Enforcing Information Flow Policies via Generation of Monitors in Java Card Runtime Environments

Alessandro Coglio
Stephen Fitzpatrick
Cordell Green
Lindsay Errington

Kestrel Institute

HCSS
May 5th, 2011

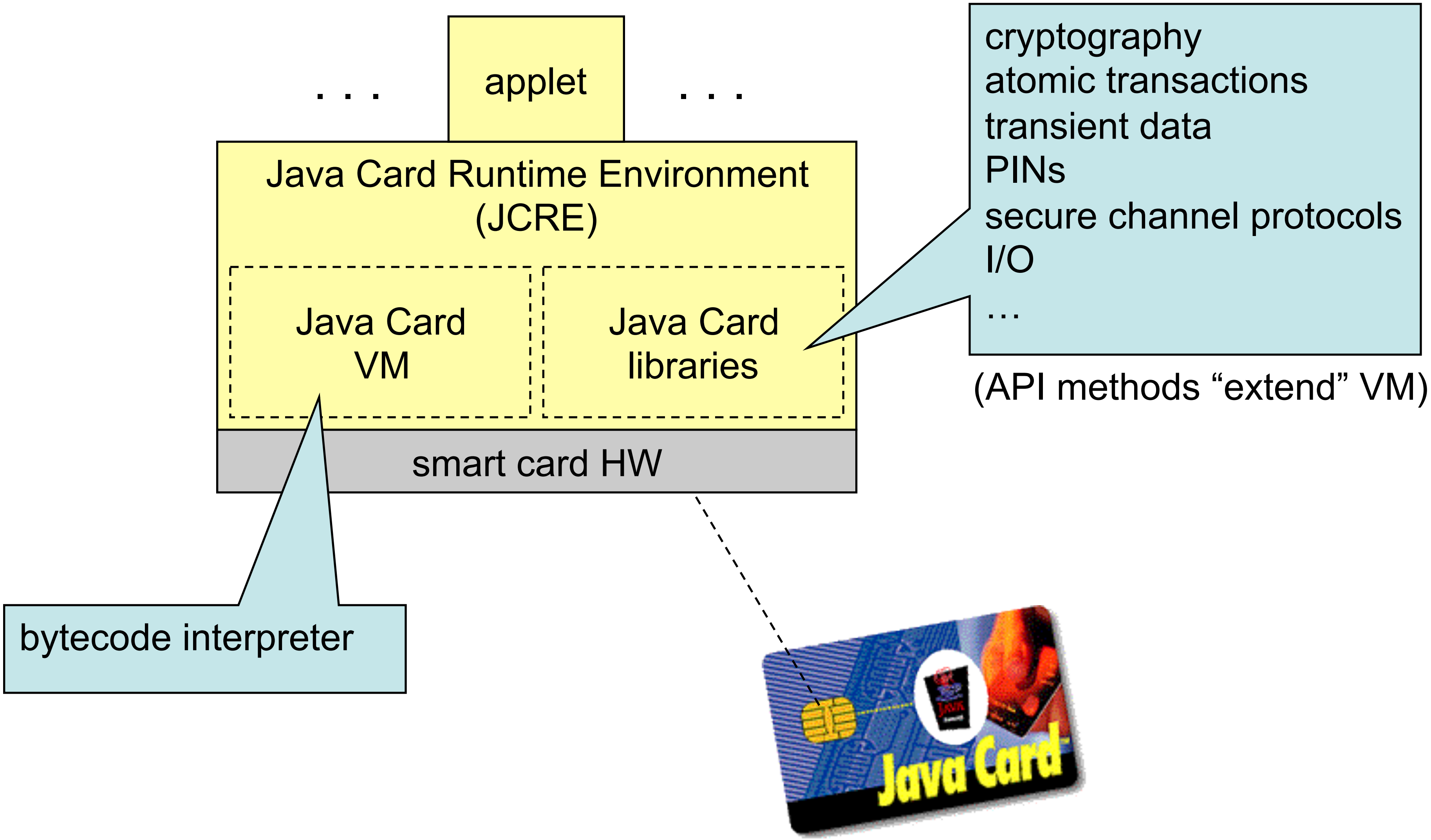
Java Card = Java for smart cards

- language subset
- different APIs

authentication,
banking,
telephony,
health care,
…

“Java Card” denotes
- language + API
- card (w/ Java SW)

Java Card Runtime Environment = smart card OS



. . .

applet

. . .

Java Card Runtime Environment (JCRE)

Java Card VM

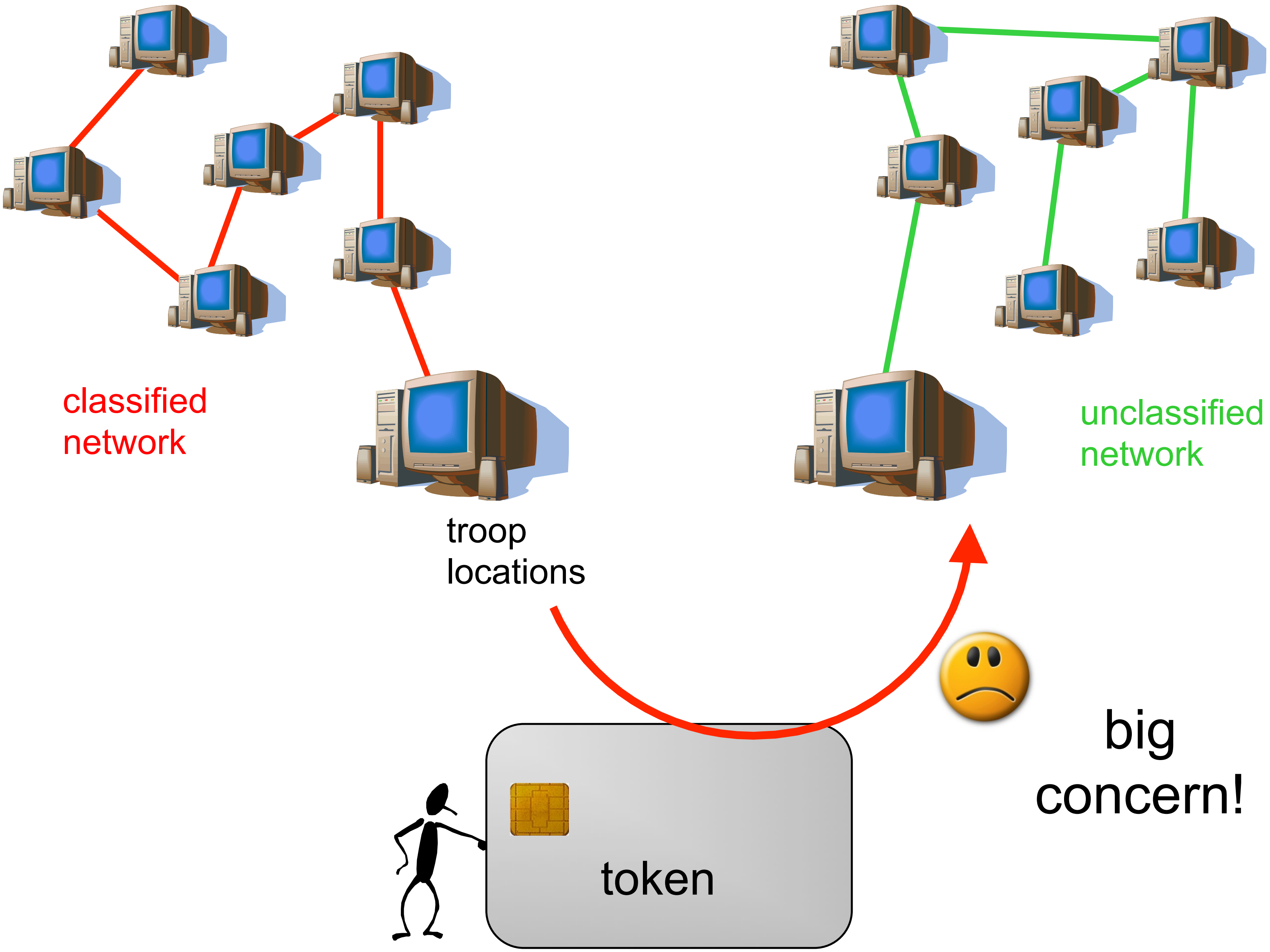
Java Card libraries

smart card HW

cryptography
atomic transactions
transient data
PINs
secure channel protocols
I/O
…

(API methods “extend” VM)

bytecode interpreter



classified
network

troop
locations

unclassified
network

token

big
concern!

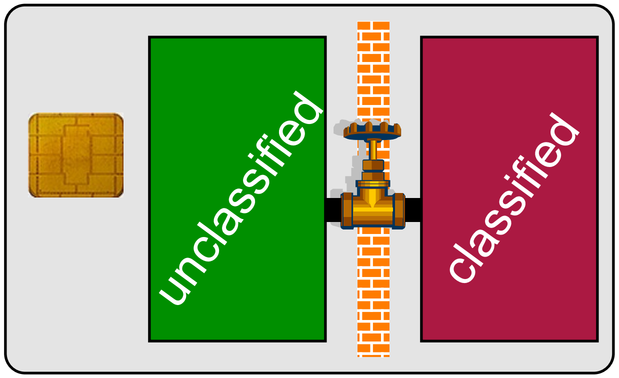
Current Solution

separate tokens

- more expensive
- no cross-domain integration

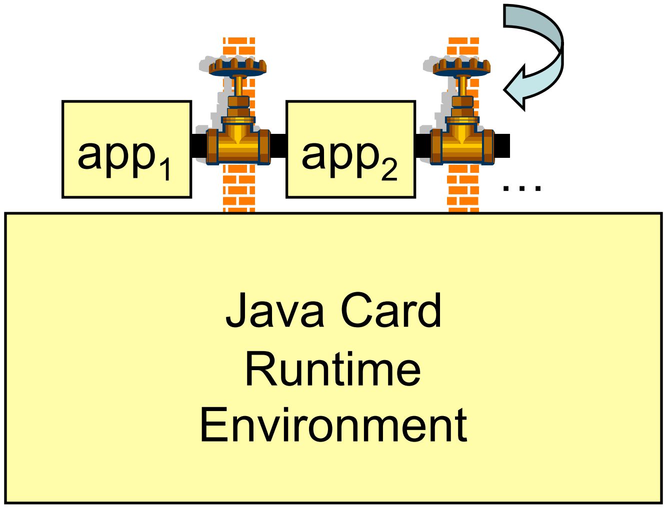
Desired Solution

multi-domain token



enforces information flow policies

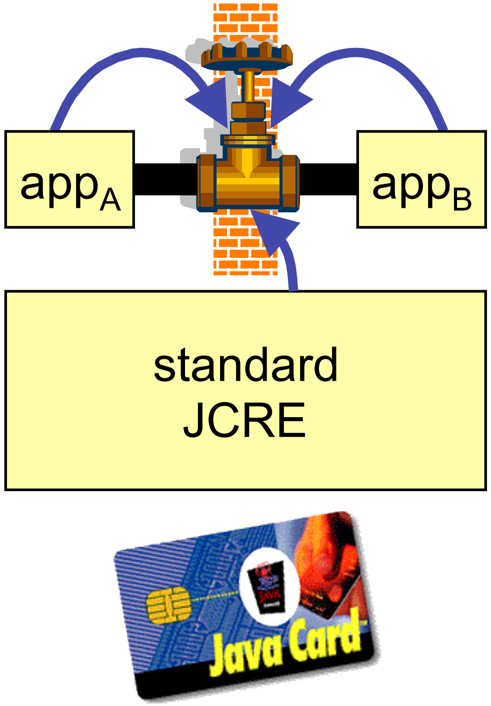
More In General:
Information Flow Policies in Java Card


app$_1$ app$_2$ …

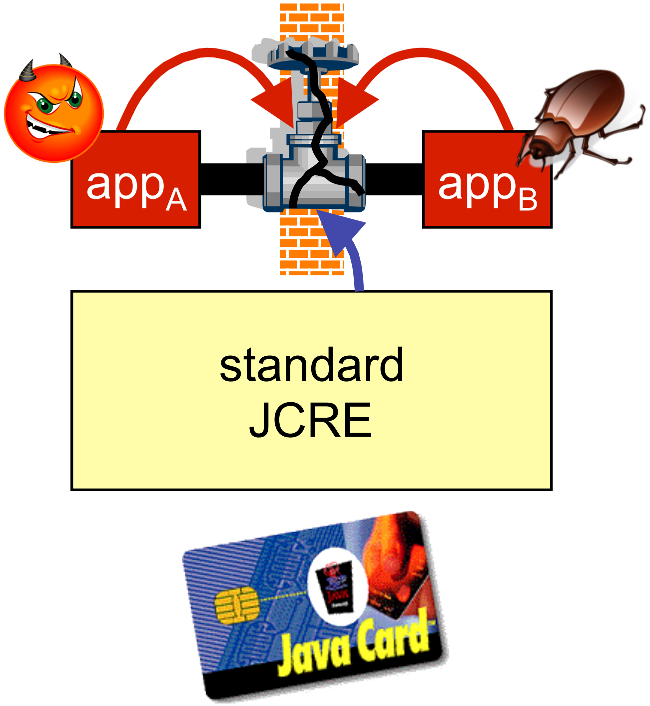
Java Card
Runtime
Environment

Standard Java Card

- Basic protection against undesired information flows
  - type safety ($\Rightarrow$ no buffer overflows)
  - applet firewall (prevents access across objects in different Java packages)
- Insufficient, because
  - two applets can bypass the firewall using
    - static fields and methods (firewall only applies to objects)
    - Shareable Interface Objects (= mechanism for explicit inter-applet communication)
  - discretionary, not mandatory access control
  - Java package boundaries may not align with domain boundaries (e.g. two instances of the same applet may operate on data belonging to different domains)



