Formally verifying exceptions in low-level code with Separation Logic

Marco Paviotti and Jesper Bengtson
IT University of Copenhagen

22nd October
Nordic Workshop on Programming Theory (NWPT)
Reykjavik, Iceland

Verification of low-level code

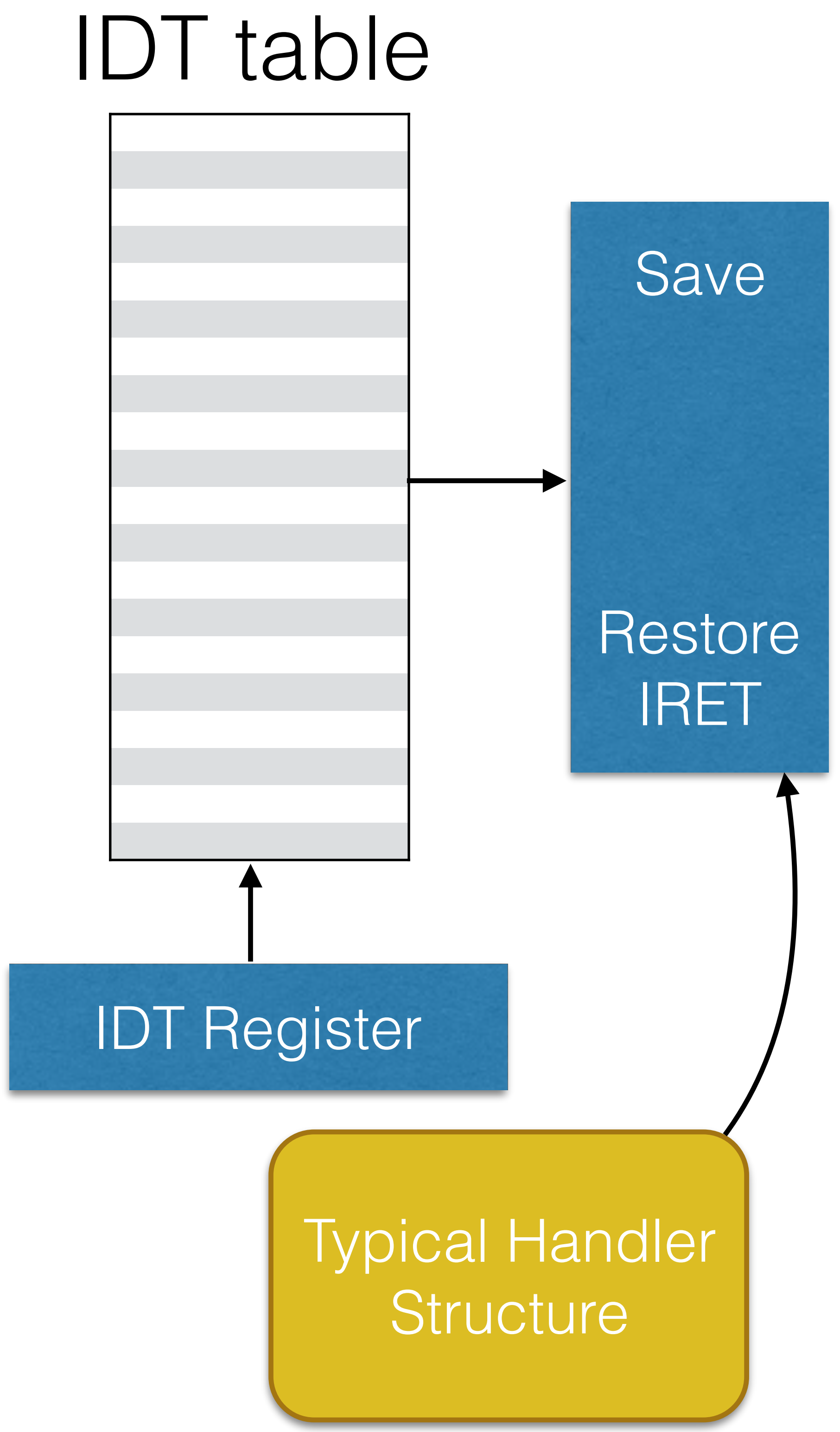
- Hand-crafted code often found in security-critical places (e.g. kernels)
- Mechanically verifying low-level, unstructured code is crucial
- Categorical models have successfully inspired separation logic
  - higher-order and shared memory concurrency (iCAP, Svendsen and Birkedal)
  - for verification of low-level code (Jensen, Kennedy and Benton)
- Kernels make heavy use of exceptions/interrupts
- There is no nice logic/model accounting for these behaviours

