Fully Abstract Trace Semantics for Low-level Isolation Mechanisms

Marco Patrignani [1] Dave Clarke [1,2]

[1]iMinds-DistriNet, Dept. Computer Science, KU Leuven, Belgium

[2]now at Dept. of Information Technology, Uppsala, Sweden

25 March 2014

Outline

Why is PMA interesting?

- provides strong encapsulation *at the lowest level of abstraction*

Why is PMA interesting?

- provides strong encapsulation *at the lowest level of abstraction*
- can be exploited to develop a secure compiler [PCP13]

Why is PMA interesting?

- provides strong encapsulation *at the lowest level of abstraction*
- can be exploited to develop a secure compiler [PCP13]
- can be exploited to enforce a control-flow safe execution of C code [APJ14]

What is a Protected Modules Architecture (PMA)

What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism

What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism
- implemented via Hypervisor, Hardware, Software

What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism
- implemented via Hypervisor, Hardware, Software
- several research prototypes: Fides [SP12], Sancus [NAD+13], Flicker [MPP+08], TrustVisor [MLQ+10], Smart [EFPT12]

What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism
- implemented via Hypervisor, Hardware, Software
- several research prototypes: Fides [SP12], Sancus [NAD+13], Flicker [MPP+08], TrustVisor [MLQ+10], Smart [EFPT12]
- industrial prototype too: Intel SGX [MAB+13]

What is a Protected Modules Architecture (PMA)

- assembly-level isolation mechanism
- implemented via Hypervisor, Hardware, Software
- several research prototypes: Fides [SP12], Sancus [NAD+13], Flicker [MPP+08], TrustVisor [MLQ+10], Smart [EFPT12]
- industrial prototype too: Intel SGX [MAB+13]

Let's see an example of PMA in action

PMA in action

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
   ⋮
0x0b52    movs r0 0x0b55
0x0b53    call 0x0002
0x0b54    movs r0 0x0001
0x0b55    ...
   ⋮
0xab00    jmp 0xb53
0xab01    ...
```

- memory space

PMA in action

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
   ⋮
0x0b52    movs r0 0x0b55
0x0b53    call 0x0002
0x0b54    movs r0 0x0001
0x0b55    ...
   ⋮
0xab00    jmp 0xb53
0xab01    ...
```

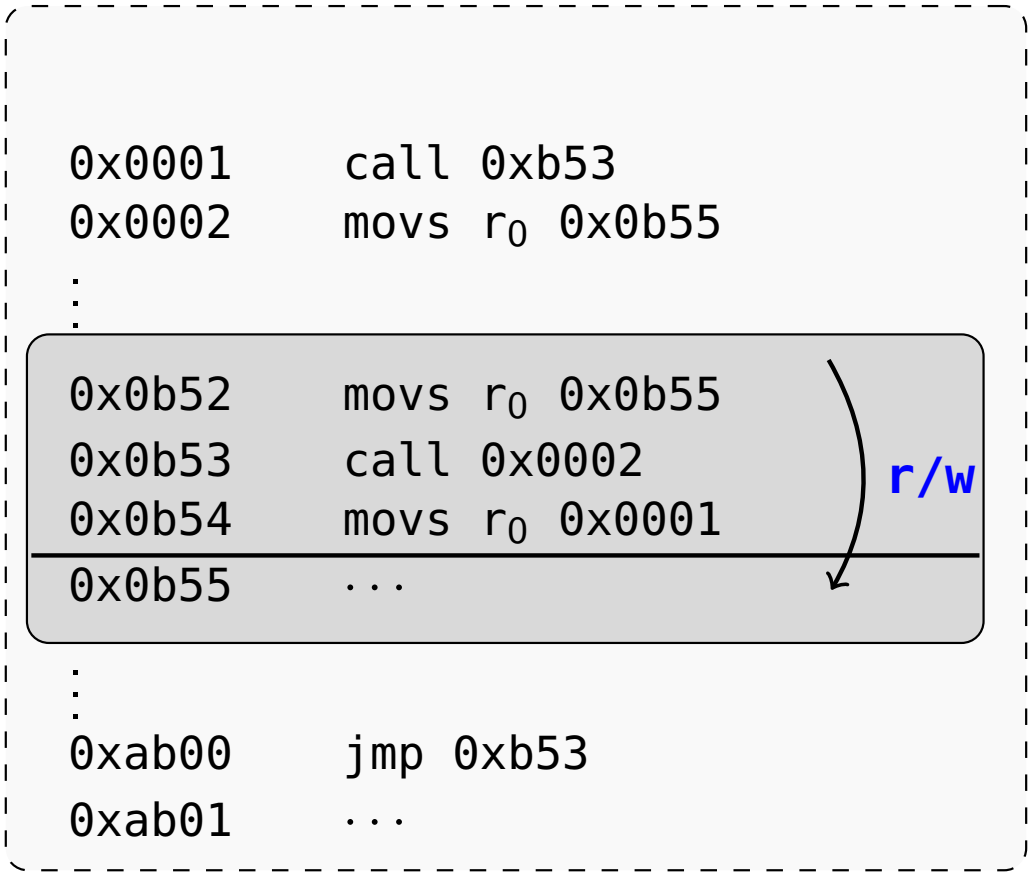
- memory space
- protected module = protected memory

PMA in action

```
0x0001    call 0xb53
0x0002    movs r0 0x0b55
   ⋮
0x0b52    movs r0 0x0b55
0x0b53    call 0x0002
0x0b54    movs r0 0x0001
0x0b55    ...
   ⋮
0xab00    jmp 0xb53
0xab01    ...
```

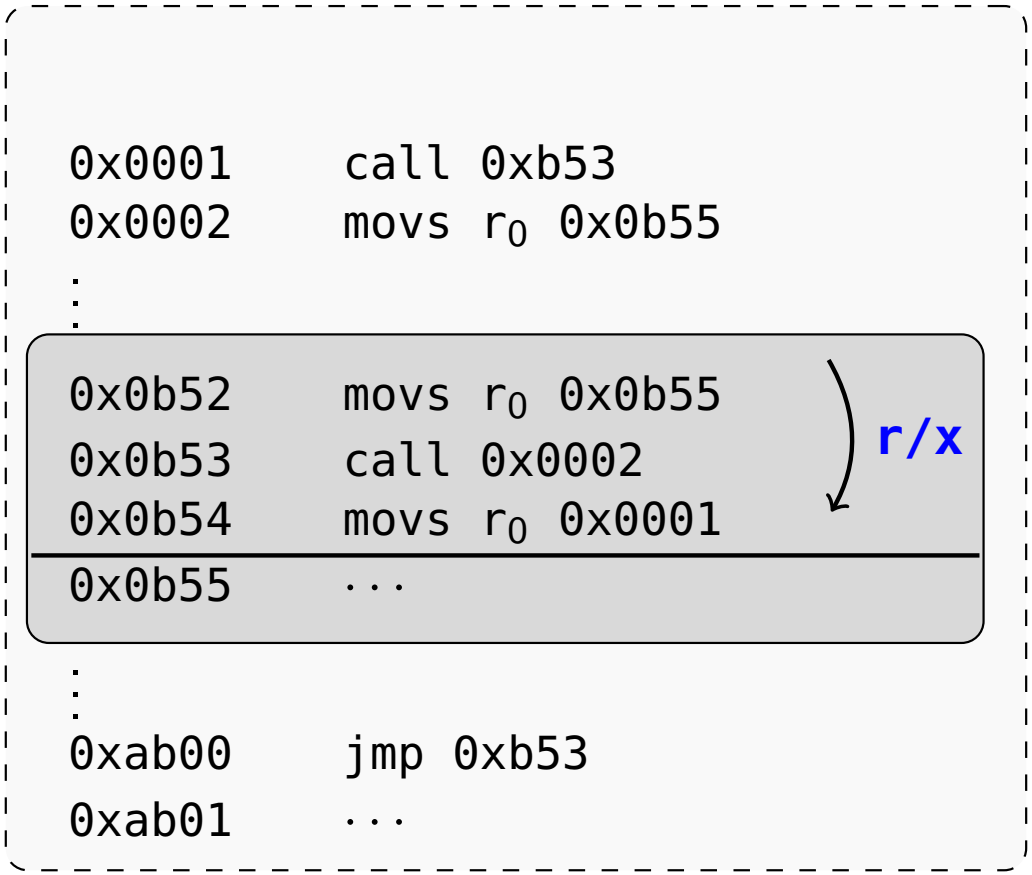
- memory space
- protected module = protected memory
- split in code and data

