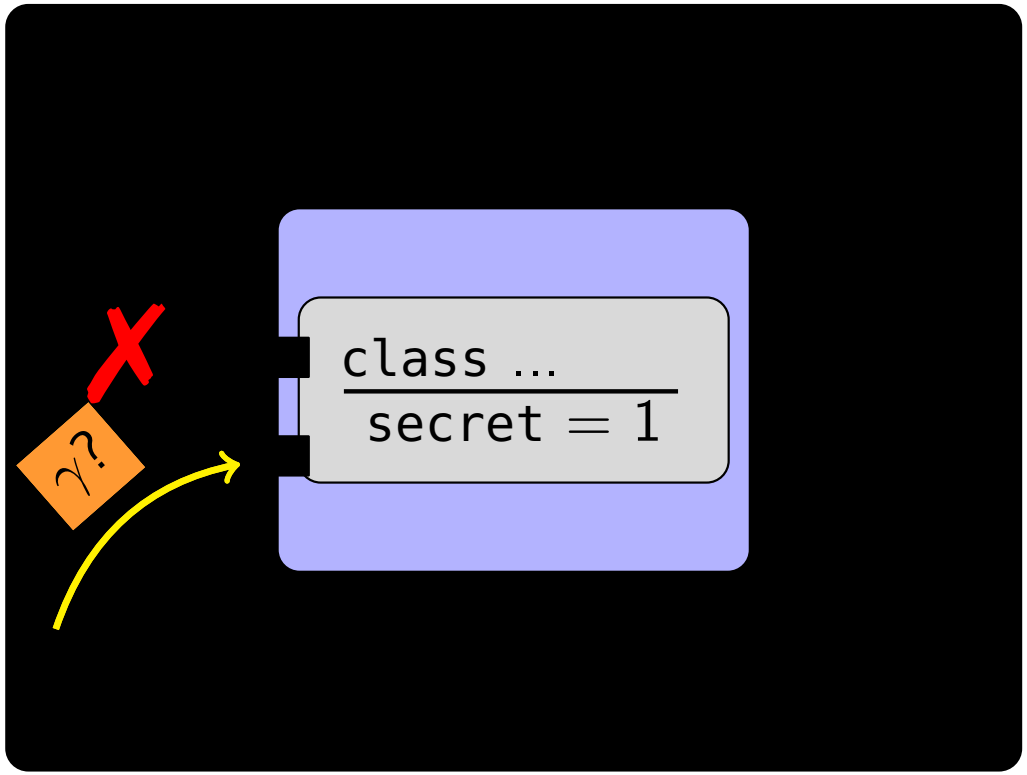


- analyse incoming traces
- let the *good* ones is
- block the *bad* ones

Why can we do this?

Source

Target level



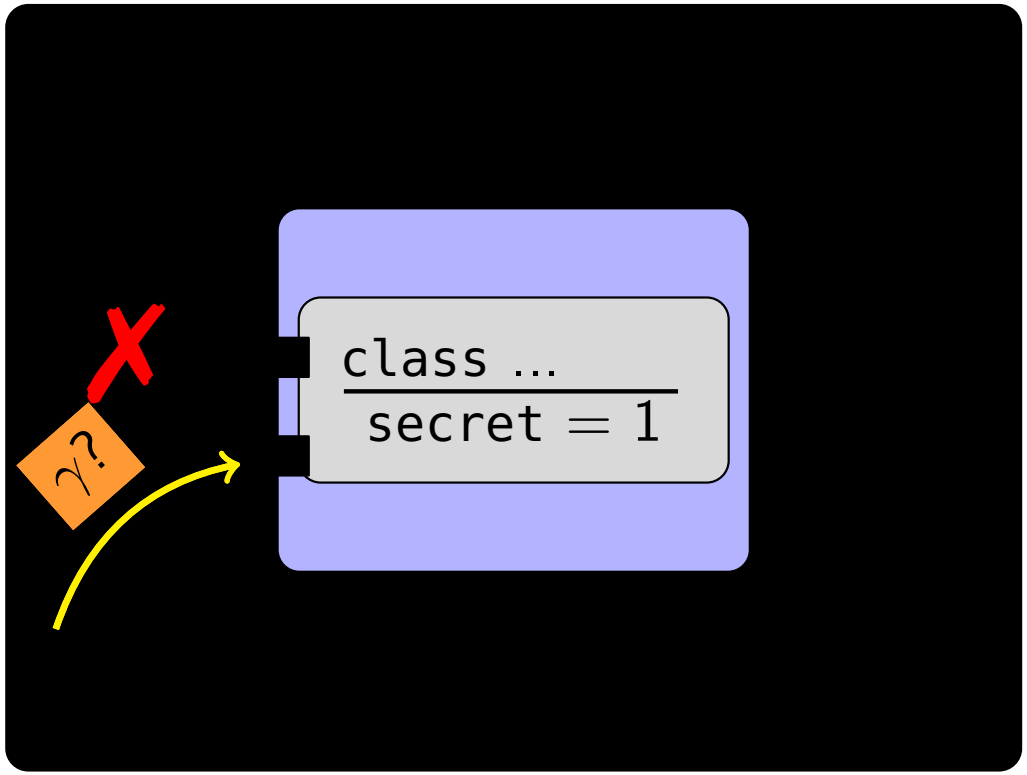
- analyse incoming traces
- let the *good* ones is
- block the *bad* ones

Why can we do this?