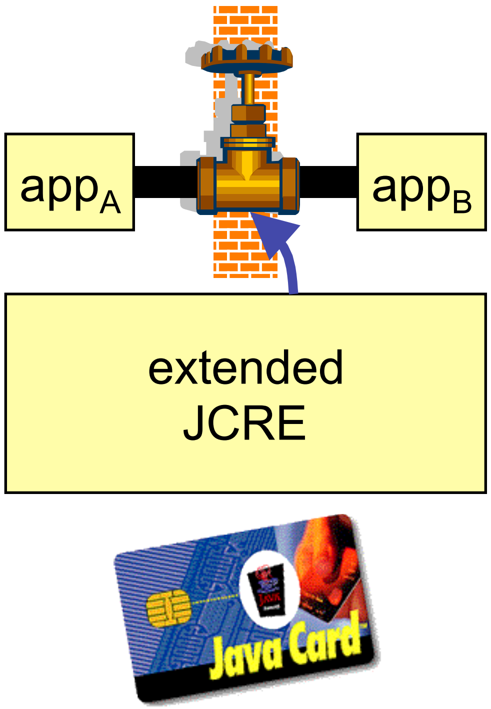
policies enforced by
JCRE + applets
(vs. JCRE alone)



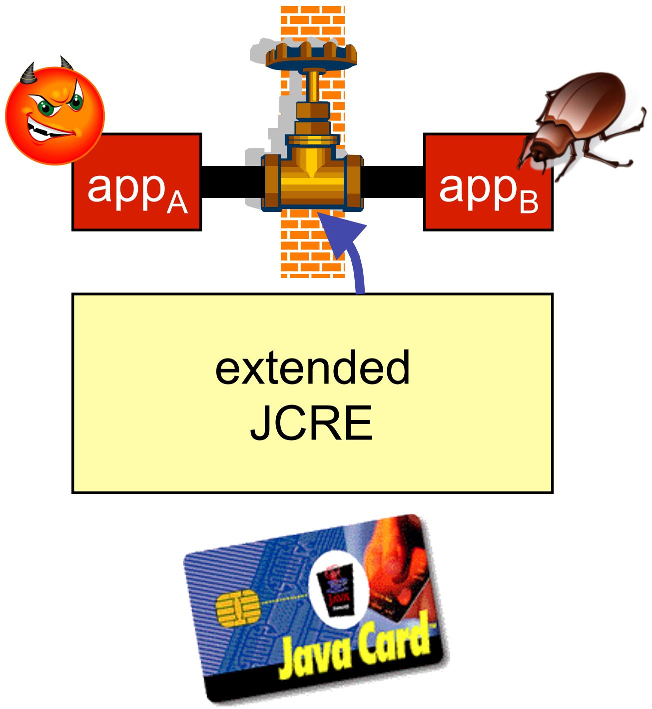
rogue or incorrect
applets may cause
policy violations