PMA in action



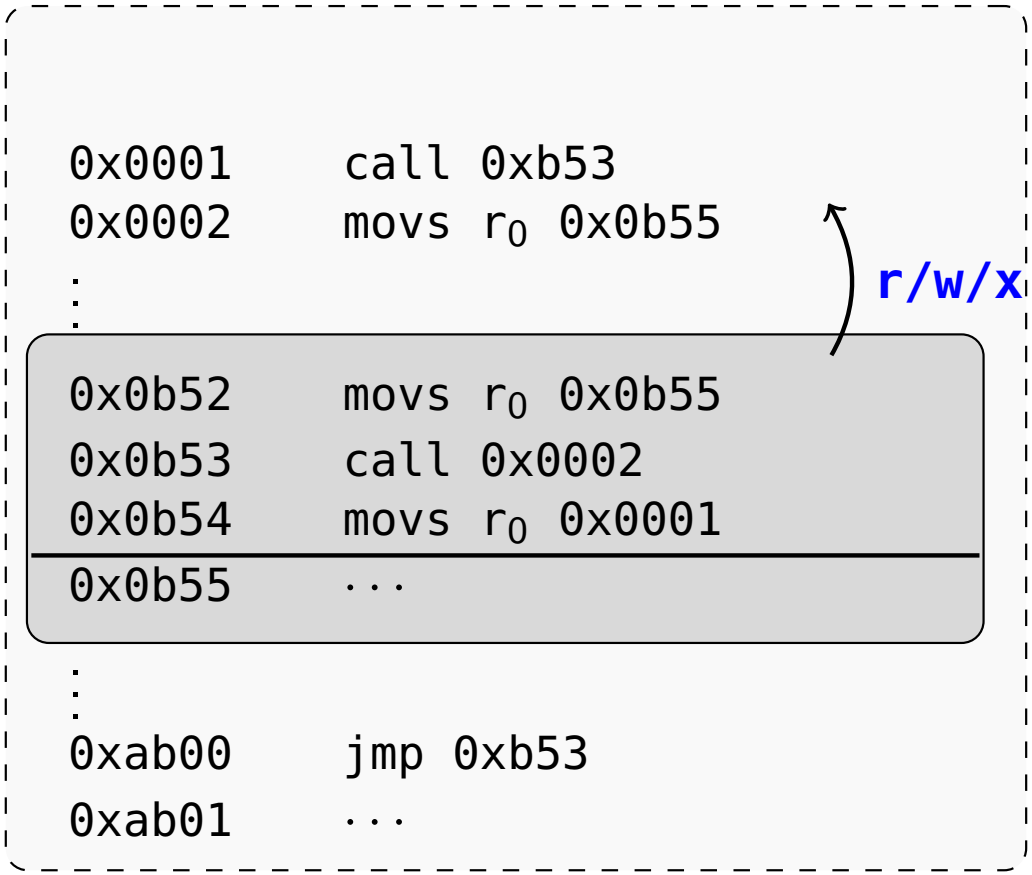
- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted

PMA in action



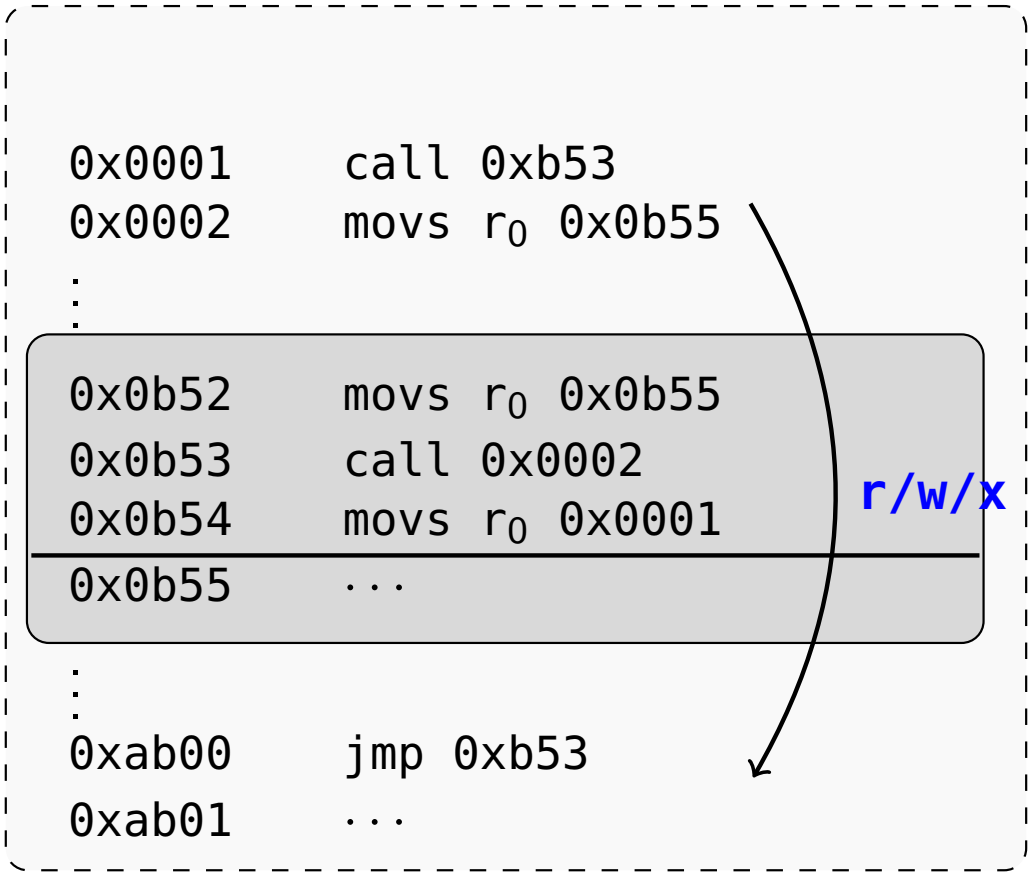
- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted

PMA in action



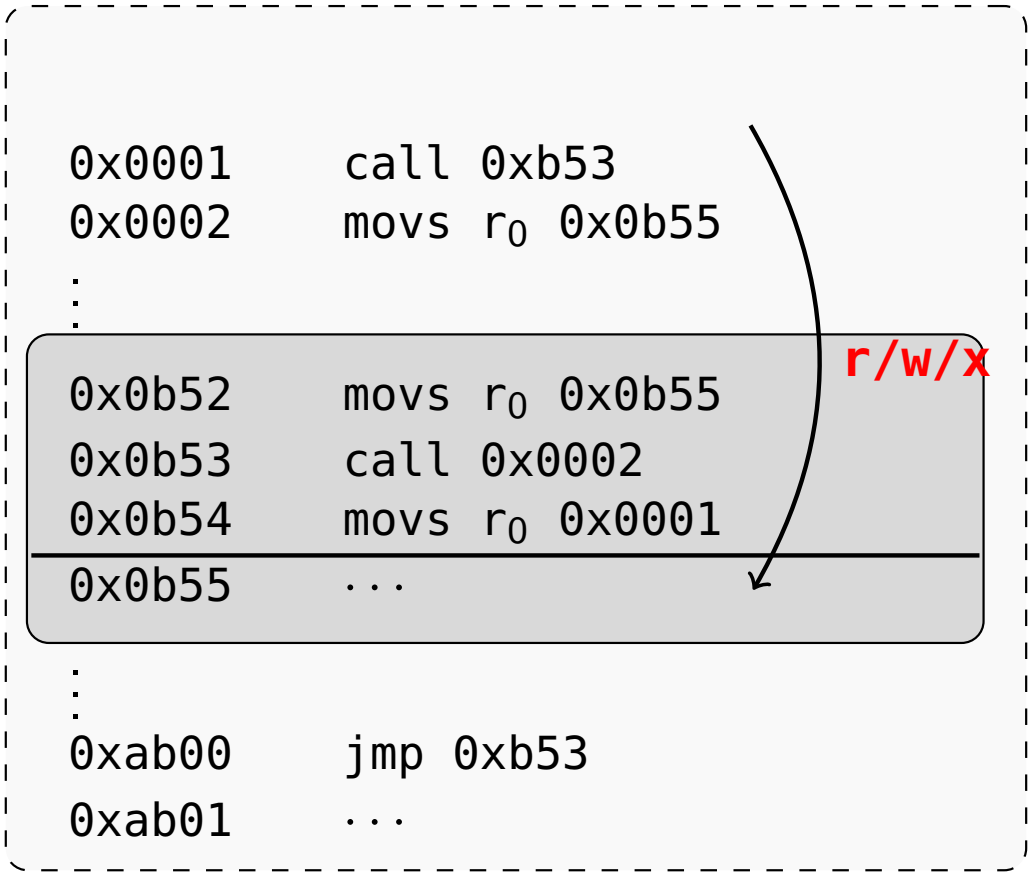
- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted

PMA in action



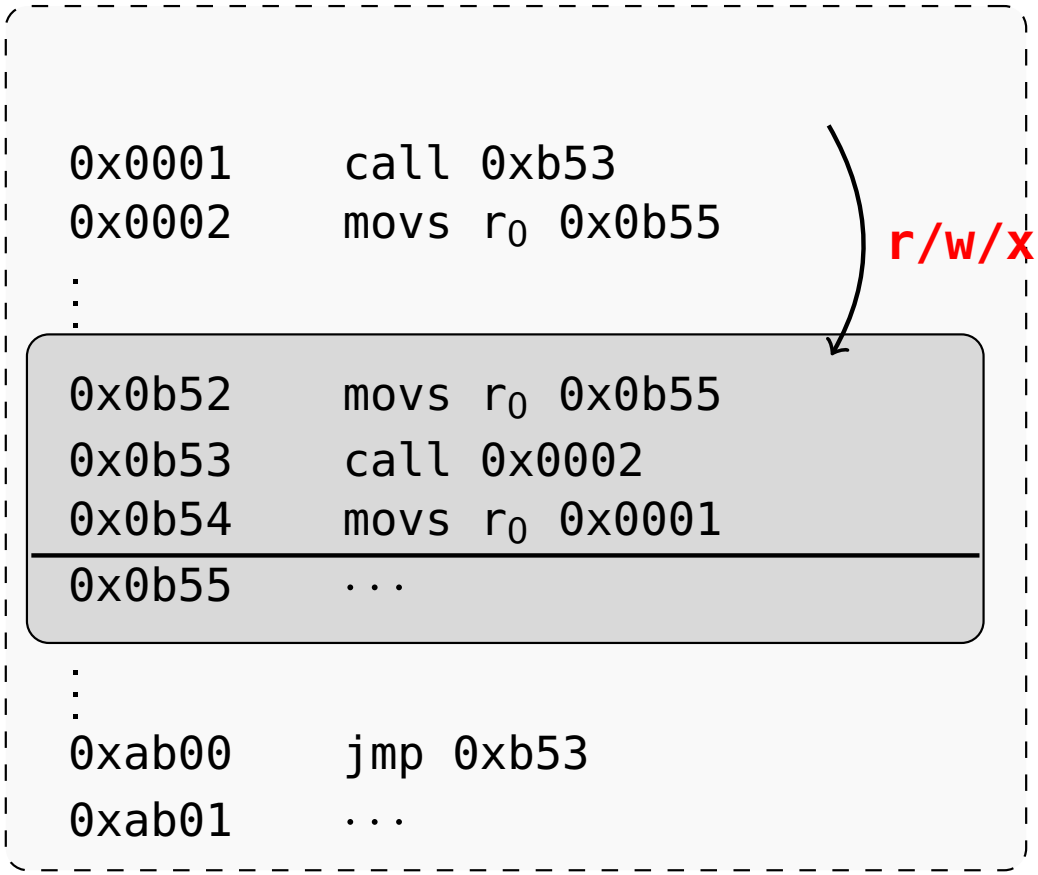
- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

PMA in action



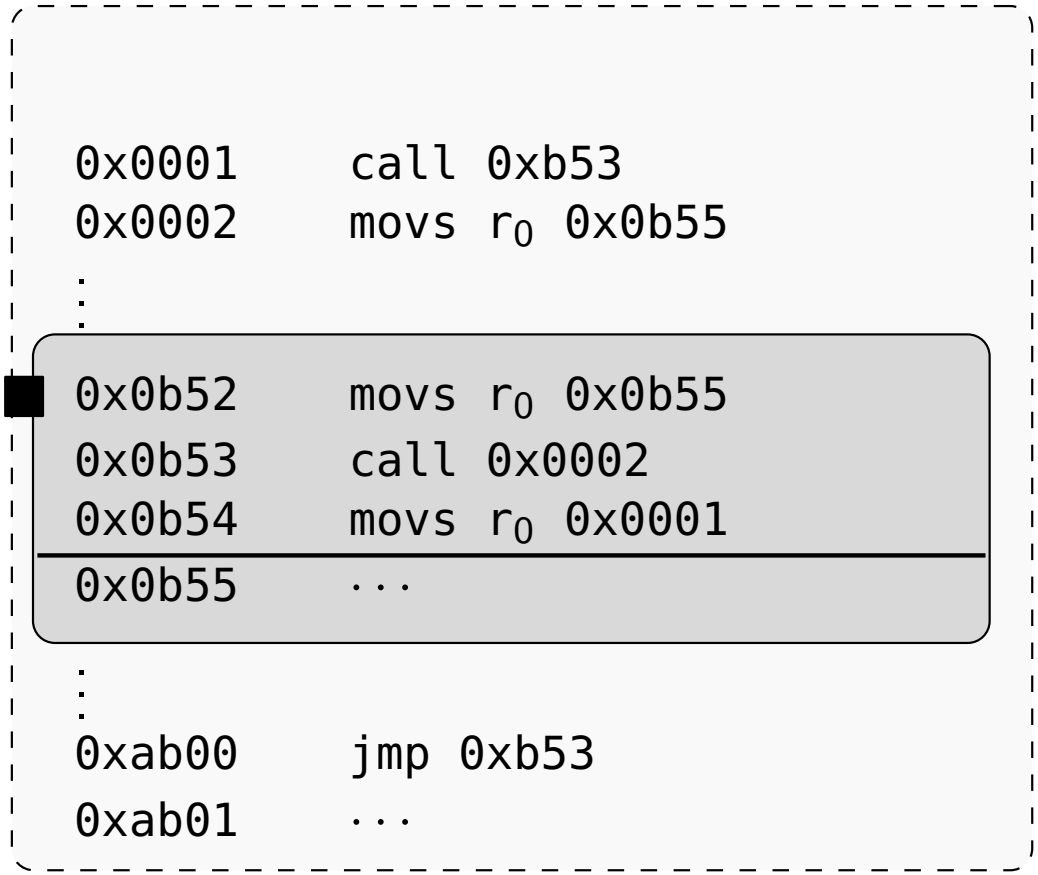
- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