Interrupts

When an interrupt fires the CPU:

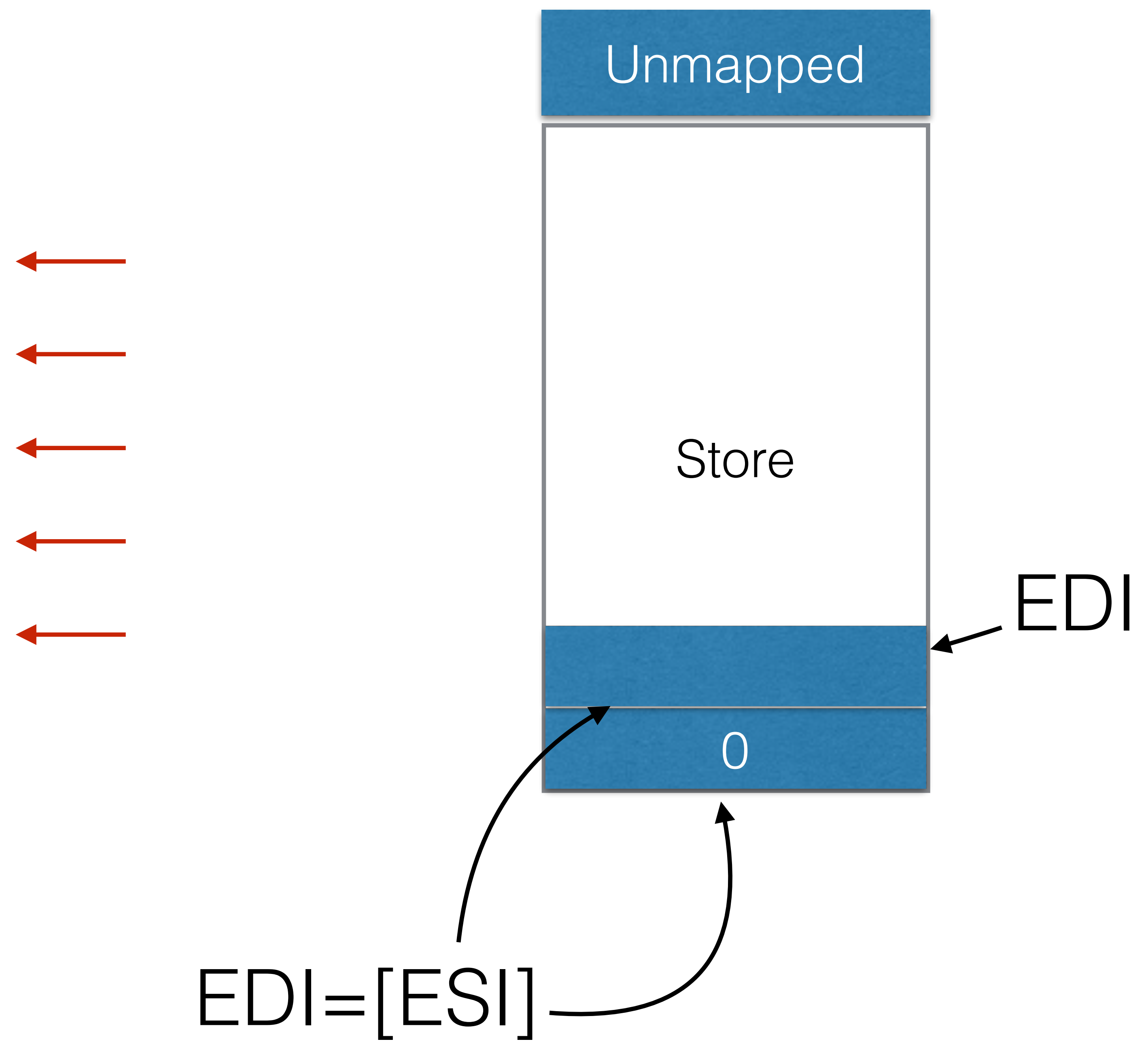
- looks up the address of the handler in the IDT table
- Stores the return address on the top of the stack
- Jumps to the handler