Approach: Extend JCRE with Run-Time Policy Monitor

standard JCRE + policy monitor = extended JCRE

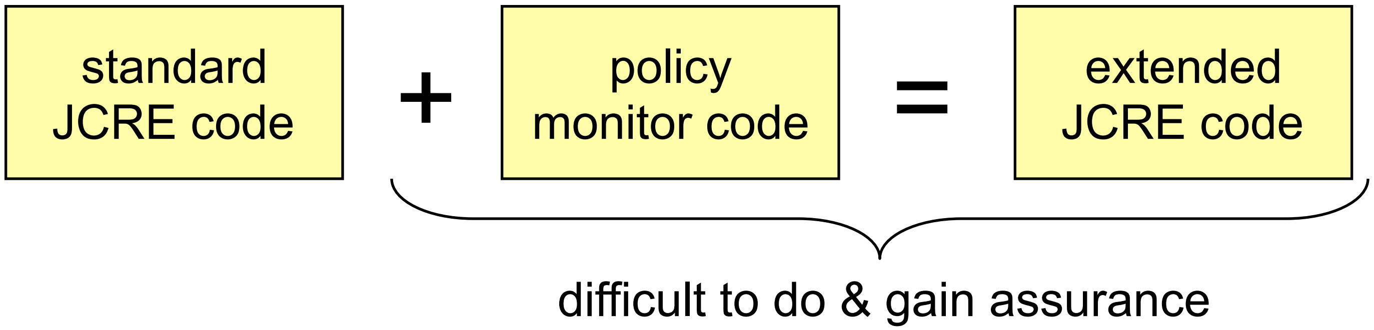


policies enforced by
JCRE alone



rogue or incorrect
applets cannot cause
policy violations

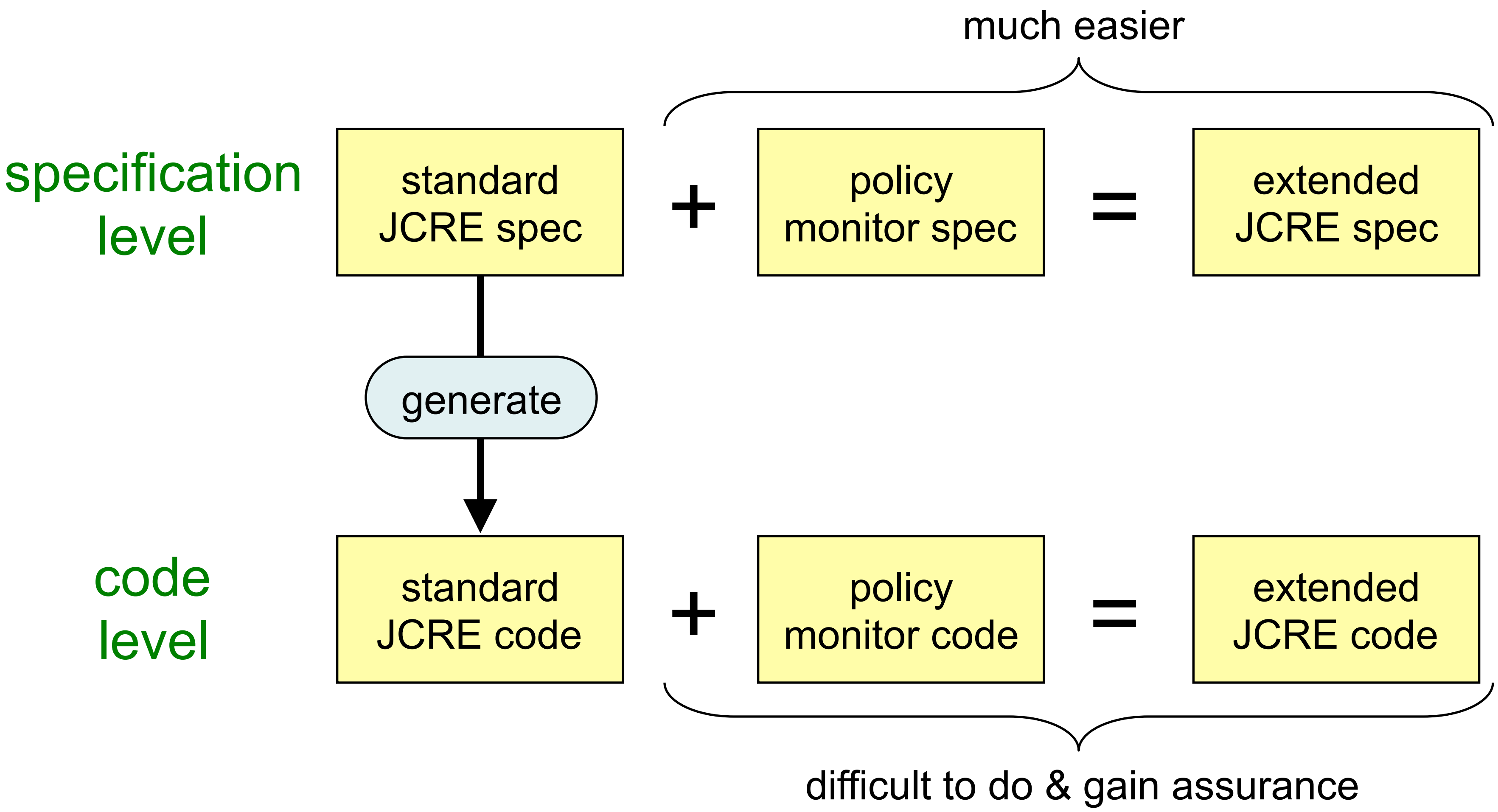
code
level



standard JCRE code + policy monitor code = extended JCRE code

difficult to do & gain assurance

Generative Approach



much easier

specification level

standard JCRE spec + policy monitor spec = extended JCRE spec

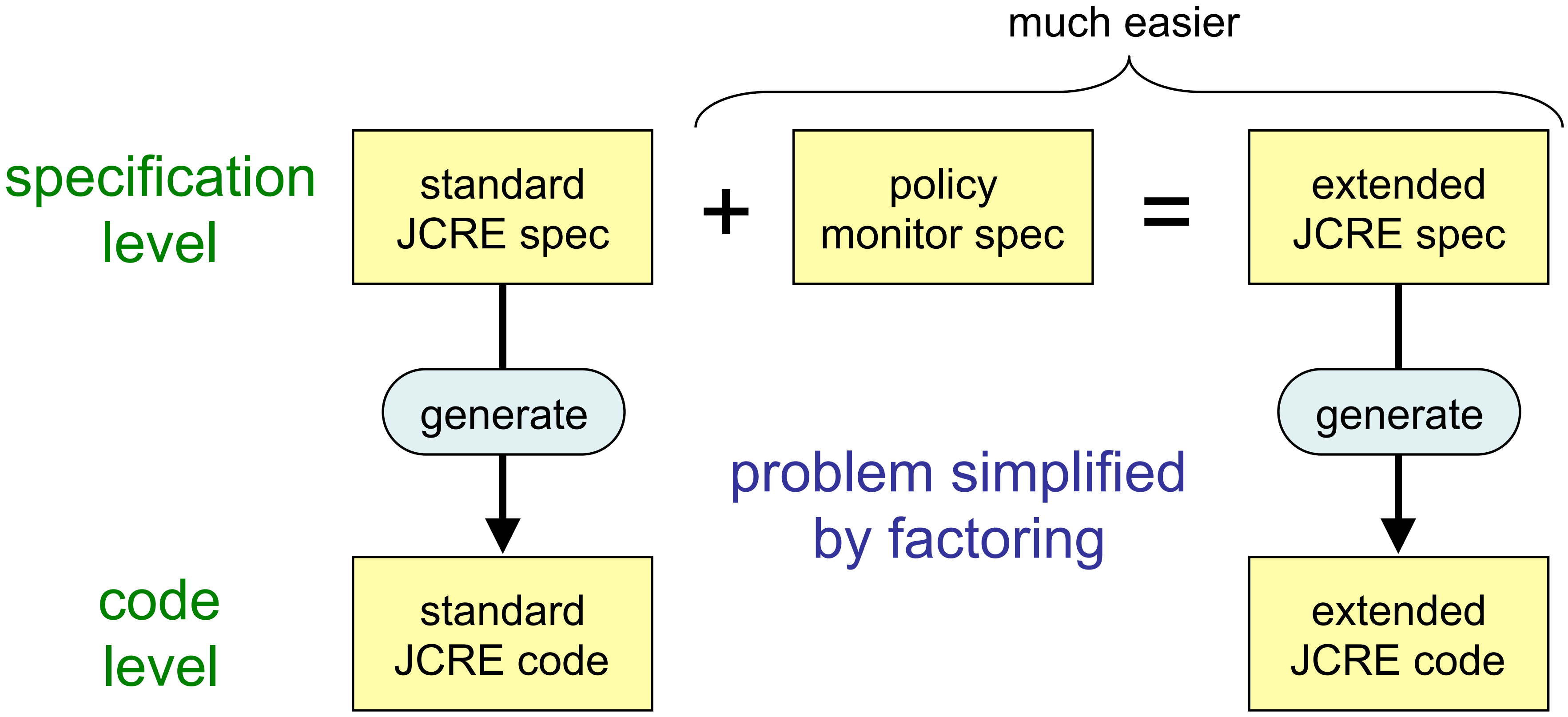
generate

code level

standard JCRE code + policy monitor code = extended JCRE code

difficult to do & gain assurance

Generative Approach



much easier

specification level

standard JCRE spec

\+

policy monitor spec

=

extended JCRE spec

generate

generate

problem simplified by factoring

code level

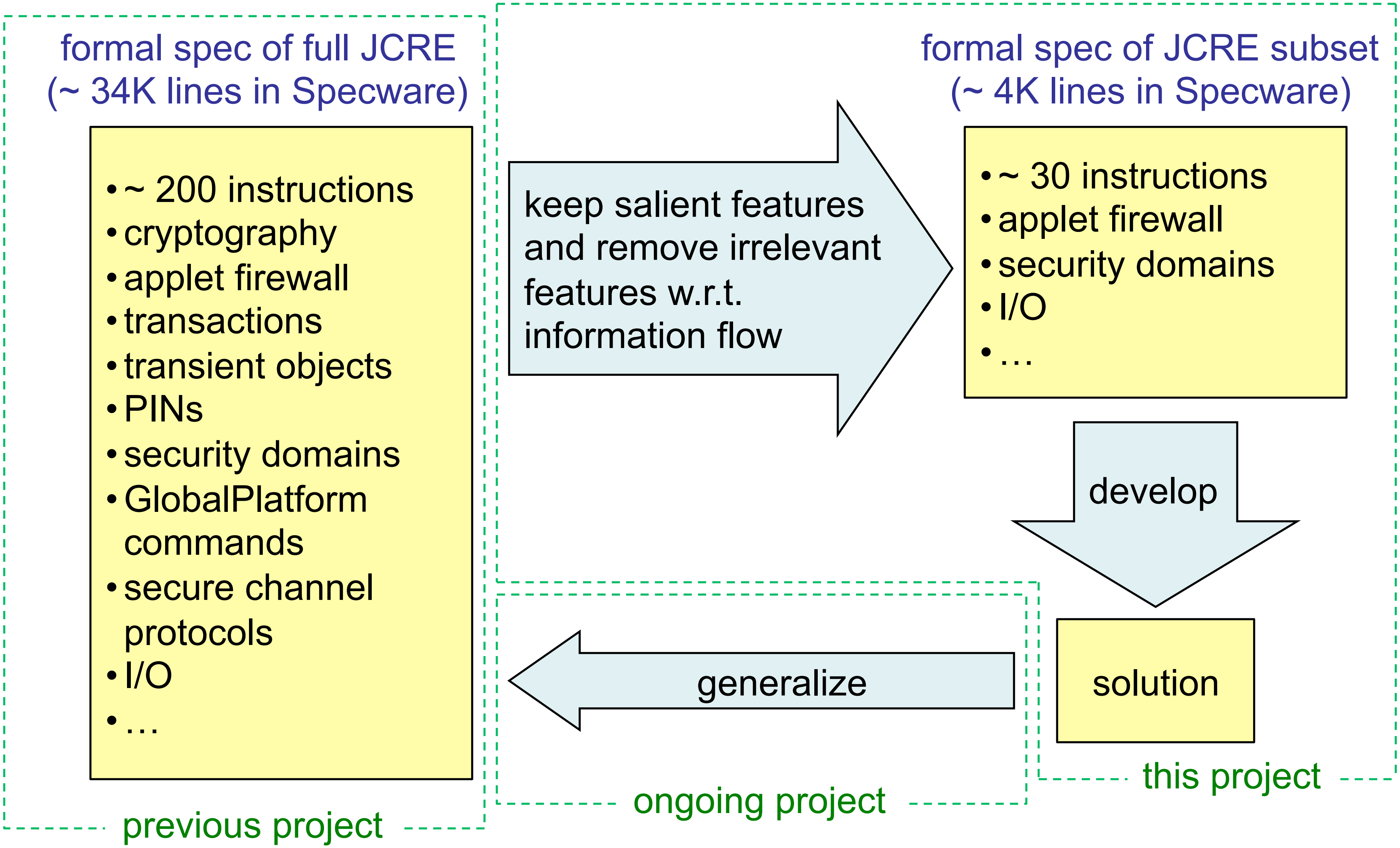
standard JCRE code

extended JCRE code

Specware

- Kestrel's main tool for generative development
- Specifications written in higher-order logic
- Refinement
  - automated via proof-generating transformations
  - manual with proof obligations
- Interfaces to theorem provers (e.g. Isabelle/ HOL)
- Automatic code generation for subset of specification language

Leverage among 3 Projects



formal spec of full JCRE
(~ 34K lines in Specware)

- ~ 200 instructions
- cryptography
- applet firewall
- transactions
- transient objects
- PINs
- security domains
- GlobalPlatform commands
- secure channel protocols
- I/O
- …

previous project

keep salient features and remove irrelevant features w.r.t. information flow

formal spec of JCRE subset
(~ 4K lines in Specware)

- ~ 30 instructions
- applet firewall
- security domains
- I/O
- …

develop

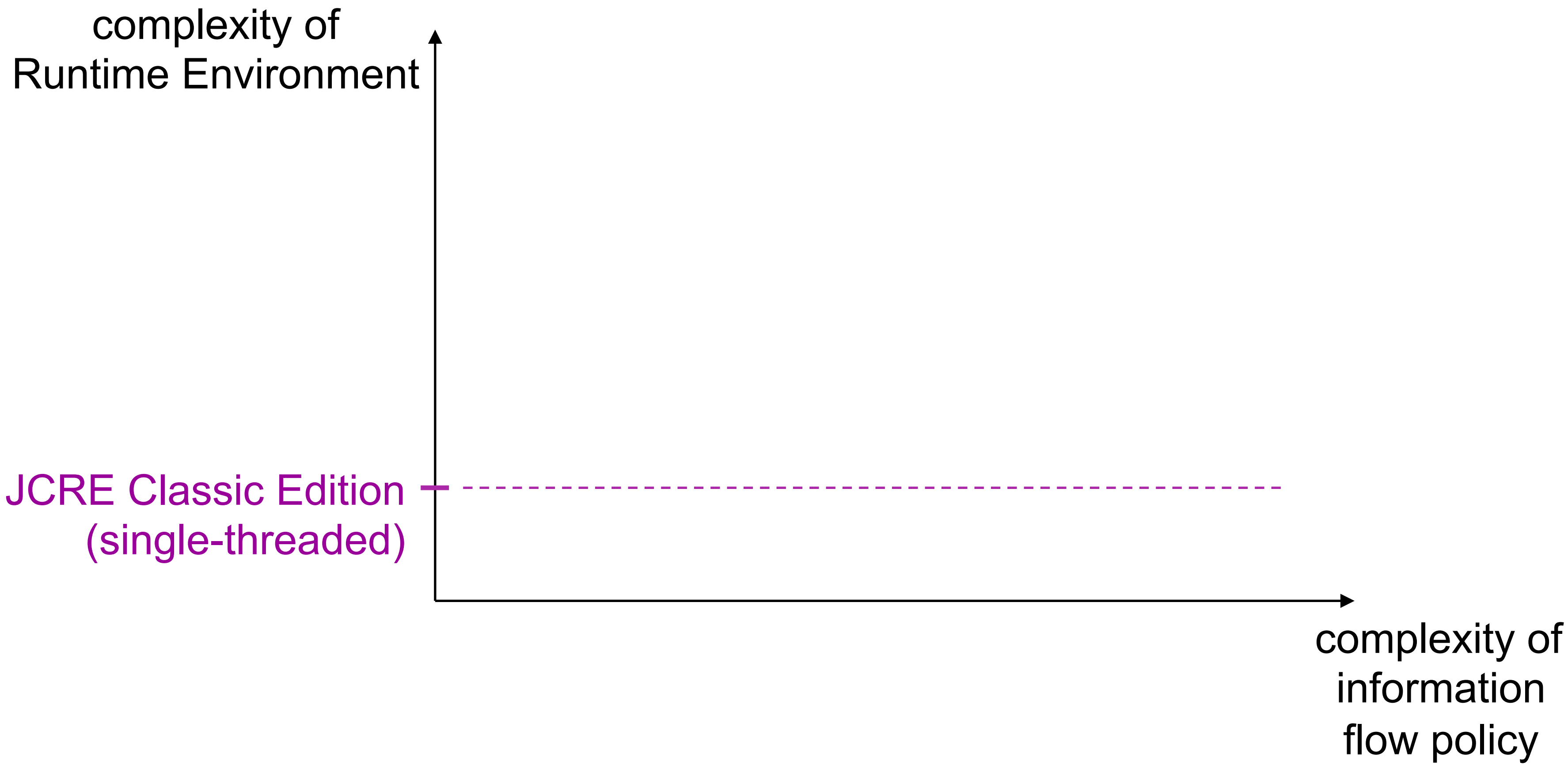
solution

this project

generalize

ongoing project

Space of Exploration of This Project

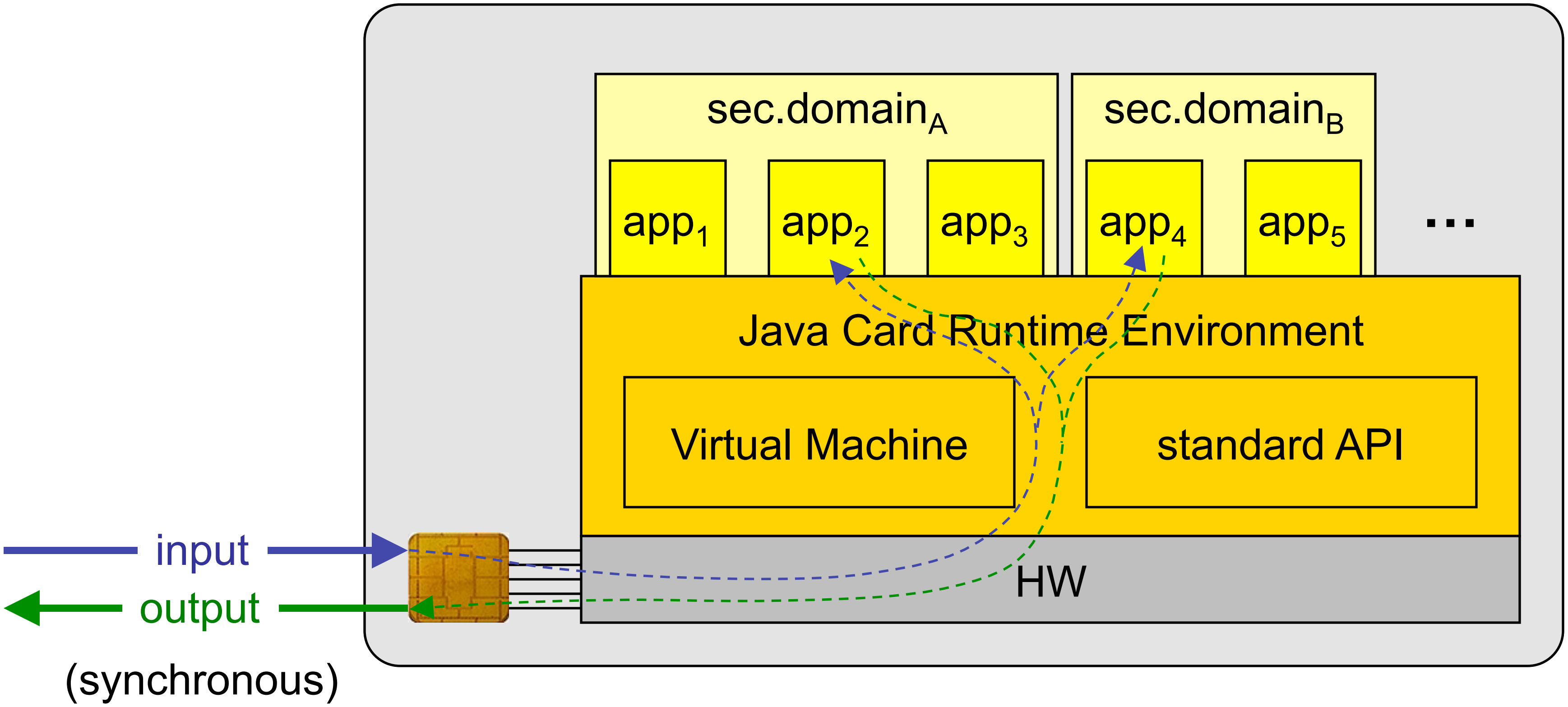


complexity of
Runtime Environment

JCRE Classic Edition
(single-threaded)

complexity of
information
flow policy

JCRE Classic Edition



inputs/outputs dispatched by JCRE to/from applets

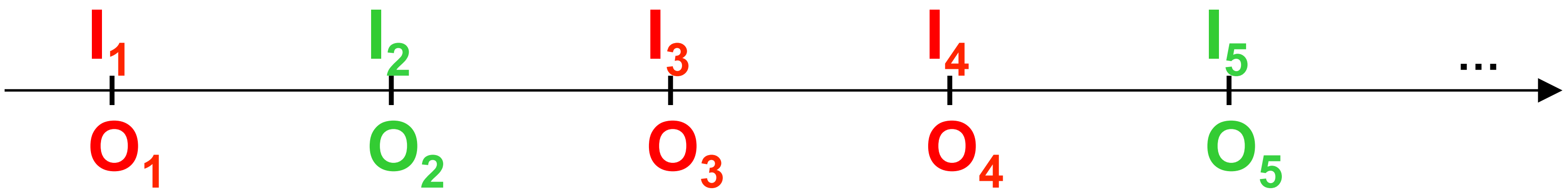
applets partitioned into security domains