PMA in action



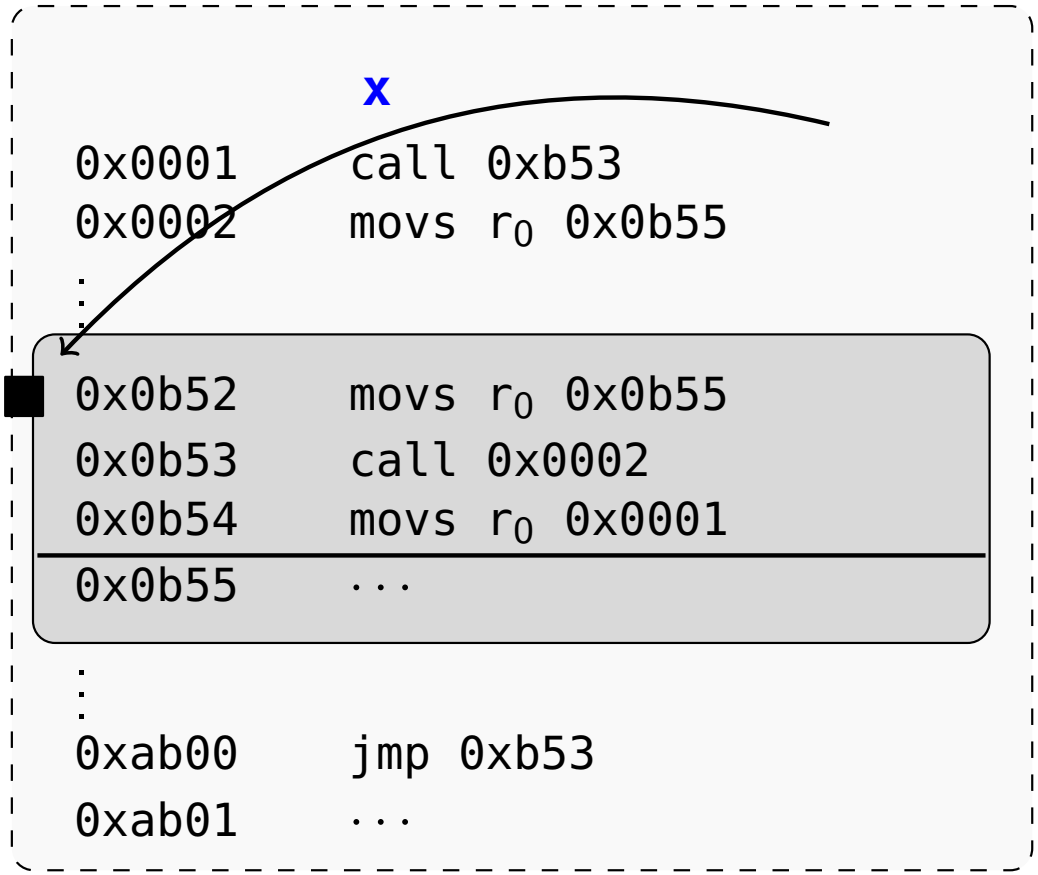
- memory space
- protected module = protected memory
- split in code and data
- protected code is unrestricted
- unprotected code is restricted

PMA in action



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

PMA in action



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

Challenge #1: Reasoning about PMA code

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ...
   ⋮
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
   ⋮
0xab01    ...
```

- to reason about this code we use contextual equivalence

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ...
   ⋮
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
   ⋮
0xab01    ...
```

- to reason about this code we use contextual equivalence
- Formally $P_1 \simeq P_2 \triangleq \forall \mathbb{C}.\ \mathbb{C}[P_1]\Uparrow \iff \mathbb{C}[P_2]\Uparrow$

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ...
   ⋮
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
   ⋮
0xab01    ...
```

- to reason about this code we use contextual equivalence
- Formally $P_1 \simeq P_2 \triangleq \forall \mathbb{C}.\ \mathbb{C}[P_1]\Uparrow \iff \mathbb{C}[P_2]\Uparrow$
- contexts are complex to reason about (but very precise)

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ...
:
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
:
0xab01    ...
```

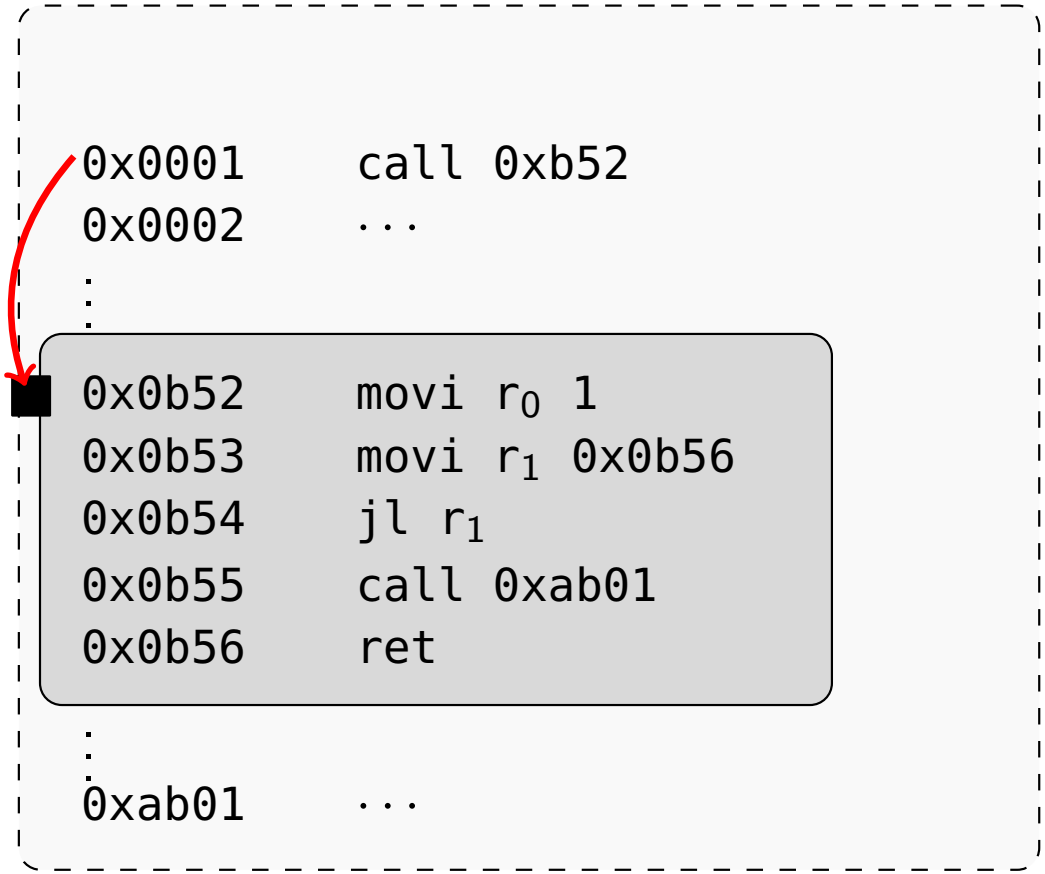
- traces are simpler [JR05]