It is the handler responsibility to restore the state and return from the interrupt



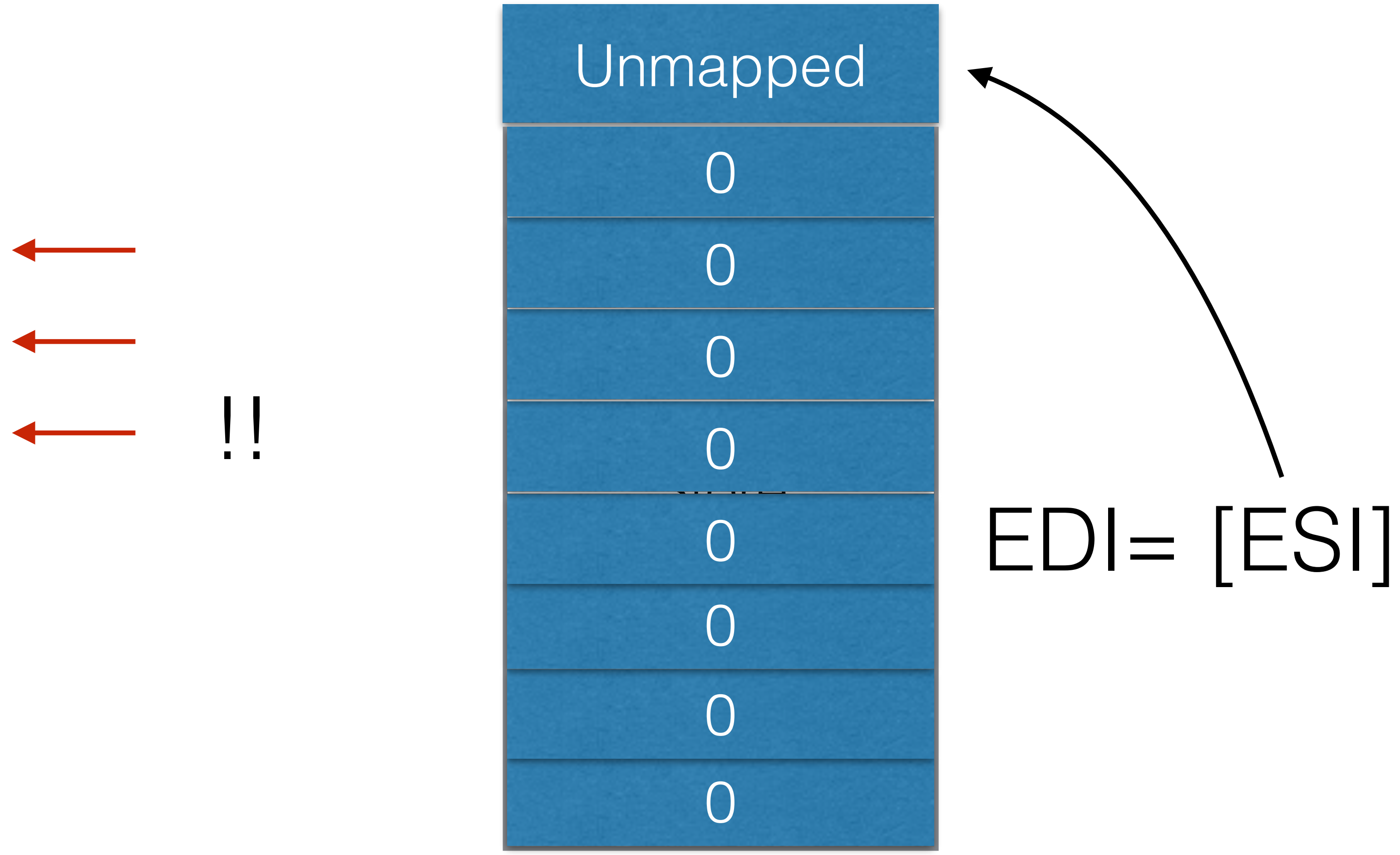
Motivating Example

$\text{mov ESI}, info;$
$\text{mov EDI}, [\text{ESI}];$
$\text{mov } [\text{EDI}],\ 0;$
$\text{add EDI}, 4;$
$\text{mov } [\text{ESI}],\ \text{EDI}.$



Motivating Example

$$\begin{array}{l} \text{mov ESI}, info; \\ \text{mov EDI}, [\text{ESI}]; \\ \text{mov [EDI]},\ 0; \\ \text{add EDI}, 4; \\ \text{mov [ESI]},\ \text{EDI}. \end{array}$$



Our contributions

We rely on an existing Coq formalisation of the assembly x86[1,2]

- Semantics and instruction rules for exceptions
- We prove their use by verifying the memory allocator example

[1]Andrew Kennedy, Nick Benton, Jonas Braband Jensen, and Pierre-Evariste Dagand. Coq: the world's best macro assembler? In PPDP 2013.

[2]J. B. Jensen, N. Benton, and A. J. Kennedy. High-level separation logic for low-level code. POPL 2013

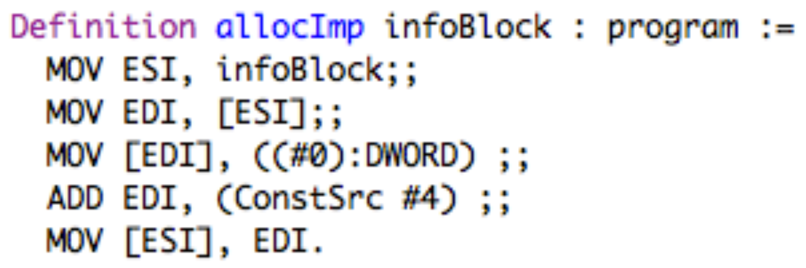
Goals and Related work

- First step towards asynchronous interrupts and thus Verification of Device Drivers and Schedulers (Concurrency)
- Our end is similar to Feng et al.'s[1], but