JCRE in Specware

```
% observables:
type Input = ...
type Output = ...
type Exchange = {in:Input, out:Output}
type Trace = Seq Exchange
type SecurityDomain = ...
op domainOf : Input -> SecurityDomain = ...
op domainOf(exch:Exchange):SecurityDomain = domainOf exch.in
```

trace of exchanges:

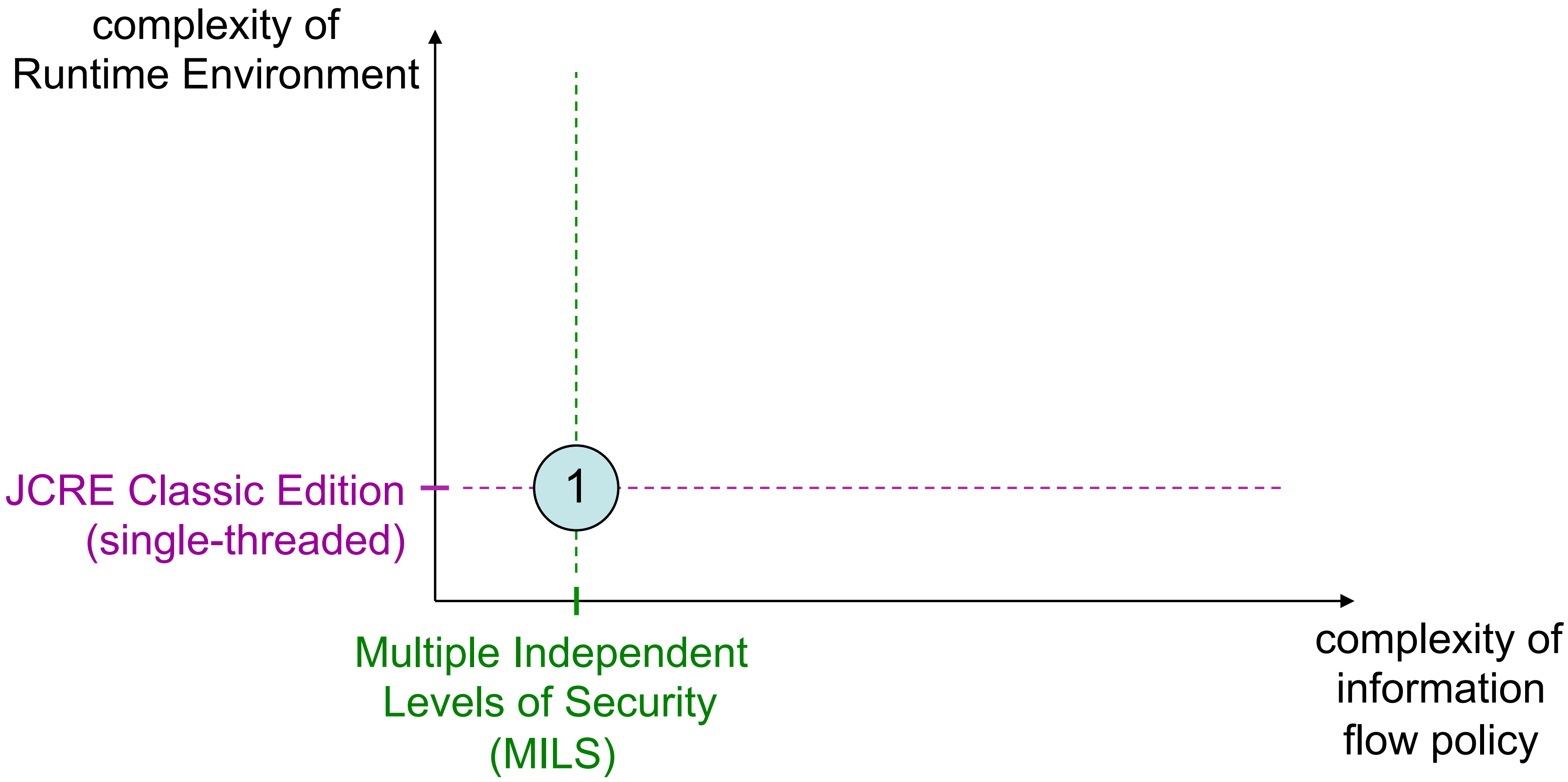
sec.domain$_A$
sec.domain$_B$



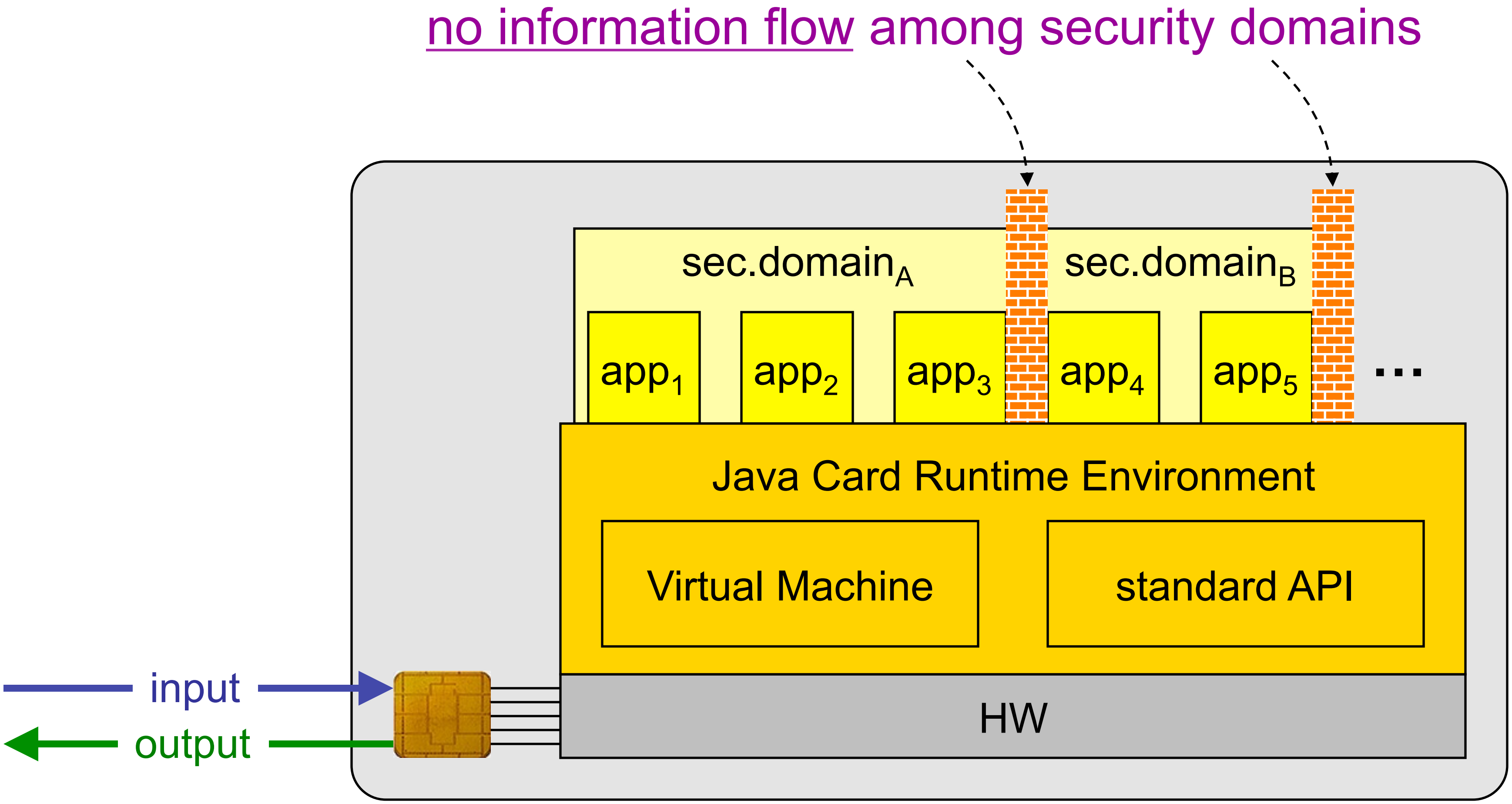
JCRE in Specware

```
% observables:
...
type Exchange = {in:Input, out:Output}
type Trace = Seq Exchange
...



% standard JCRE's set of traces:
...
type Object = | clinst ClassInstance | array Array
type Heap = Set Object
type Frame = Method * ProgramCounter * ...
type State = Heap * (Seq Frame) * ...
op initState : State = ...   % includes installed applets
op step : State -> State = ...
op process : Input * State -> Output * State = ...
op standardTraces : Set Trace = ... process ...
```

Space of Exploration



MILS Policy



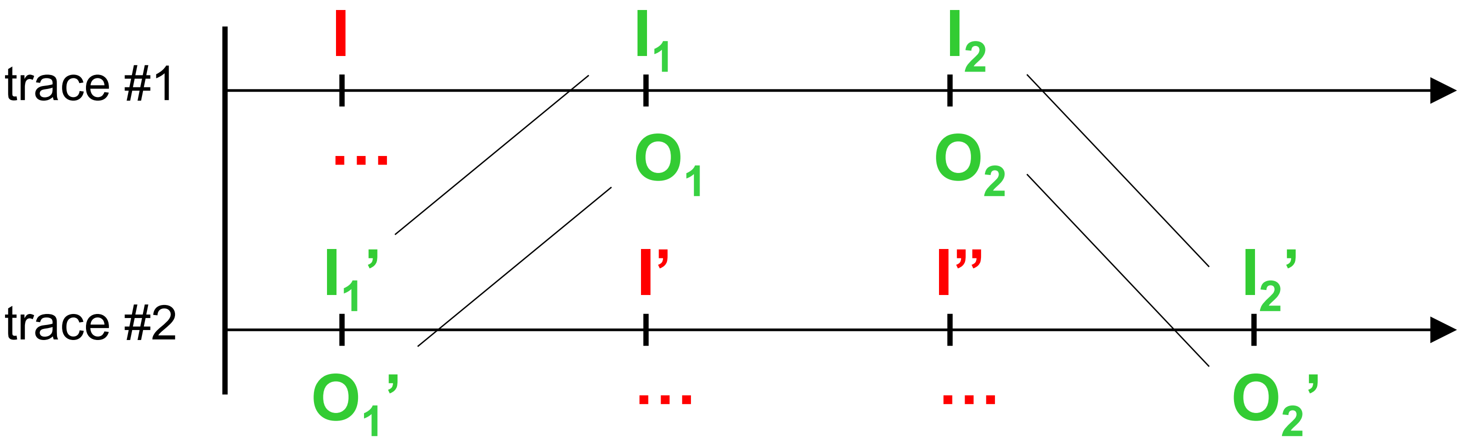
no information flow among security domains

Scope of our MILS Policy

- Only inputs & outputs are observable
- Internal state is not directly observable
  - only indirectly via I/O exchanges
  - smart cards have HW protections against direct physical access to internal memory
- Under these assumptions, policy is expressed in terms of I/O exchanges only
- Policy, intuitively: running a security domain together with other domains yields the same results as running that domain alone

MILS Policy Graphically

sec.domain_A
sec.domain_B



trace #1

I … I_1 O_1 I_2 O_2

trace #2

I_1' O_1' I' … I'' … I_2' O_2'

$$I_1 = I_1' \;\wedge\; I_2 = I_2' \;\Rightarrow\; O_1 = O_1' \;\wedge\; O_2 = O_2'$$

(regardless of I, I', I'', …)

MILS Policy in Specware

```
% non-interference predicate over sets of traces:
op satisfiesMILS? (TRS:Set Trace) : Boolean =
  % given a security domain and two traces:
  fa (sd:SecDomain, tr1:Trace, tr2:Trace)
    tr1 in? TRS && tr2 in? TRS &&
  % extract the inputs and outputs for the domain:
    (let subtr1 = filterTrace sd tr1 in
     let subtr2 = filterTrace sd tr2 in
  % if the inputs coincide:
    mapSeq (project in) subtr1 =
    mapSeq (project in) subtr2 =>
  % then the outputs must coincide:
    mapSeq (project out) subtr1 =
    mapSeq (project out) subtr2
```

MILS Monitor

runtime monitor sees only current trace

$\Downarrow$

it cannot directly check MILS policy predicate,
which is over sets of traces

$\Downarrow$

we find stronger policy predicate over single traces
(more than one choice possible)

$\Downarrow$

monitor enforces the stronger policy

$\Downarrow$

MILS policy is satisfied

MILS Monitor

- Several choices possible
  - block illicit information flow sooner vs. later
  - corrective action could throw exception vs. turn attempt into no-op
- Our choice
  - block attempts to access static fields across domains
  - block attempts to obtain shareable objects across domains
  - throw security exception if any of these attempts take place