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ···
   ⋮
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
   ⋮
0xab01    ···
```

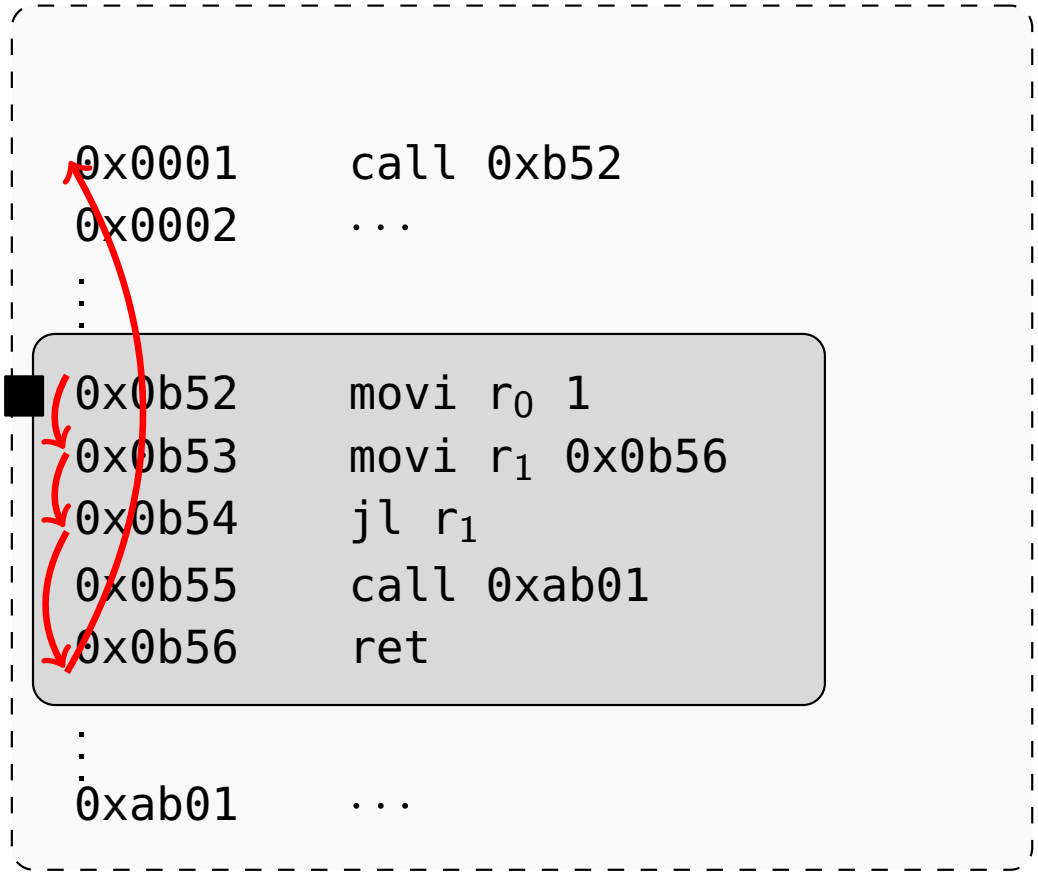
- traces are simpler [JR05]
- behaviour in this case is:

Challenge #1: Reasoning about PMA code



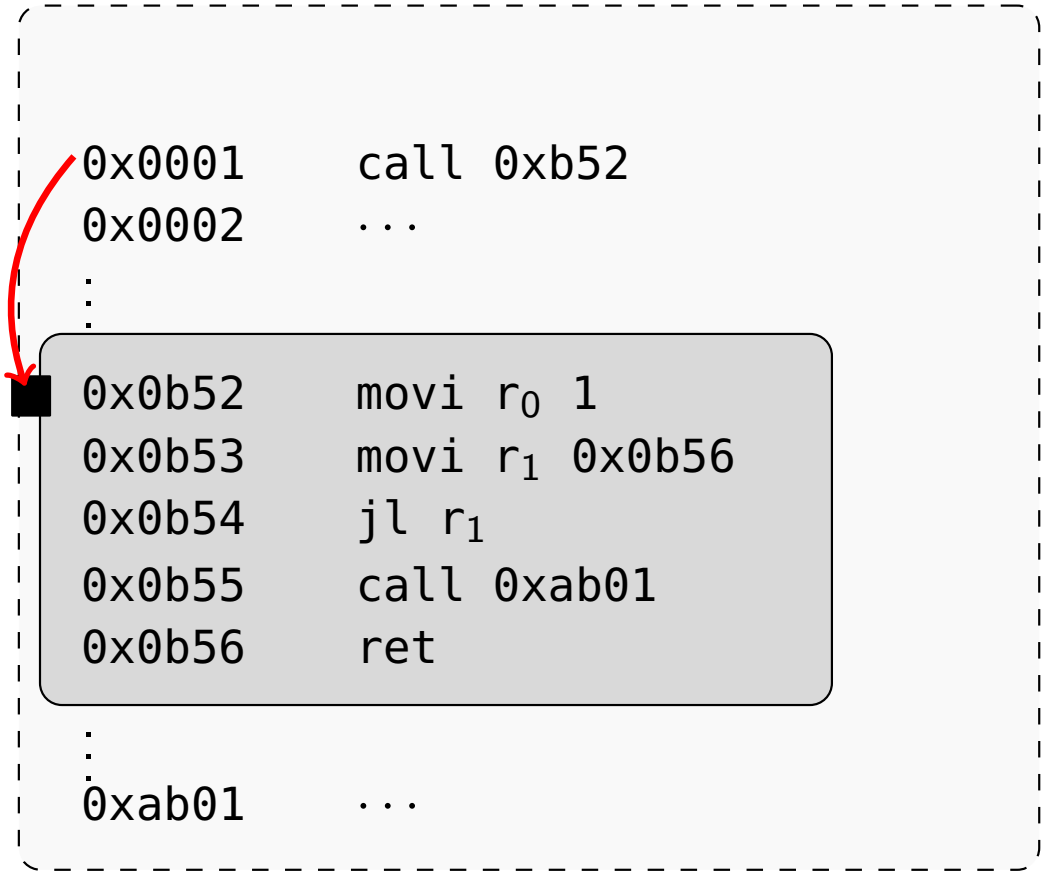
- traces are simpler [JR05]
- behaviour in this case is: `call in`

Challenge #1: Reasoning about PMA code



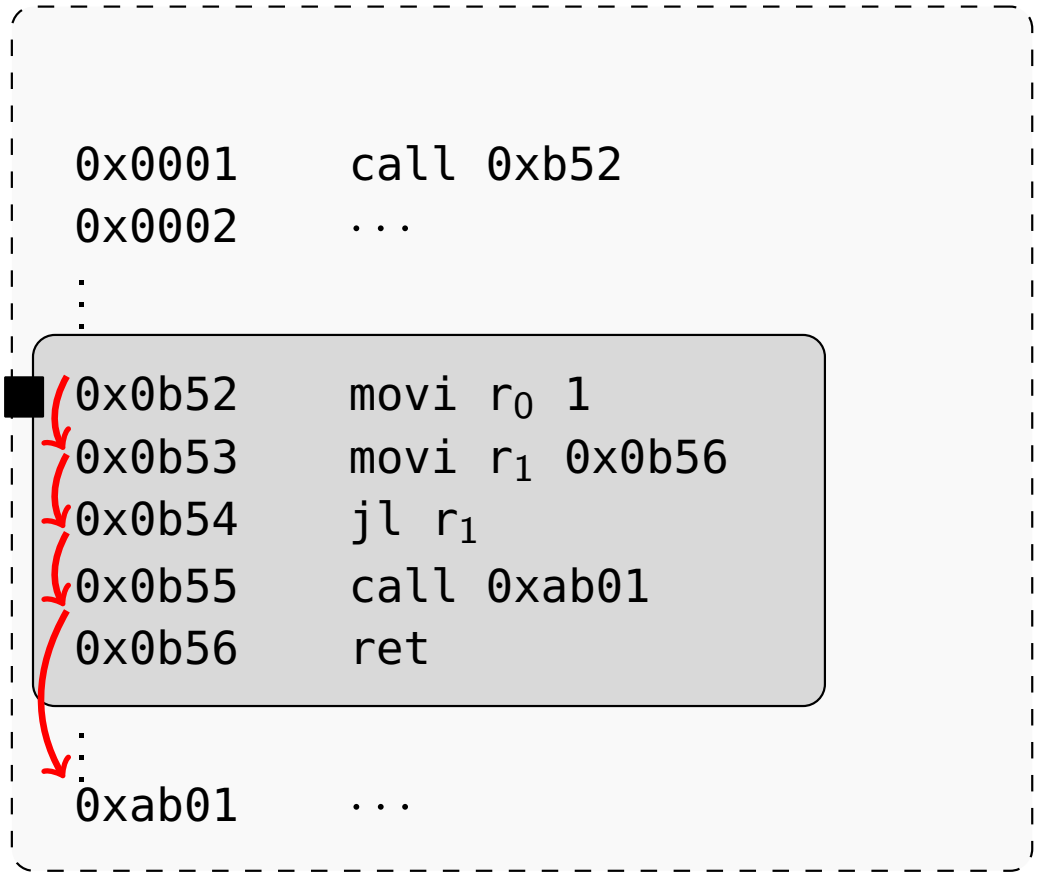
- traces are simpler [JR05]
- behaviour in this case is: `call in`, `ret 1`

Challenge #1: Reasoning about PMA code



- traces are simpler [JR05]
- behaviour in this case is:
  `call in, ret 1`
  or `call in,`

Challenge #1: Reasoning about PMA code



- traces are simpler [JR05]
- behaviour in this case is:
  `call in, ret 1`
  or `call in, call out`

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ···
   ⋮
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
   ⋮
0xab01    ···
```