    - here we don't rely on abstractions
    - Want: a nice model

[1] X. Feng, Z. Shao,Y. Guo, Y. Dong. Certifying Low-Level Programs with Hardware Interrupts and Preemptive Threads

Coq and assembly

- Code is data
- Unstructured assembly code
- Memory : $\text{list } 32 \text{ bool} \rightharpoonup \text{list } 32 \text{ bool}$
- Using notation can write assembly code in Coq



```
Definition allocImp infoBlock : program :=
  MOV ESI, infoBlock;;
  MOV EDI, [ESI];;
  MOV [EDI], ((#0):DWORD) ;;
  ADD EDI, (ConstSrc #4) ;;
  MOV [ESI], EDI.
```

Higher-Order Separation Logic for low-level code[1]

Post-Condition

Precondition

$$\vdash (\mathrm{safe} \otimes \mathrm{EIP} \mapsto j \star Q) \to (\mathrm{safe} \otimes \mathrm{EIP} \mapsto i \star P) \oslash i..j \mapsto c$$

Code

Meaning

$\{P\}\ c\ \{Q\}$ (when c is a code block)

“If the code is safe to run from Q then it is safe to run from the state P ”

[1]J. B. Jensen, N. Benton, and A. J. Kennedy. High-level separation logic for low-level code. POPL 2013

Loop example

It is safe to sit in a tight loop forever:

$$\vdash (\mathrm{safe} \otimes \mathrm{EIP} \mapsto i) \oslash i \mapsto \mathrm{JMP}\ i$$

Proof.