MILS Monitor in Specware

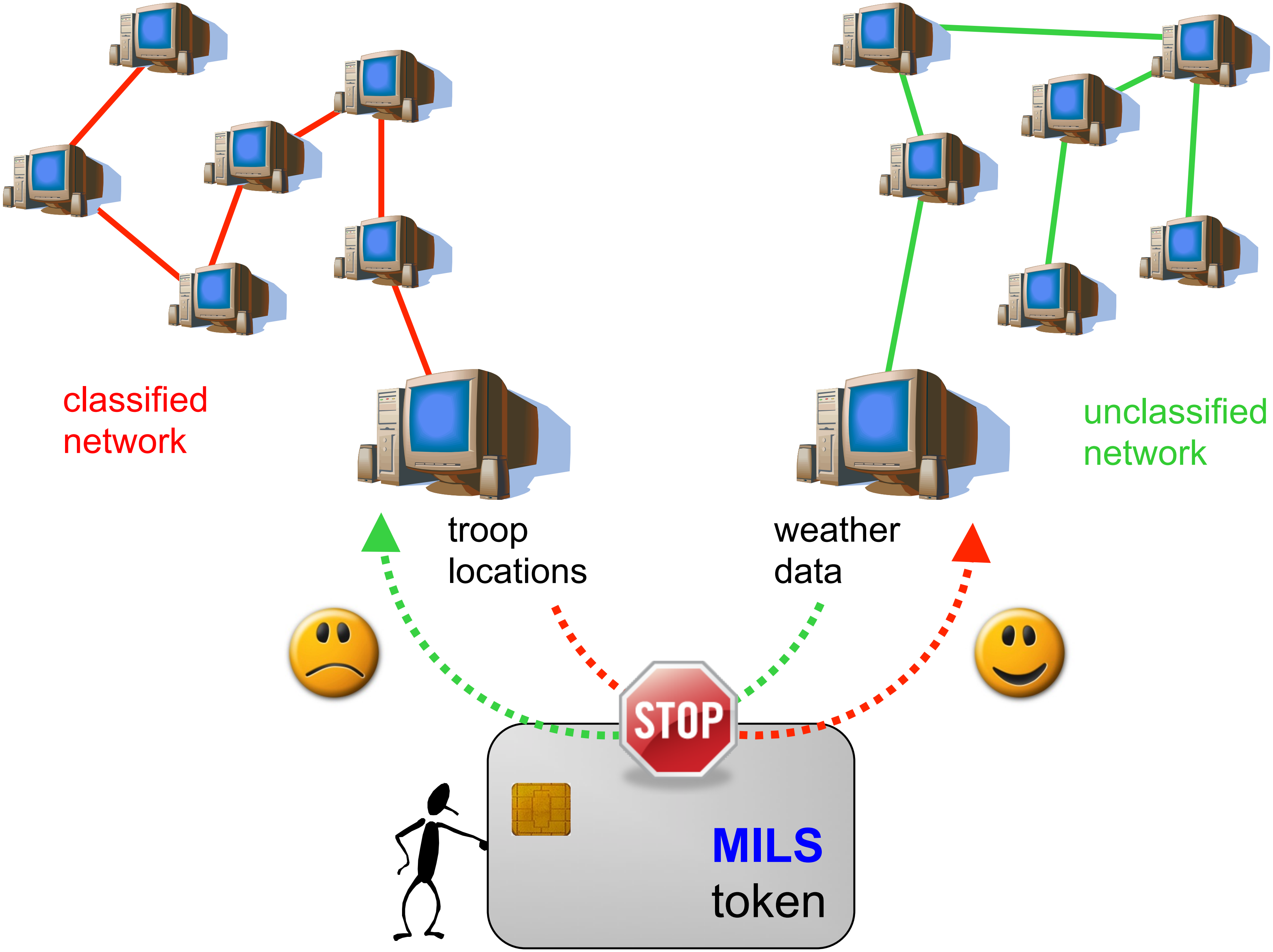
```
% standard JCRE’s set of traces:
...
op step : State -> State = ...
op standardTraces : Set Trace = ... step ...


% recognize operations that may transfer info cross-domain:
op violates? : State -> Boolean = ...
% define corrective action (e.g. throw Java exception):
op correct : (State | violates?) -> State = ...


% compose monitor with standard JCRE:
op step’(st:State):State = if violates? st then correct st
                           else step st  % do as usual
op monitoredTraces: Set Trace = ... step’ ...
% stronger monitoring policy implies MILS policy:
theorem monitor_enforces_MILS is
  satisfiesMILS? monitoredTraces
```

Proof that Monitor Guarantees MILS

- Partitioning of state by domains
- Closure: pointers in each partition reference only objects in the same partition
  - proved by cases on all possible execution steps (every bytecode, every API call, etc.)
  - run-time monitor curbs execution steps that would break closure
- Thus, execution step in a partition does not change, and is not affected by, other partitions
- Thus, two traces with the same inputs to a domain yield “parallel” executions w.r.t. the domain
  - equivalent sub-states (i.e. partitions)
  - in particular, same outputs

Proof that Monitor Guarantees MILS

- Some tricky bits
  - sub-state equivalence is modulo consistent pointer renaming
  - I/O buffer shared among domains
    - but OK because zeroed before each new I/O exchange
  - exception object shared among domains
    - not zeroed before each new I/O exchange
    - but only way to reference it is via an API that overwrites its content, thus destroying any value stored there belonging to other domains
- The fact that the main theorem is proved, means that our run-time monitor does not miss any case (if it did, the theorem could not be proved)



classified
network

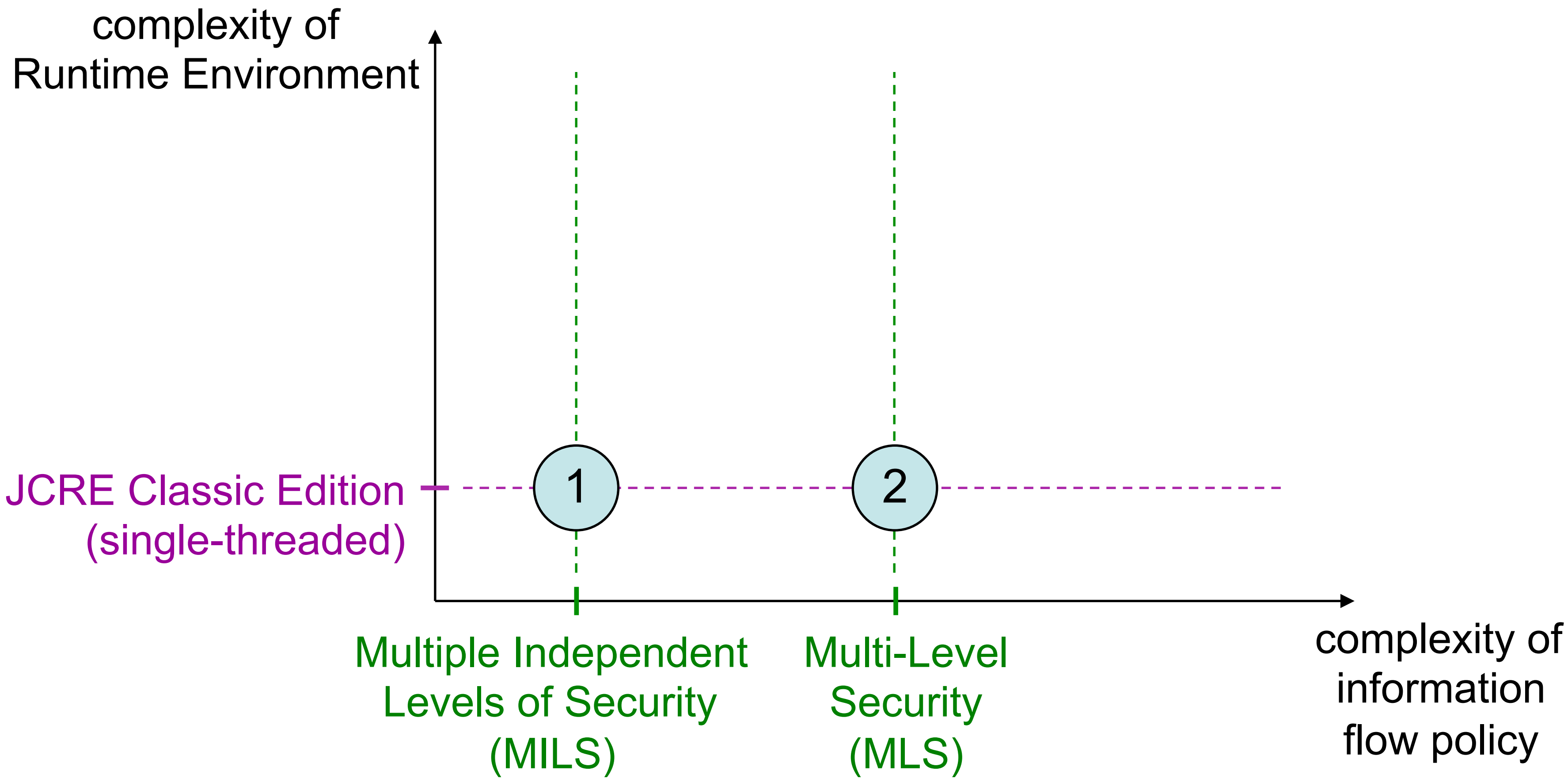
unclassified
network

troop
locations

weather
data

MILS
token

Space of Exploration



MLS vs. MILS

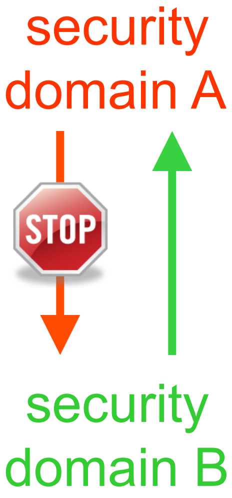
MILS

security domain A

security domain B

symmetric

MLS



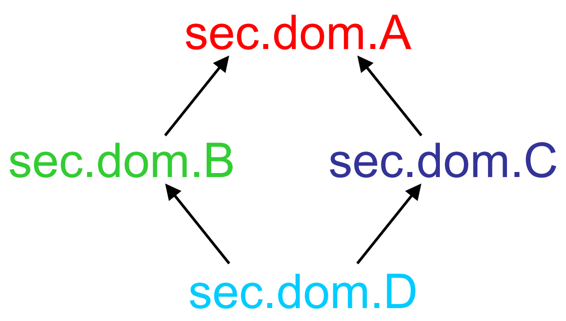
security domain A

security domain B

partial order

MLS Policy

partial order:

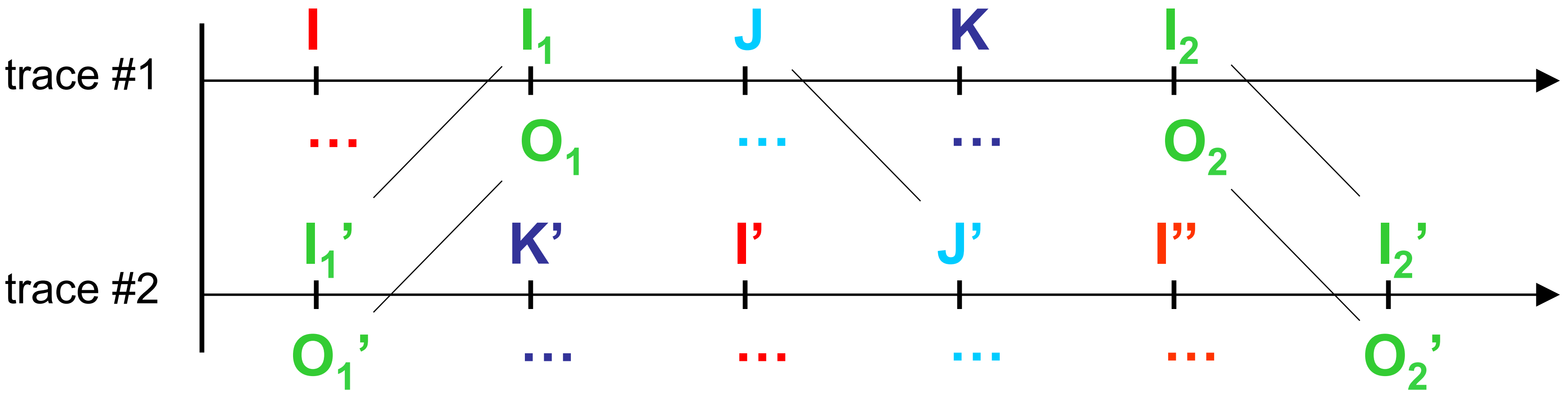


sec.dom.A

sec.dom.B

sec.dom.C

sec.dom.D

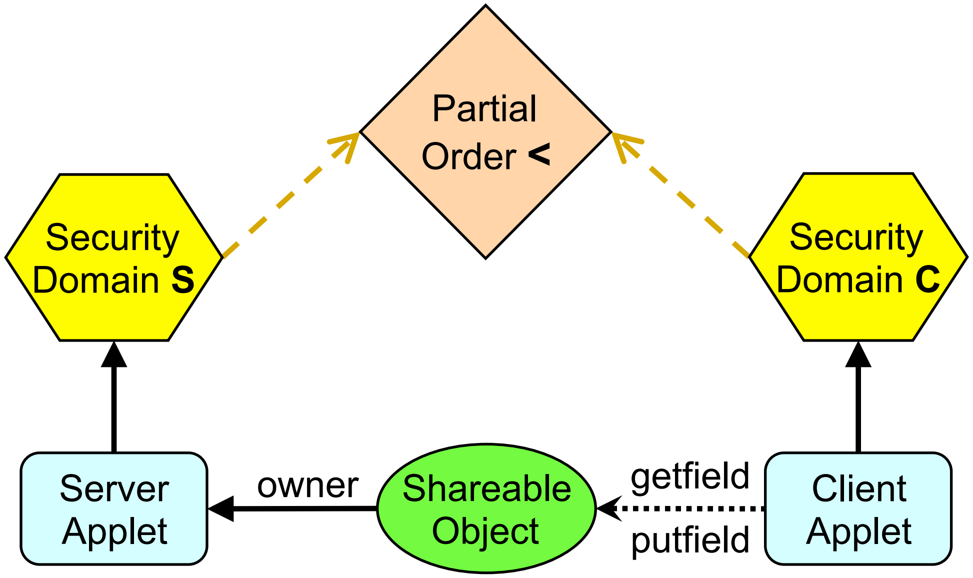


$$I_1 = I_1' \;\wedge\; J = J' \;\wedge\; I_2 = I_2' \;\Rightarrow\; O_1 = O_1' \;\wedge\; O_2 = O_2'$$