- traces are simpler [JR05]
- behaviour in this case is: `call in, ret 1` or `call in, call out`
- traces rely only on the PMA code

Challenge #1: Reasoning about PMA code

```
0x0001    call 0xb52
0x0002    ···
   ⋮
0x0b52    movi r0 1
0x0b53    movi r1 0x0b56
0x0b54    jl r1
0x0b55    call 0xab01
0x0b56    ret
   ⋮
0xab01    ···
```

- traces are simpler [JR05]
- behaviour in this case is: `call in, ret 1` or `call in, call out`
- traces rely only on the PMA code
- they describe what can be observed from the outside of protected PMA code

A trace semantics for PMA

- define states S for programs

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$
- define labels (observables) α

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$
- define labels (observables) α : `call p` $\overline{\texttt{r}}$

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$
- define labels (observables) α : `call p` $\overline{\texttt{r}}$ | `ret` $\texttt{r}_0$

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$
- define labels (observables) α : $\texttt{call p } \overline{\texttt{r}} \mid \texttt{ret } \texttt{r}_0$
- define a semantics with labels $\stackrel{\alpha}{\Longrightarrow}: S \times \alpha \times S$

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$
- define labels (observables) α : $\texttt{call p } \overline{\texttt{r}} \mid \texttt{ret } \texttt{r}_0$
- define a semantics with labels $\stackrel{\alpha}{\Longrightarrow}: S \times \alpha \times S$
- $\mathrm{TR}(P) = \{\overline{\alpha} \mid \exists S'.S(P) \stackrel{\alpha_1}{\Longrightarrow} \cdots \stackrel{\alpha_n}{\Longrightarrow} S'\}$

A trace semantics for PMA

- define states S for programs
- define a semantics for PMA only: $\xrightarrow{i}: S \times S$
- define labels (observables) α : $\texttt{call p } \overline{\texttt{r}} \mid \texttt{ret r}_0$
- define a semantics with labels $\overset{\alpha}{\Longrightarrow}: S \times \alpha \times S$
- $\text{TR}(P) = \{\overline{\alpha} \mid \exists S'.S(P) \overset{\alpha_1}{\Longrightarrow} \cdots \overset{\alpha_n}{\Longrightarrow} S'\}$

$$\text{TR} = \left\{ \alpha = \left\{ \begin{array}{c} \xrightarrow{i}; \\ \texttt{call p } \overline{\texttt{r}} \\ \texttt{ret r}_0 \\ \overset{\alpha}{\Longrightarrow} \end{array} \right\}; \right\}$$

Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔

Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔
- precise formalism? ✗

Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔
- precise formalism? ✗
  1. PMA code can write in unprotected memory

Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔
- precise formalism? ✗
  1. PMA code can write in unprotected memory
  2. flags convey information across function calls

Challenge #2: Precise reasoning

- formalism to reason about PMA code simply: ✔
- precise formalism? ✗
  1. PMA code can write in unprotected memory
  2. flags convey information across function calls
  3. registers besides r_0 in `ret` as well

Fully abstract trace semantics

To ensure maximal precision, prove the trace semantics to be fully abstract

Fully abstract trace semantics

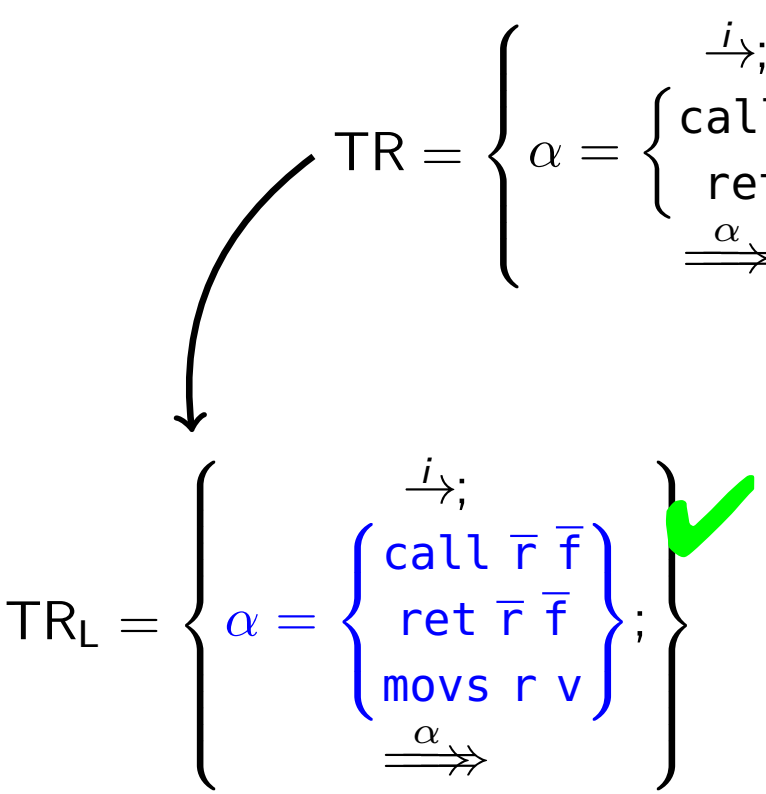
To ensure maximal precision, prove the trace semantics to be fully abstract