It suffices to show that
if the loop is safe for k-1 steps (“later”) then it is safe for k steps (“now”)

$$\frac{\rhd\ \mathrm{safe} \otimes \mathrm{EIP} \mapsto i \oslash i \mapsto \mathrm{JMP}\ i \vdash \mathrm{safe} \otimes \mathrm{EIP} \mapsto i \oslash i \mapsto \mathrm{JMP}\ i}{\vdash \mathrm{safe} \otimes \mathrm{EIP} \mapsto i \oslash i \mapsto \mathrm{JMP}\ i}$$

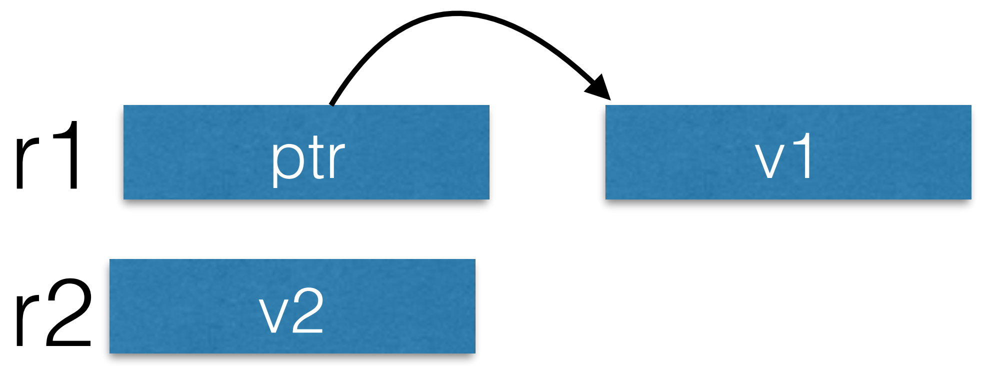
The “later” modality is due to [Nakano, 2000]

Löb Induction

$$\frac{\rhd S \vdash S}{\vdash S}$$

Rule format

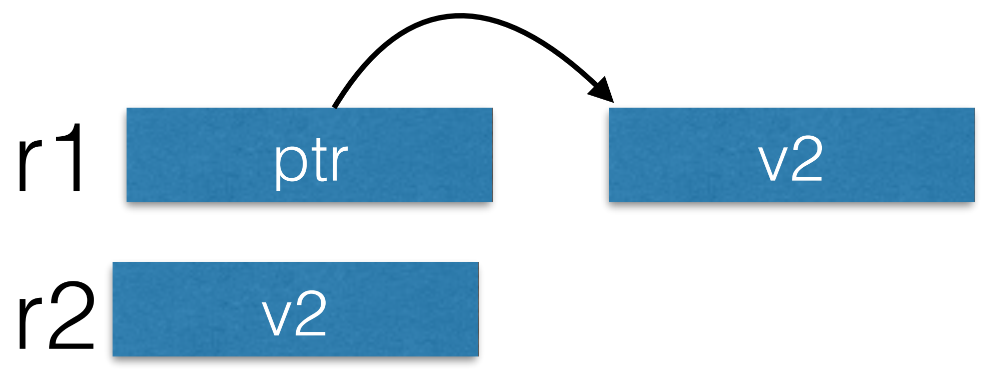
$\Big\{$ safe $\otimes$ (EIP $\mapsto i \star$