(regardless of **I**, **I'**, **I''**, **K**, **K'**, …)

MLS Monitor

- 2 possible ways to share information across domains
  - instance fields in shared objects
  - static fields
- MLS monitor blocks operations that would violate policy



Partial Order **<**

Security Domain **S**

Security Domain **C**

Server Applet

owner

Shareable Object

getfield

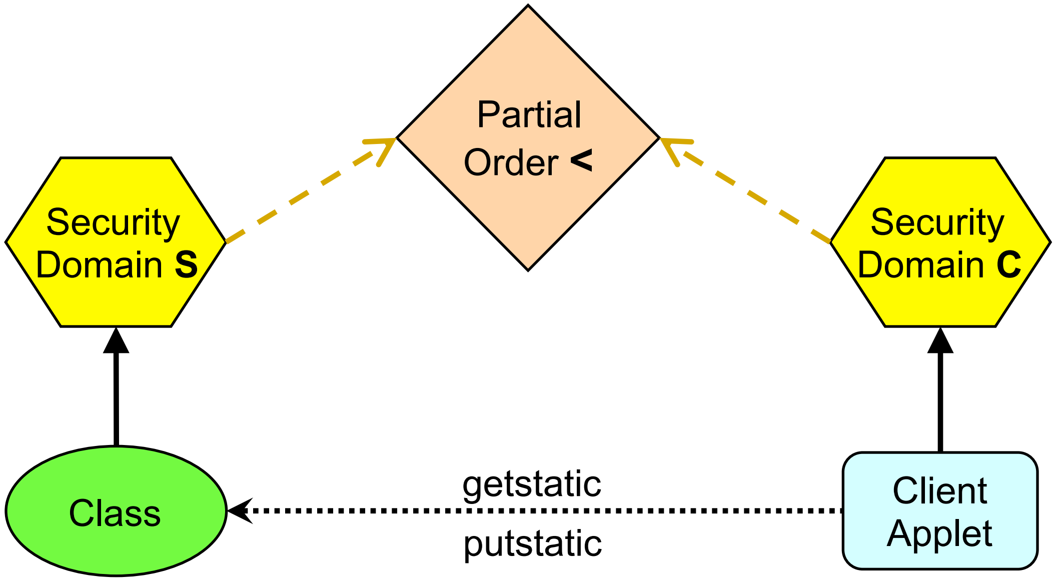
putfield

Client Applet

Instance Field

	get field	put field
C < S	X	✓
C = S	✓	✓
S < C	✓	X
C & S incomparable	X	X

MLS Monitor



Partial Order **<**

Security Domain **S**

Security Domain **C**

Class

getstatic

putstatic

Client Applet

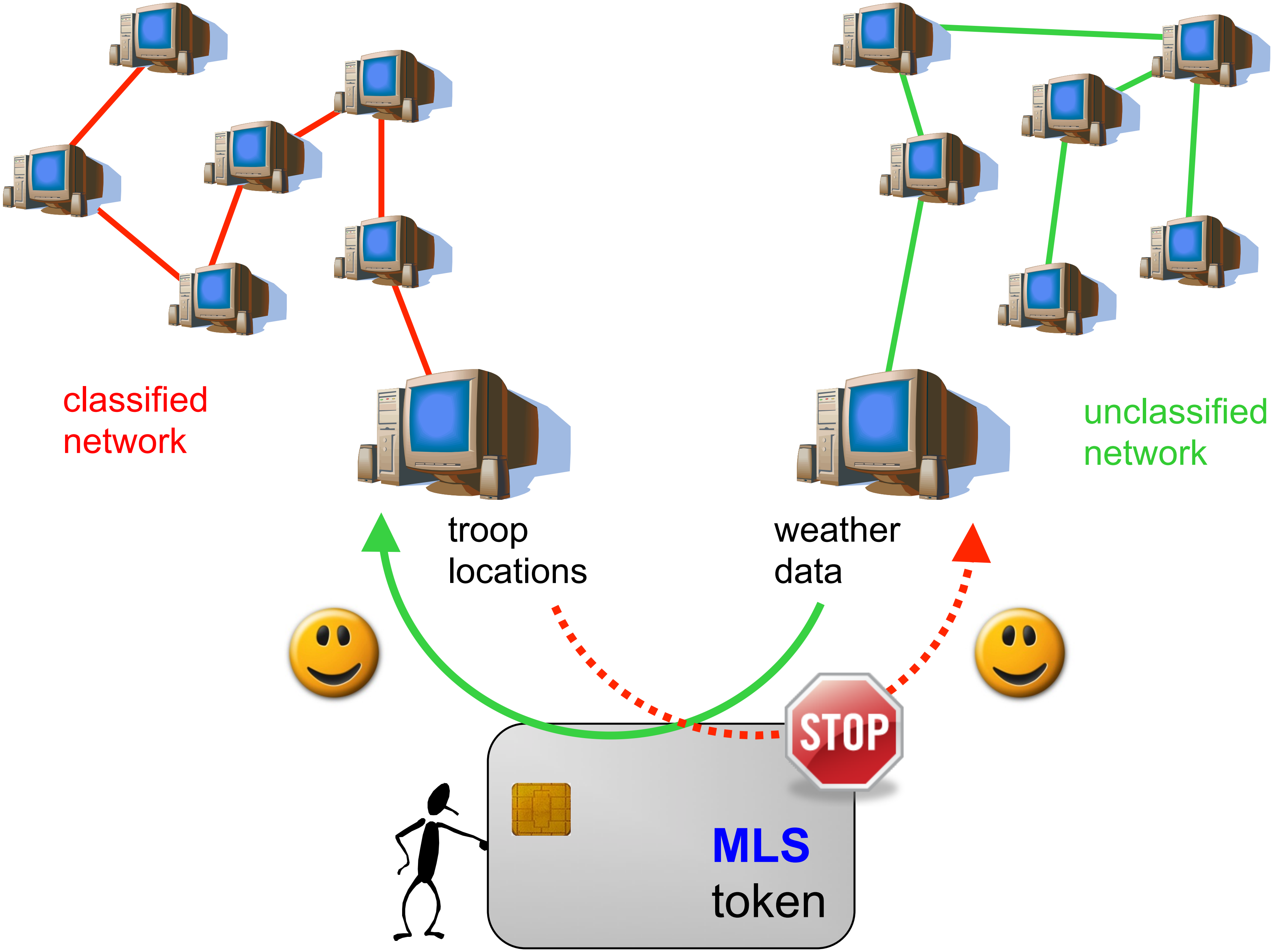
	getstatic	putstatic
C < S	X	✓
C = S	✓	✓
S < C	✓	X
C & S incomparable	X	X

(same table as for instance field)

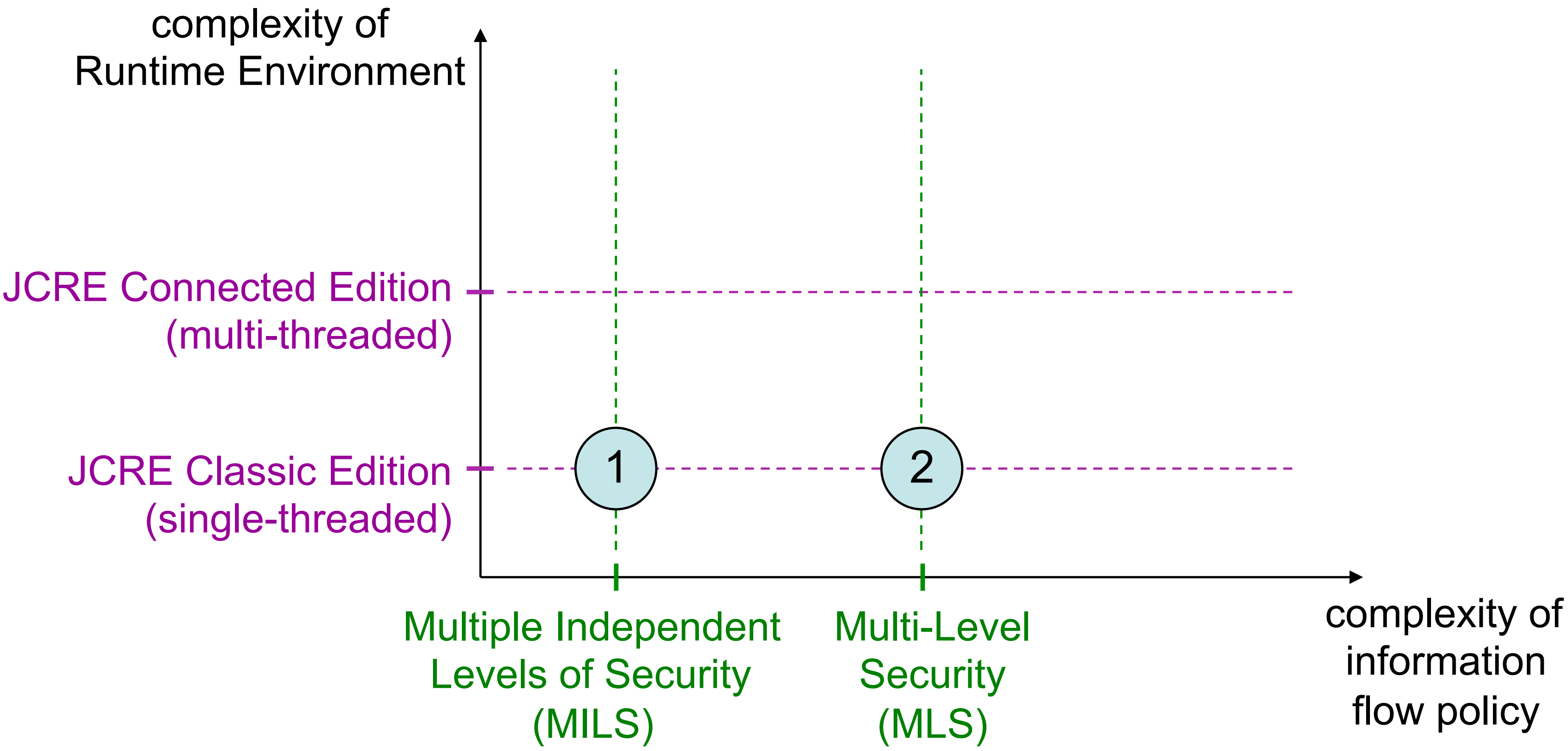
Static Field

Proof that Monitor Guarantees MLS

- Analogous to proof for MILS
- Weaker notions and invariants, e.g.
  - closure of pointers in each domain & lower domains (no references to higher or incomparable domains)
  - execution step in a domain
    - does not change lower or incomparable domains
    - is not affected by higher or incomparable domains