Source

Program $P \approx \gamma$?

Target level



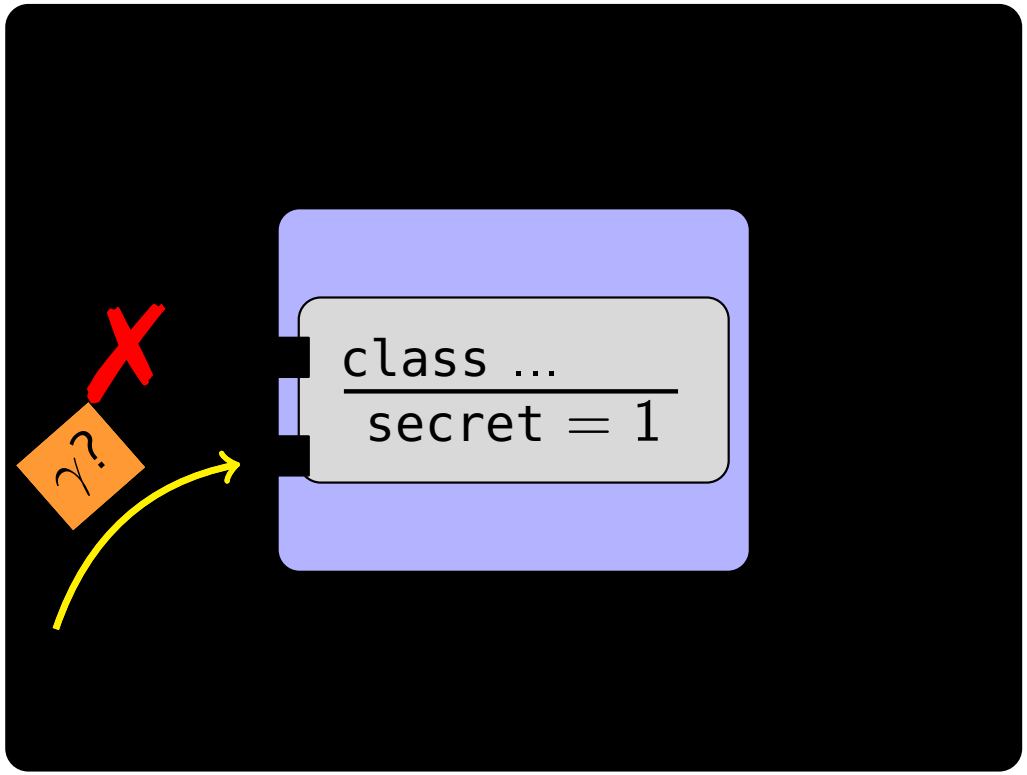
- analyse incoming traces
- let the *good* ones is
- block the *bad* ones

Why can we do this?

Source

Program $P \not\approx \gamma$?

Target level



- analyse incoming traces
- let the *good* ones is
- block the *bad* ones

Secure Compilation with Equivalence classes

Equivalence classes!!

Conclusion

- PMA is an interesting security architecture

Conclusion

- PMA is an interesting security architecture
- you can reason about it with simple traces

Conclusion

- PMA is an interesting security architecture
- you can reason about it with simple traces
- you can securely compile Java-like code to it

Conclusion

- PMA is an interesting security architecture
- you can reason about it with simple traces
- you can securely compile Java-like code to it
- you can link that code to other secure code and preserve security

Conclusion

Thank you!

- interesting security architecture
- you can reason about it with simple traces
- you can securely compile Java-like code to it
- you can link that code to other secure code and preserve security

Conclusion

Thank you!

- interesting security architecture
- you can reason about it with simple traces
- you can securely compile Java-like code to it
- you can link that code to other secure code and preserve security

Qs ?