$) \Big\}$

$$i..j \mapsto \text{mov}[r_1], r_2$$

$\Big\{$ safe $\otimes$ (EIP $\mapsto j \star$



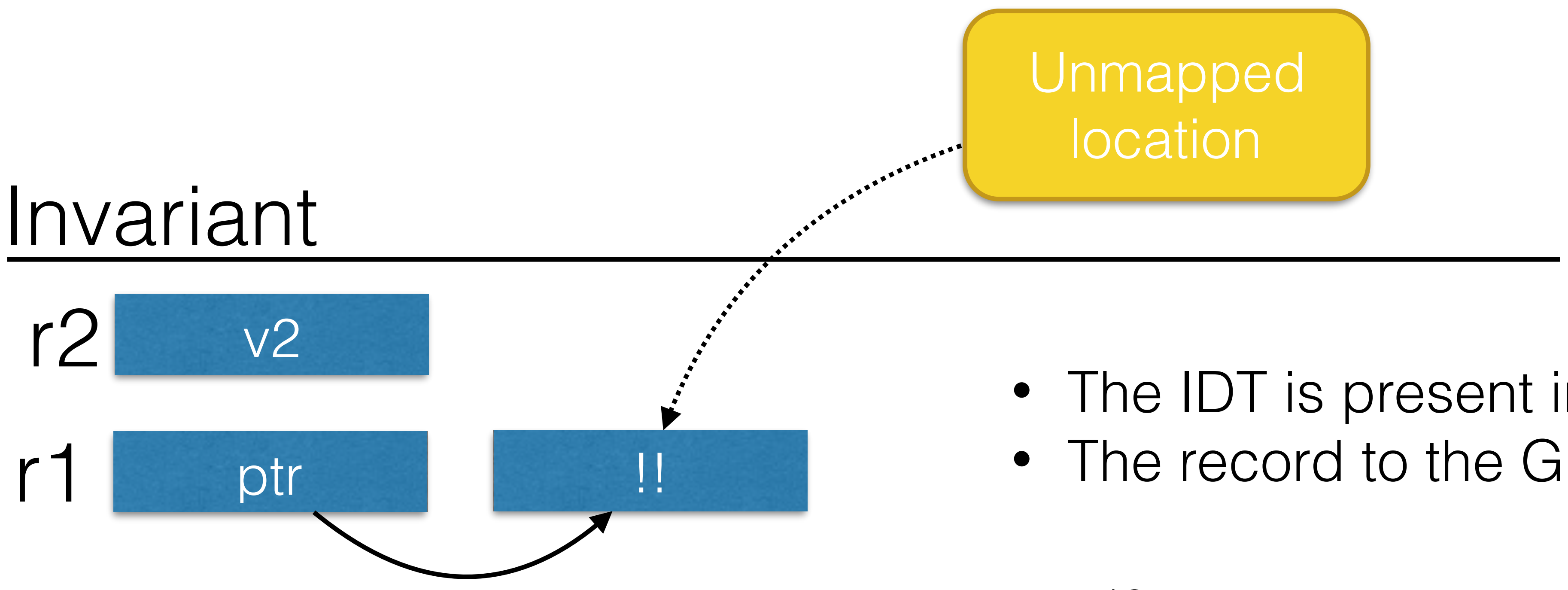
$) \Big\}$

Exceptions: mov as jumps

$$\left\{ \text{safe} \otimes (\text{EIP} \mapsto i \star\ \text{ESP}\ \boxed{\text{ptr}} \xrightarrow{} \boxed{?}\ ) \right\}$$

$$i..j \mapsto \text{mov}[r_1], r_2$$

$$\rhd \left\{ \text{safe} \otimes (\text{EIP} \mapsto fail \star\ \text{ESP}\ \boxed{\text{ptr}} \xrightarrow{} \boxed{\text{j}}\ ) \right\}$$



- The IDT is present in the memory,
- The record to the GPE links to the *fail* address

Memory allocator

P

$$\text{safe} \otimes \text{EIP} \mapsto i \star \text{EDI} \mapsto \_$$
$$\star\, \text{ESP} \mapsto s \star s-4..s \mapsto sv$$

```
mov ESI, info;
mov EDI, [ESI];
mov [EDI], 0;
add EDI, 4;
mov [ESI], EDI.
```