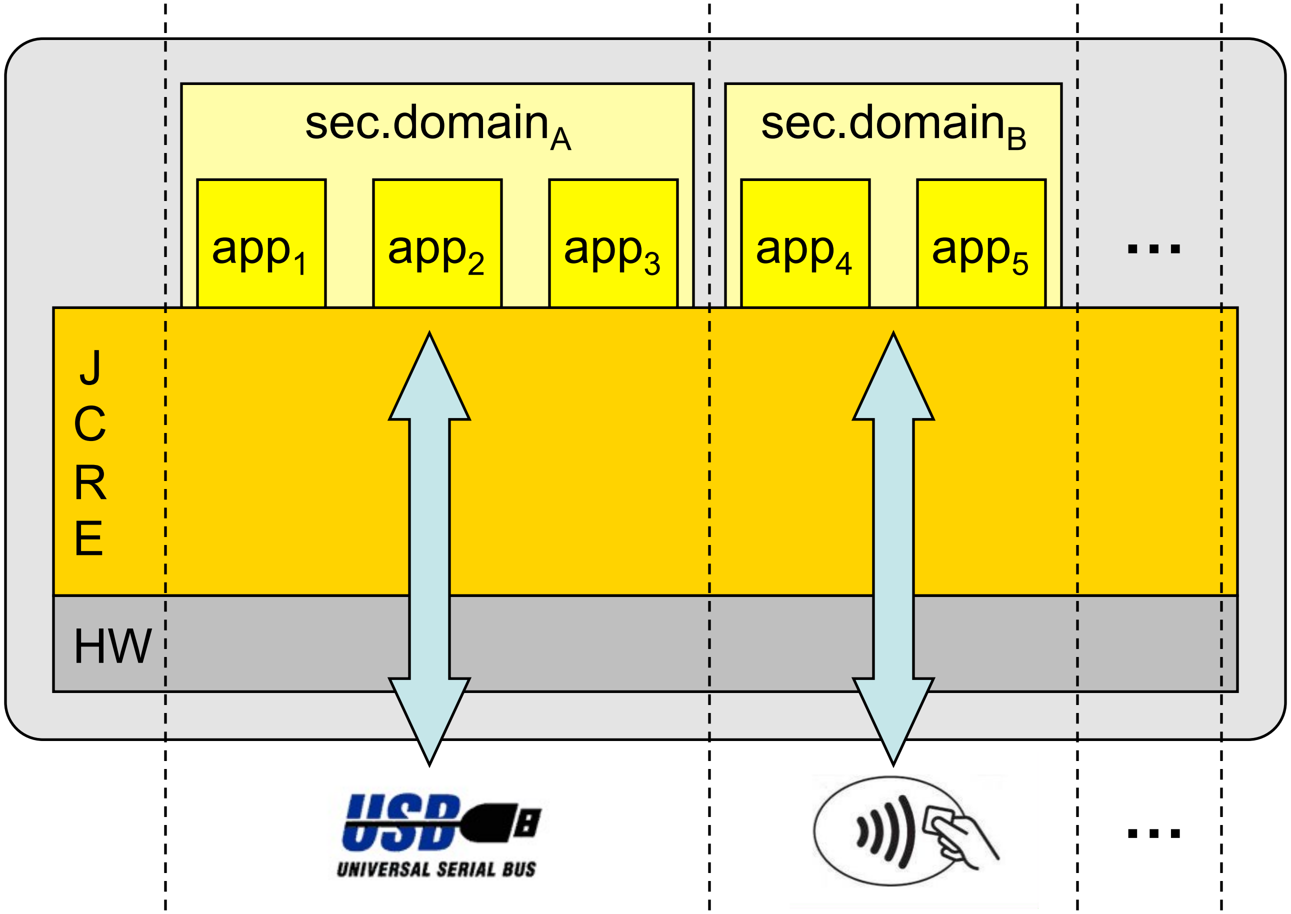

classified
network
unclassified
network
troop
locations
weather
data
STOP
MLS
token

Space of Exploration



Multi-Threading Model

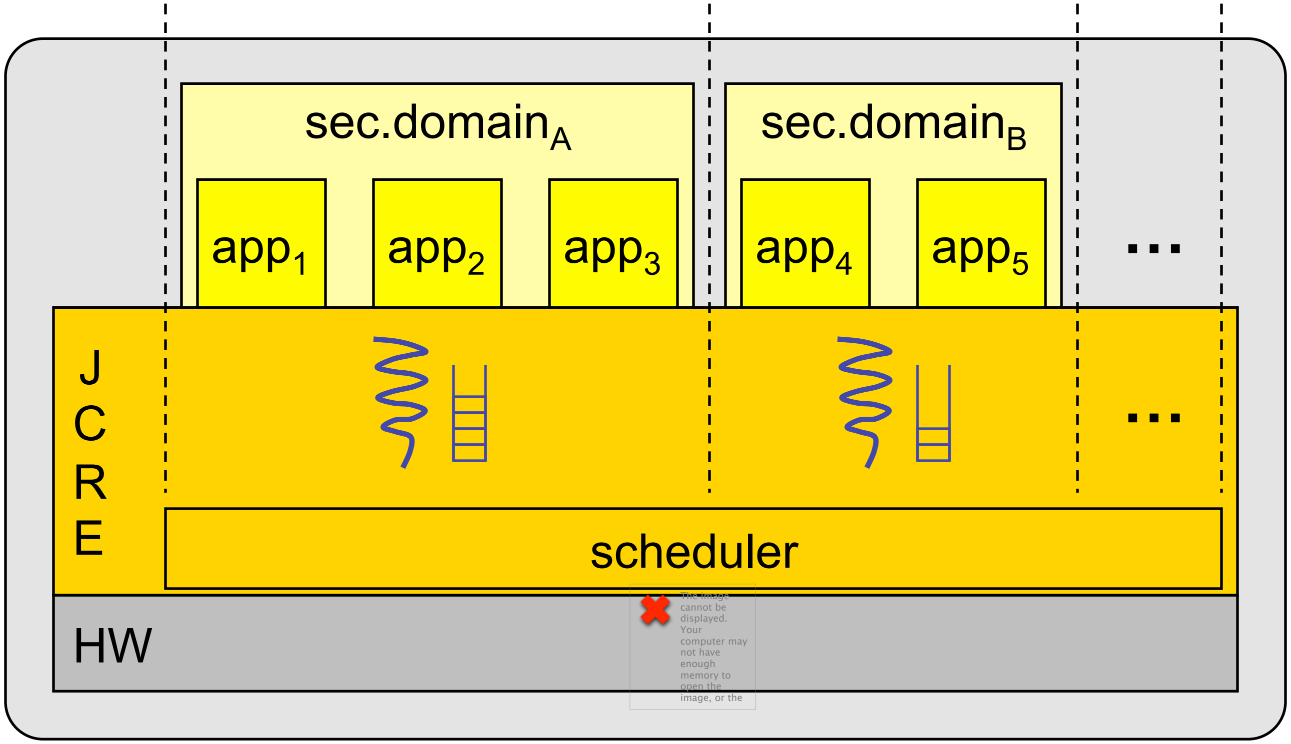
one I/O interface per security domain



I/O interfaces operate in parallel, independently

Multi-Threading Model

one thread per security domain



scheduler interleaves threads' execution steps

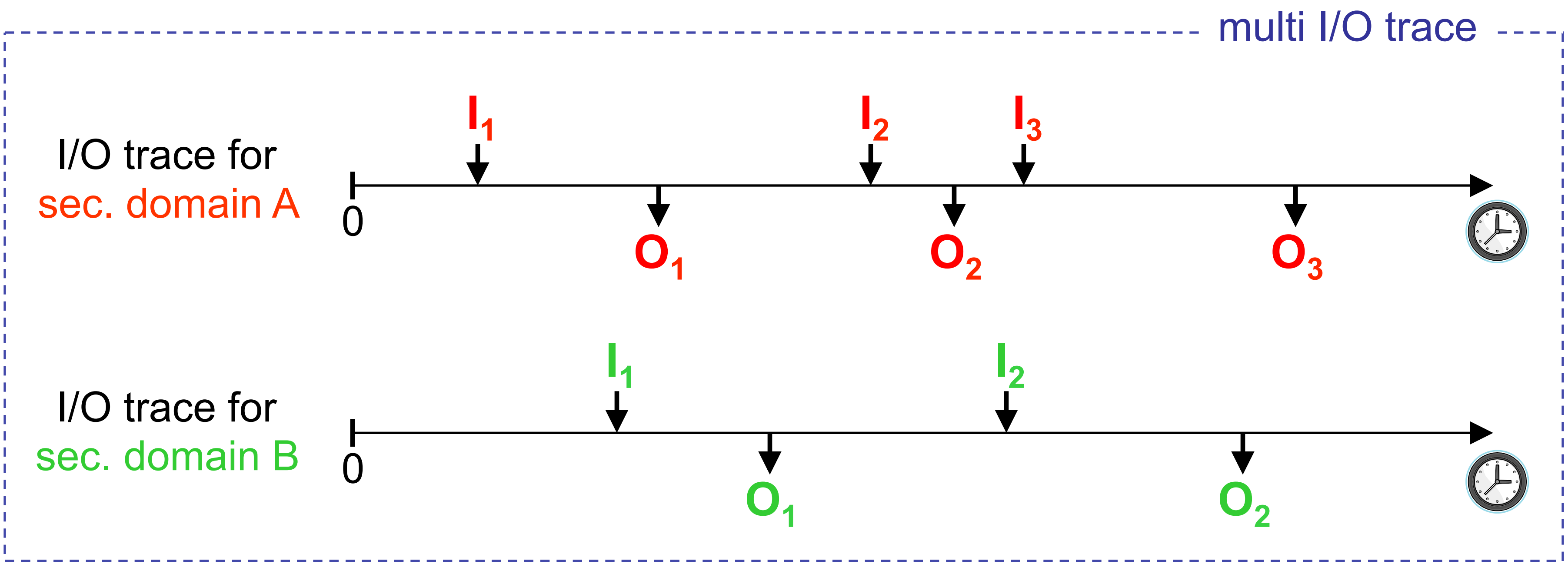
⇒ potential for <u>timing channels</u>

Multi-Threading Model

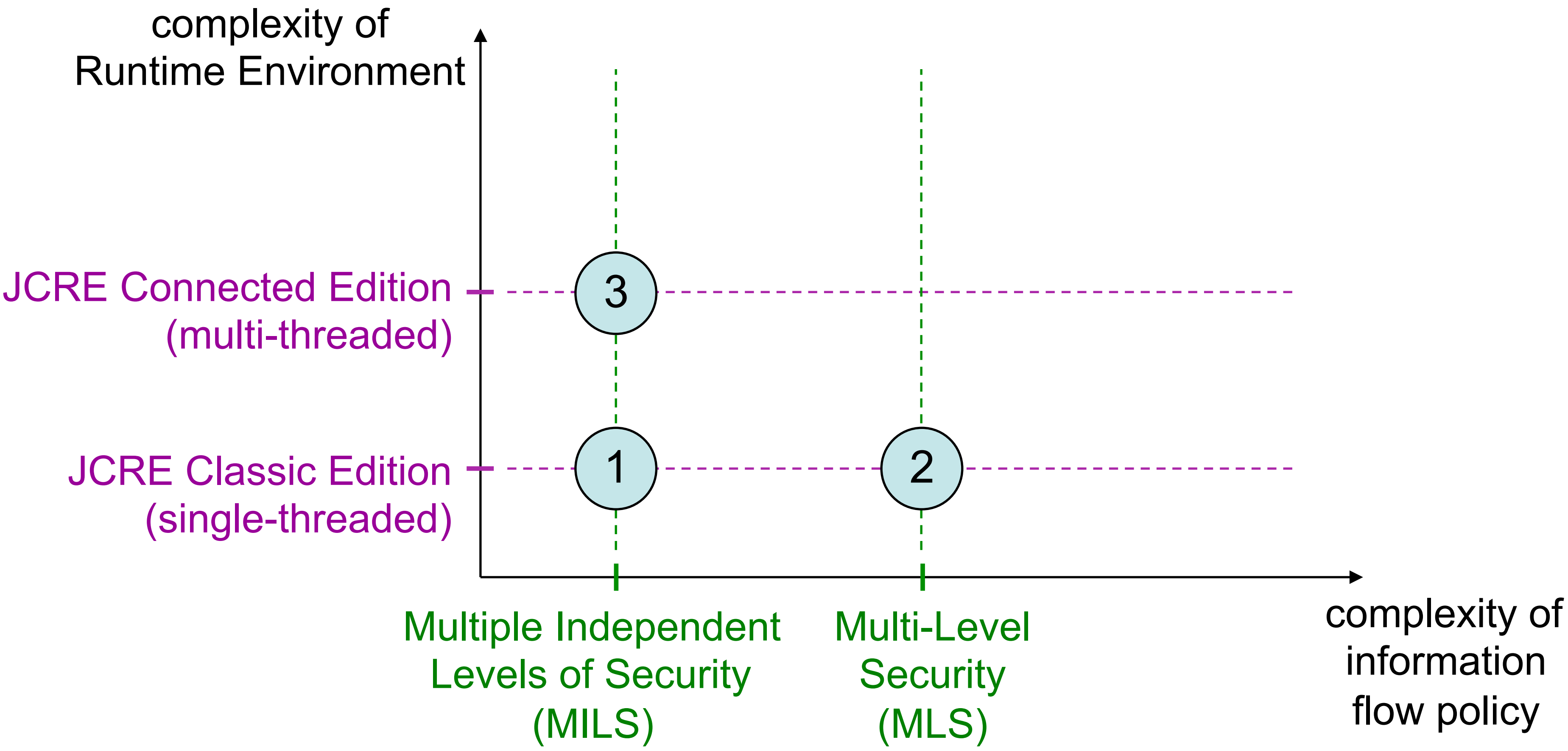
- Simple, but exhibits salient features (e.g. potential for timing channels)
- Consistent with Java Card Connected Edition
- Parameterized over scheduling policy
  - scheduling policy may affect information flow (e.g. domain A may preempt domain B)