Compiler compositionality

Labels of $\text{Traces}^{\mathsf{L}}_{\mathsf{A+I}}$

Labels	$\mathfrak{l} ::= \tau \mid \mathfrak{a}$
Observable actions	$\mathfrak{a} ::= \mathfrak{g}? \mid \mathfrak{d}! \mid \surd$
Actions	$\mathfrak{g} ::= \texttt{call}\ p(r; f) \mid \texttt{ret}\ p(r; f)$
Prefixable actions	$\mathfrak{d} ::= \mathfrak{g} \mid \mathfrak{o}(a, v).\mathfrak{d}$
Prefixes	$\mathfrak{o} ::= \texttt{read} \mid \texttt{write}$

Labels of Traces^{S}_{A+I}

Labels	$\lambda ::= \tau \mid \alpha$
Observable actions	$\alpha ::= \gamma? \mid \gamma! \mid \surd$
Actions	$\gamma ::= \texttt{call}\ \overline{v} \mid \texttt{ret}\ p\ v$

StripNI(·,)

$$
\begin{aligned}
\mathsf{StripNI}(\Theta, \mathfrak{g}) &= \mathfrak{g} \\
\mathsf{StripNI}(\Theta, \mathtt{write}(a, v)\mathfrak{d}) &= \mathtt{write}(a, v)\mathfrak{d}' \\
&\quad \text{if } \mathsf{StripNI}(\Theta, \mathfrak{d}) = \mathfrak{d}' \\
\mathsf{StripNI}(\Theta, \mathsf{read}(a, v)\mathfrak{d}) &= \mathfrak{d}' \\
&\quad \text{if } \mathsf{StripNI}(\Theta, \mathfrak{d}) = \mathfrak{d}' \text{ and } \mathsf{NI}(\Theta, a) \\
\mathsf{StripNI}(\Theta, \mathsf{read}(a, v)\mathfrak{d}) &= \mathsf{read}(a, v)\mathfrak{d}' \\
&\quad \text{if } \mathsf{StripNI}(\Theta, \mathfrak{d}) = \mathfrak{d}' \text{ and } \neg\mathsf{NI}(\Theta, a)
\end{aligned}
$$

NI(·)

$$\begin{aligned}\mathsf{NI}(\Theta, a) \triangleq \forall v, w. \quad & \Theta \overset{\mathfrak{a}_1}{\Longrightarrow} \Theta' \text{ and } \Theta \overset{\mathfrak{a}_2}{\Longrightarrow} \Theta'' \\ & \text{and } \mathfrak{a}_1 = \overline{\mathfrak{o}(a', v')}\mathsf{read}(a, v)\mathfrak{d}! \\ & \text{and } \mathfrak{a}_2 = \overline{\mathfrak{o}(a', v')}\mathsf{read}(a, w)\mathfrak{d}! \\ & \text{and } \mathsf{Tr\text{-}state}(\Theta') = \mathsf{Tr\text{-}state}(\Theta'')\end{aligned}$$

Definitions

Definition (Contextual equivalence for A+I)

Definition (Fully abstract trace semantics for A+I)

Given that $\mathsf{Traces}^{\mathsf{L}}_{\mathsf{A+I}}(P) = \mathsf{Tr\text{-}state}(\Theta_0(P))$,
$P_1 \simeq^{\mathsf{A+I}} P_2 \iff P_1 \stackrel{\mathsf{T}}{=}^{\mathsf{A+I}} P_2$

Definition (Fully abstract compilation)

For any two source-level components C_1 and C_2, we have:
$C_1 \simeq^{\mathsf{J+E}} C_2 \iff [\![C_1]\!]^{\mathsf{J+E}}_{\mathsf{A+I}} \simeq^{\mathsf{A+I}} [\![C_2]\!]^{\mathsf{J+E}}_{\mathsf{A+I}}$.

Randomisation and Program Equivalence for AIM

Definition (Contextual preorder for AIM)

$$P_1 \sqsubseteq^{\mathsf{AIM}}_{\sim\sigma} P_2 \triangleq \mathtt{Pr}(\forall \mathbb{P},\ O_1.\ \exists O_2.\ \ \mathbb{P}[P_1, O_1]\Uparrow \iff \mathbb{P}[P_2, O_2]\Uparrow) > \sigma.$$

Definition (Contextual equivalence for AIM)

$$P_1 \simeq^{\mathsf{AIM}}_{\sigma} P_2 \triangleq P_1 \sqsubseteq^{\mathsf{AIM}}_{\sim\sigma} P_2 \text{ and } P_2 \sqsubseteq^{\mathsf{AIM}}_{\sim\sigma} P_1.$$

Definition (Trace equivalence for AIM)

$$P_1 \overset{\mathsf{T}}{=}^{\mathsf{AIM}}_{\sigma} P_2 \triangleq \mathtt{Pr}(\mathsf{Traces}_{\mathsf{AIM}}(P_1, O_1) = \mathsf{Traces}_{\mathsf{AIM}}(P_2, O_2)) > \sigma.$$