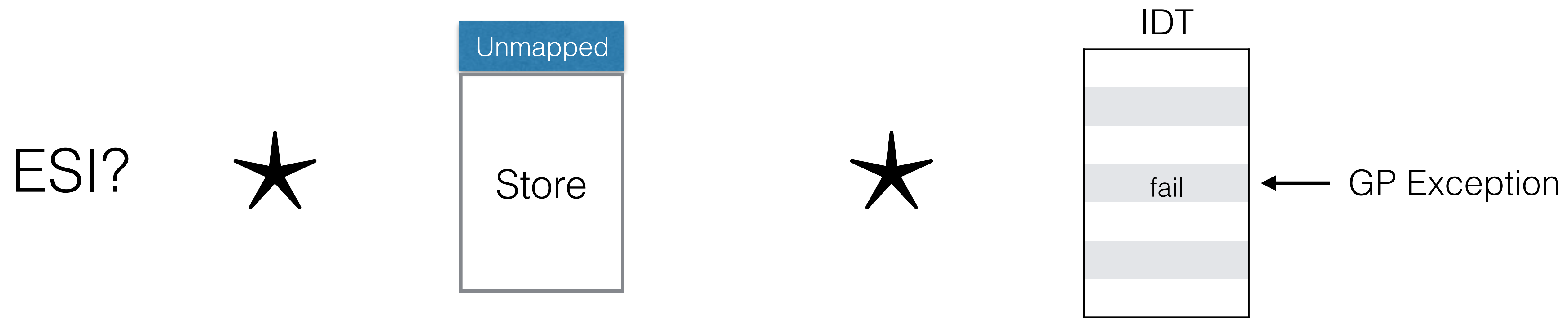
Q1

$$\text{safe} \otimes \text{EIP} \mapsto j \star \text{INTL} \mapsto 0$$
$$\star\, \text{ESP} \mapsto s \star s-4..s \mapsto sv$$
$$\star\, \exists p.\text{EDI} \mapsto p+4 \exists unk.p..(p+4) \mapsto unk$$

Q2

$$\text{safe} \otimes \text{EIP} \mapsto fail \star \text{INTL} \mapsto 1$$
$$\star\, \text{ESP} \mapsto s-4 \star \exists ret.s-4..s \mapsto ret$$

Invariant

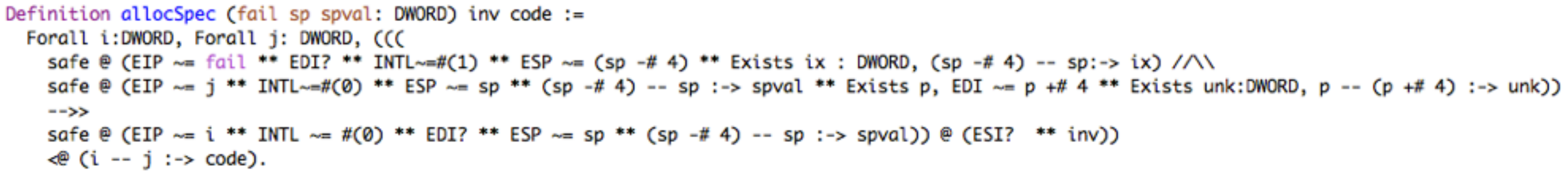


Correctness of the Allocator

Theorem

$$\vdash ((\text{safe} \otimes \underset{\text{Success}}{Q_1} \wedge \text{safe} \otimes \overset{\text{Exception occurs}}{Q_2}) \rightarrow \text{safe} \otimes \overset{\text{Precondition}}{P}) \oslash i..j \mapsto c \otimes \underset{\text{IDT}}{R}$$

Coq code



```
Definition allocSpec (fail sp spval: DWORD) inv code :=
  Forall i:DWORD, Forall j: DWORD, (((
    safe @ (EIP ~= fail ** EDI? ** INTL~=#(1) ** ESP ~= (sp -# 4) ** Exists ix : DWORD, (sp -# 4) -- sp:-> ix) //\\
    safe @ (EIP ~= j ** INTL~=#(0) ** ESP ~= sp ** (sp -# 4) -- sp :-> spval ** Exists p, EDI ~= p +# 4 ** Exists unk:DWORD, p -- (p +# 4) :-> unk))
    -->>
    safe @ (EIP ~= i ** INTL ~= #(0) ** EDI? ** ESP ~= sp ** (sp -# 4) -- sp :-> spval)) @ (ESI?  ** inv))
    <@ (i -- j :-> code).
```

Conclusions

- We extended Jensen et al.'s formalisation to cover programs with exceptions
- The logic is robust: we didn't need a new model/logic
- I didn't have time for showing: a lot of Coq code

What do we do next?

- Are interrupts effects or threads?
- Concurrency as a primitive (CSL/Shared memory)

Thanks!