i.e. there are no other things that we missed

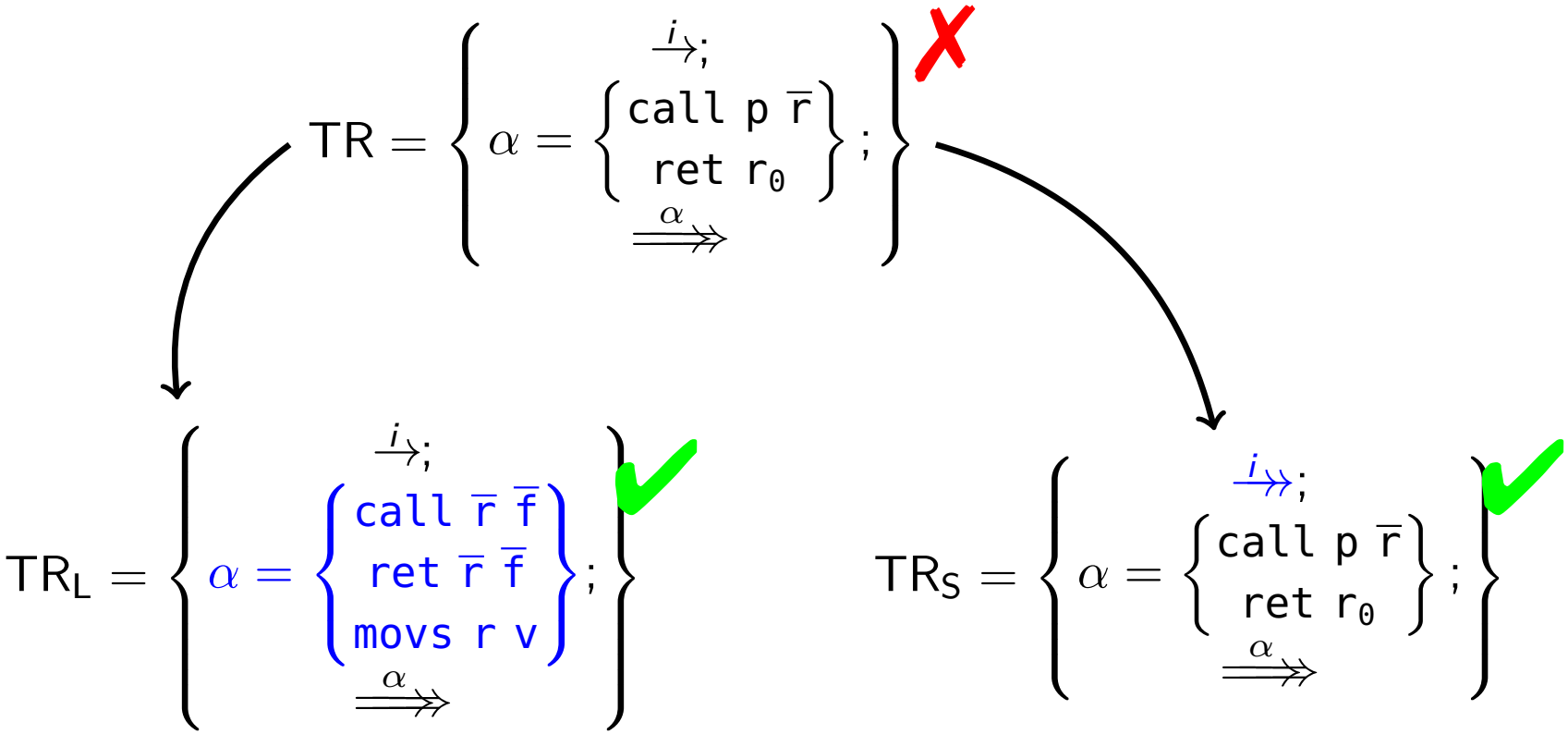
The spectrum of full abstraction [Cur07]

$$\mathsf{TR} = \left\{ \alpha = \left\{ \begin{array}{l} \xrightarrow{i}; \\ \texttt{call p } \overline{\texttt{r}} \\ \texttt{ret r}_0 \\ \Longrightarrow^{\alpha} \end{array} \right\}; \right\}^{\checkmark}$$

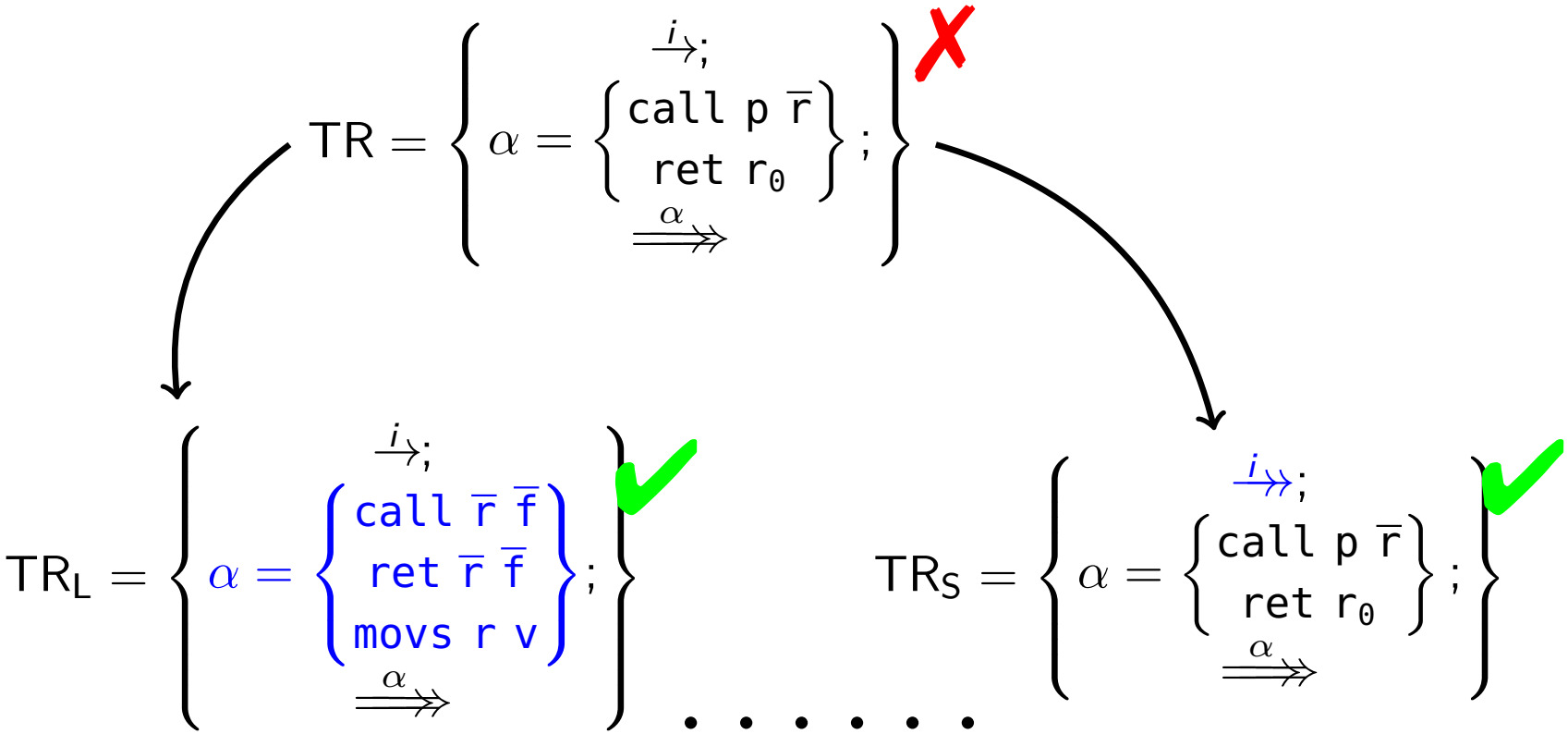
The spectrum of full abstraction [Cur07]