Multi-Threading Model in Specware

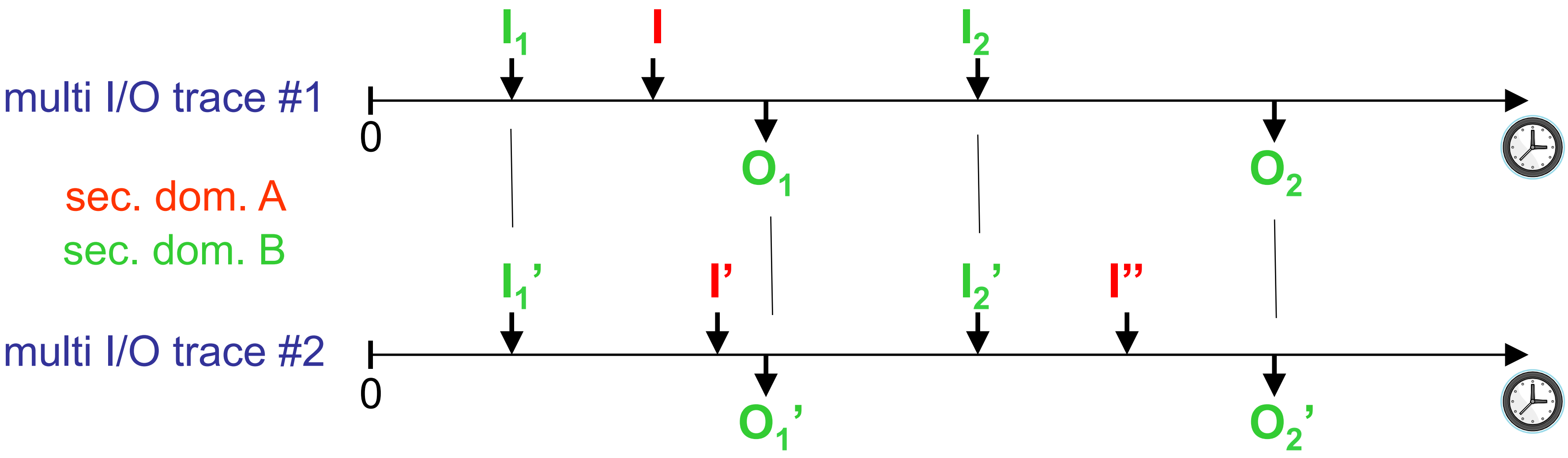
```
% observables include time:
type Input = ...
type Output = ...
type Time = NonNegReal
type IOTrace = Map (Time, (Input | Output))
type SecurityDomain = ...
type MultiIOTrace = Map (SecurityDomain, IOTrace)
```



Space of Exploration



MILS Policy that Includes Time



$$I_1 = I_1' \quad \wedge \quad I_2 = I_2' \quad \wedge \quad t(I_1) = t(I_1') \quad \wedge \quad t(I_2) = t(I_2')$$

$$\Rightarrow$$

$$\boxed{O_1 = O_1' \quad \wedge \quad O_2 = O_2'} \quad \wedge \quad \boxed{t(O_1) = t(O_1') \quad \wedge \quad t(O_2) = t(O_2')}$$

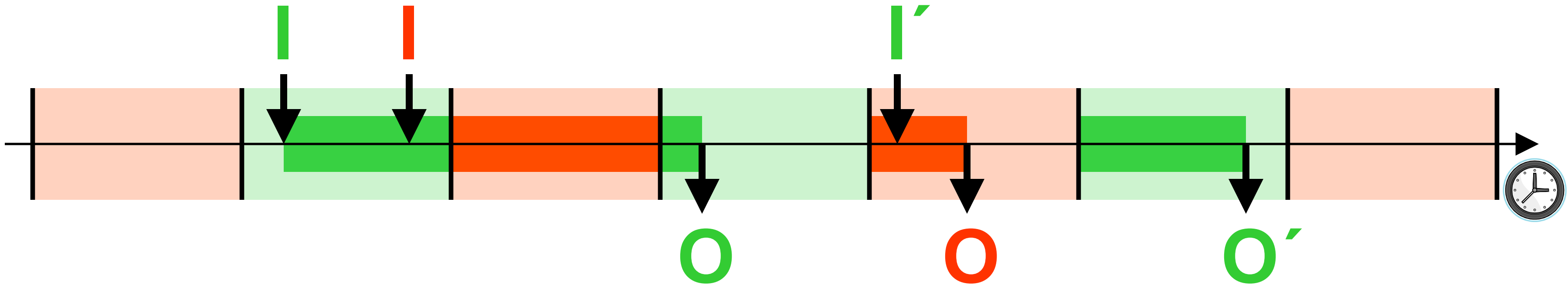
no explicit channels

no timing channels

(regardless of I, I’, I”, …)

MILS Monitor + Scheduler

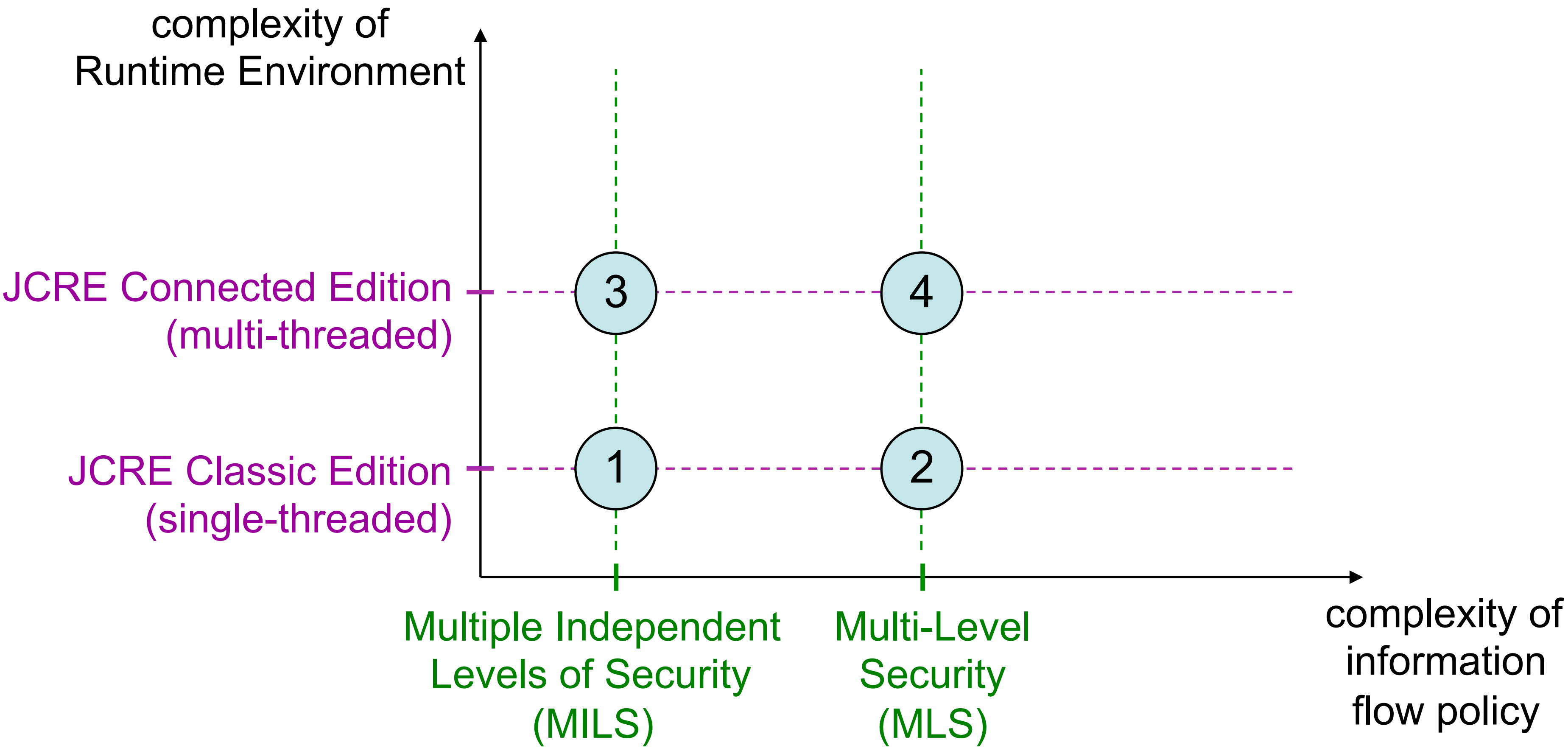
- Run-time checks
  - block instructions that move data across security domains
    - same as single-threaded JCRE
  - closes explicit channels
- Scheduling policy
  - each security domain is allocated a <u>fixed</u> time slot in a fixed cycle
    - time slot is allocated even if security domain not active
  - closes timing channels



Proof that Monitor + Scheduler Guarantee Policy

- Proof that monitor blocks explicit flows is similar to single-threaded case
  - in particular, execution in a domain does not affect and is not affected by other domains, i.e. depends on domain’s sub-state only
- Proof that scheduler blocks timing flows
  - scheduling decision depends on thread’s (= domain’s) sub-state only
  - therefore, two arbitrary traces with the same inputs at the same times to a domain have parallel executions also w.r.t. timing

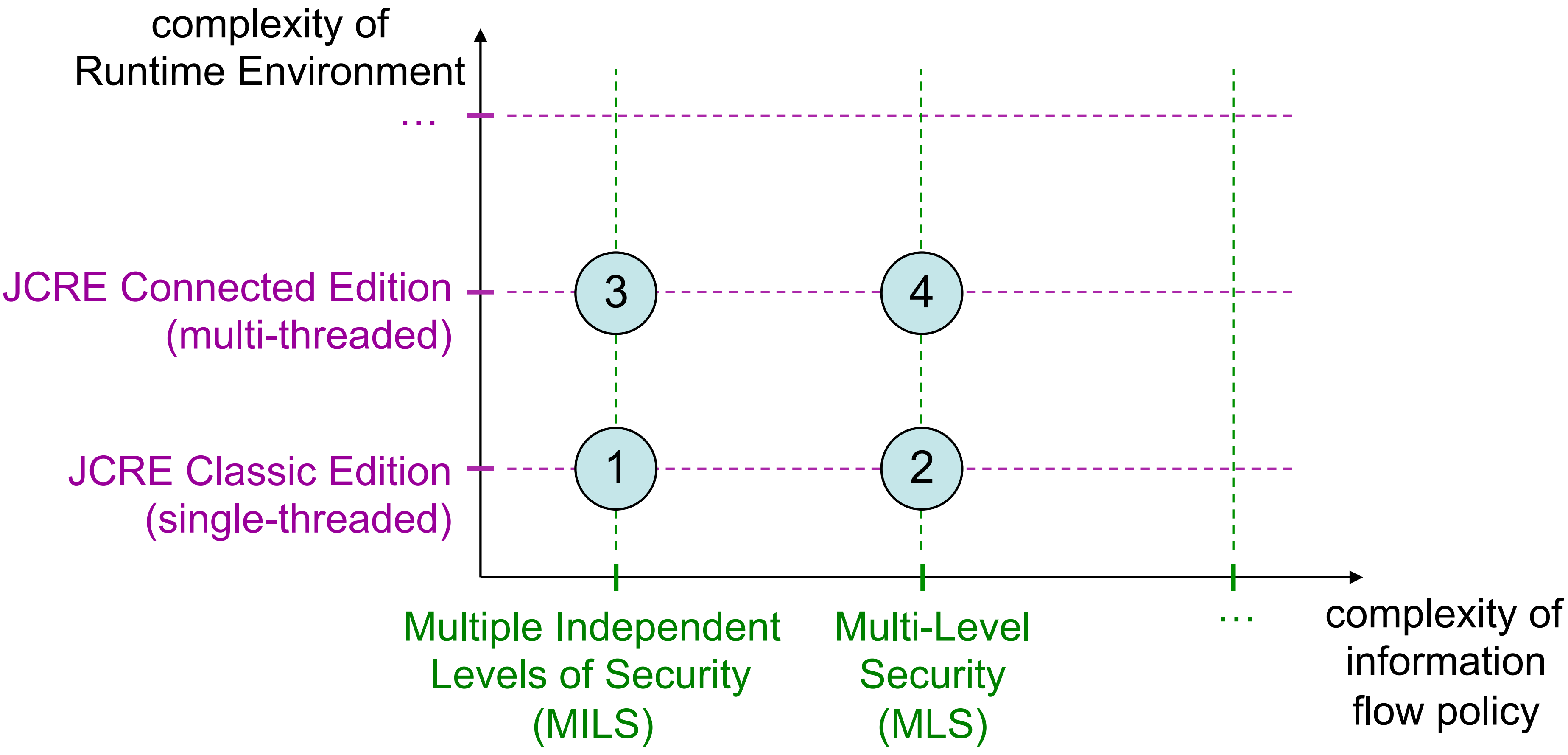
Space of Exploration



MLS for Multi-Threaded JCRE

- Combines features of MLS and multi-threading
- We use the same MLS monitor as in the single-threaded case
- We use the same scheduler as in the MILS case (i.e. fixed time slot in a fixed cycle)
  - adequate, because we just need to close timing channels (no need to allow timing flows from lower to higher, because there are explicit flow mechanisms)
  - but it could be relaxed, for better processor utilization (allow timing flows from lower to higher, if that improves processor utilization)

Space of Exploration



Recap

- Java Card
- Multi-domain token
- Information flow policies in Java Card
- Generative approach
- MILS in single-threaded Java Card
  - policy as non-interference of I/O exchanges
  - run-time monitor
  - proof that monitor guarantees policy
- MLS in single-threaded Java Card
  - more flexible policy
  - more flexible monitor
  - proof that monitor guarantees policy
- MILS in multi-threaded Java Card
  - time and timing channels
  - scheduling policy to prevent timing channels
  - proof that monitor + scheduler guarantee policy
- MLS in multi-threaded Java Card
  - combines features from previous two cases