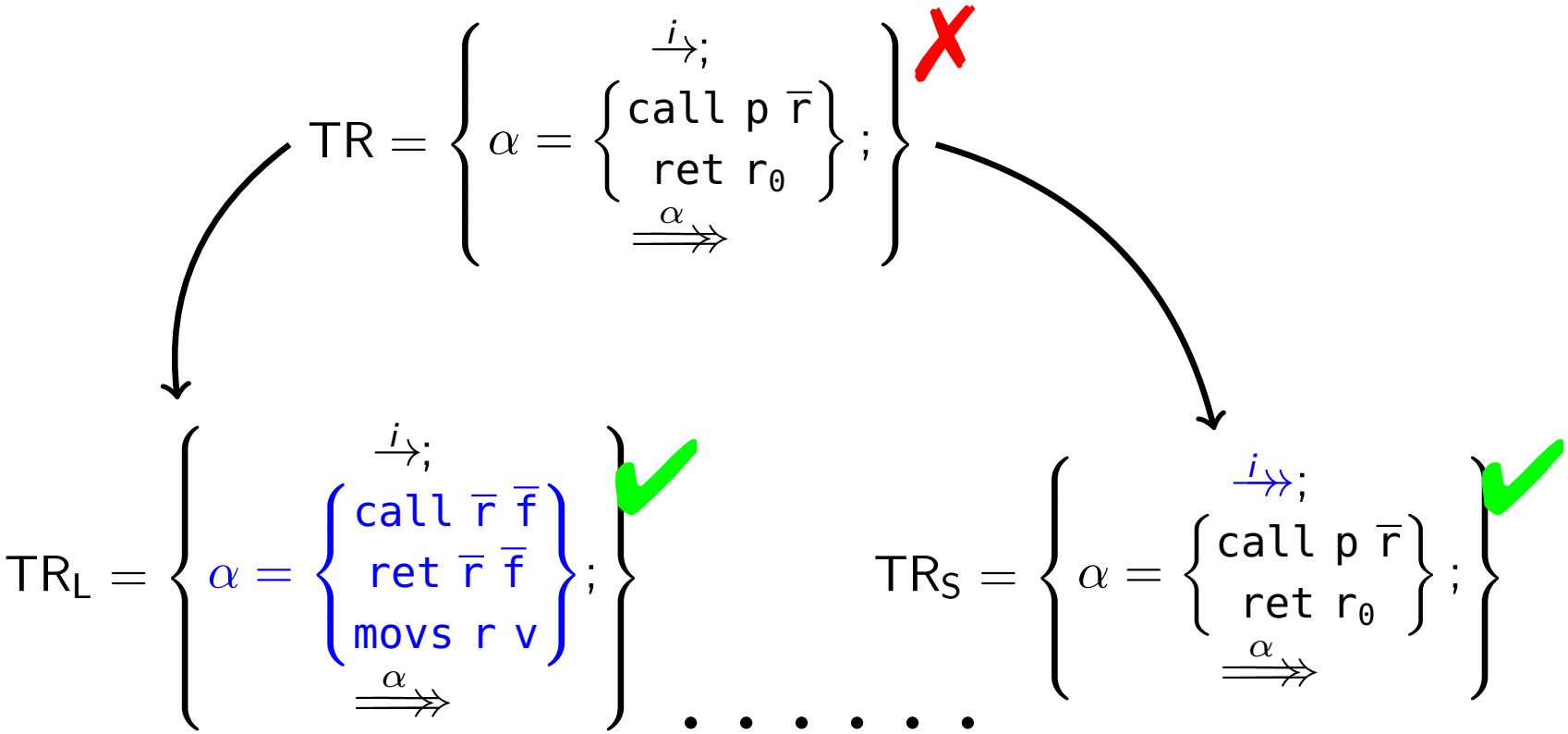
The spectrum of full abstraction [Cur07]



The spectrum of full abstraction [Cur07]



The spectrum of full abstraction [Cur07]



$$\mathrm{TR}_X(P_1) = \mathrm{TR}_X(P_2) \iff P_1 \simeq P_2$$

Conclusion

Conclusion

- PMA is a very powerful, emerging security construct

Conclusion

- PMA is a very powerful, emerging security construct
- trace semantics for PMA enables easier reasoning about PMA code

Conclusion

- PMA is a very powerful, emerging security construct
- trace semantics for PMA enables easier reasoning about PMA code
- if the trace semantics is fully abstract, trace equivalence can replace contextual equivalence altogether

Conclusion

- PMA is a very powerful, emerging security construct
- trace semantics for PMA enables easier reasoning about PMA code
- if the trace semantics is fully abstract, trace equivalence can replace contextual equivalence altogether
- the paper provides two, examples of fully abstract trace semantics for PMA

Questions

Thank you!

Qs ?

Bibliographical References I


Pieter Agten, Frank Piessens, and Bart Jacobs, *Sound modular verification in an unverified context*, 2014, to appear.


Pierre-Louis Curien, *Definability and full abstraction*, Electron. Notes Theor. Comput. Sci. **172** (2007), 301–310.


Karim Eldefrawy, Aurélien Francillon, Daniele Perito, and Gene Tsudik, *SMART: Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust*, NDSS 2012, 19th Annual Network and Distributed System Security Symposium (San Diego, United States), 2012.


Alan Jeffrey and Julian Rathke, *Java Jr.: fully abstract trace semantics for a core Java language*, ESOP'05, LNCS, vol. 3444, Springer, 2005, pp. 423–438.


Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar, *Innovative instructions and software model for isolated execution*, HASP '13, ACM, 2013, pp. 10:1–10:1.


Jonathan M. McCune, Yanlin Li, Ning Qu, Zongwei Zhou, Anupam Datta, Virgil Gligor, and Adrian Perrig, *Trustvisor: Efficient TCB reduction and attestation*, SP '10 (Washington, DC, USA), IEEE, 2010, pp. 143–158.


Jonathan M. McCune, Bryan J. Parno, Adrian Perrig, Michael K. Reiter, and Hiroshi Isozaki, *Flicker: an execution infrastructure for TCB minimization*, SIGOPS Oper. Syst. Rev. **42** (2008), no. 4, 315–328.

Bibliographical References II

Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens, *Sancus: Low-cost trustworthy extensible networked devices with a zero-software Trusted Computing Base*, Proceedings of the 22nd USENIX conference on Security symposium, USENIX Association, 2013.

Marco Patrignani, Dave Clarke, and Frank Piessens, *Secure Compilation of Object-Oriented Components to Protected Module Architectures*, Proceedings of the 11th Asian Symposium on Programming Languages and Systems (APLAS'13), LNCS, vol. 8301, 2013, pp. 176–191.

Raoul Strackx and Frank Piessens, *Fides: selectively hardening software application components against kernel-level or process-level malware*, Proceedings of the 2012 ACM conference on Computer and communications security (New York, NY, USA), CCS '12, ACM, 2012, pp. 2–13.