ProvenCore: Towards a Verified Isolation Micro-Kernel

Stéphane Lescuyer

Prove & Run

MILS Workshop, Amsterdam, 20/01/2015

Summary

PROVE & RUN

ProvenCore's objectives

Objectives

- formally prove security properties of a μ-kernel
  - → absence of runtime-errors
  - → functional specifications
  - → integrity / confidentiality
- using IDE and language developed at Prove & Run
- target high-level Common Criteria evaluation
- generate documentation from the specs
- generate executable C code from the specs

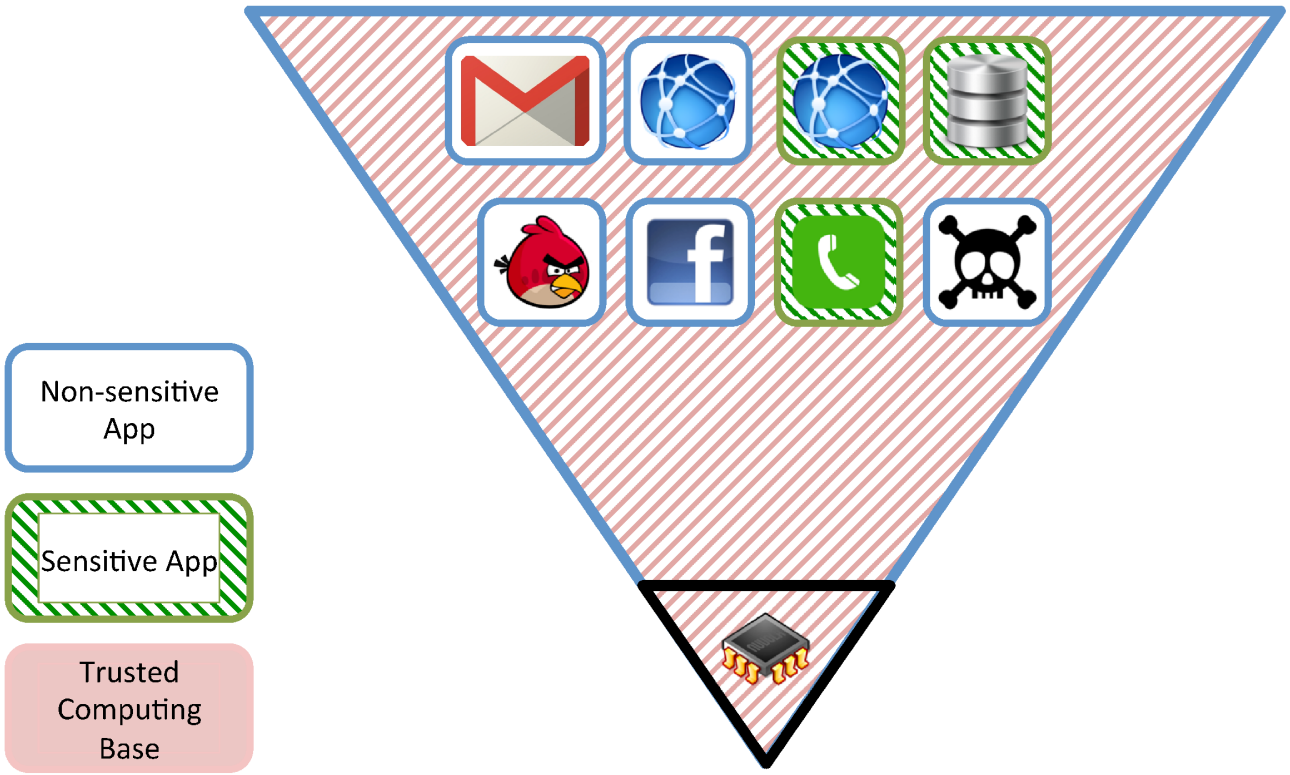
ProvenCore's objectives

Objectives

- formally prove security properties of a μ-kernel
  - → absence of runtime-errors
  - → functional specifications
  - → integrity / confidentiality
- using IDE and language developed at Prove & Run
- target high-level Common Criteria evaluation
- generate documentation from the specs
- generate executable C code from the specs

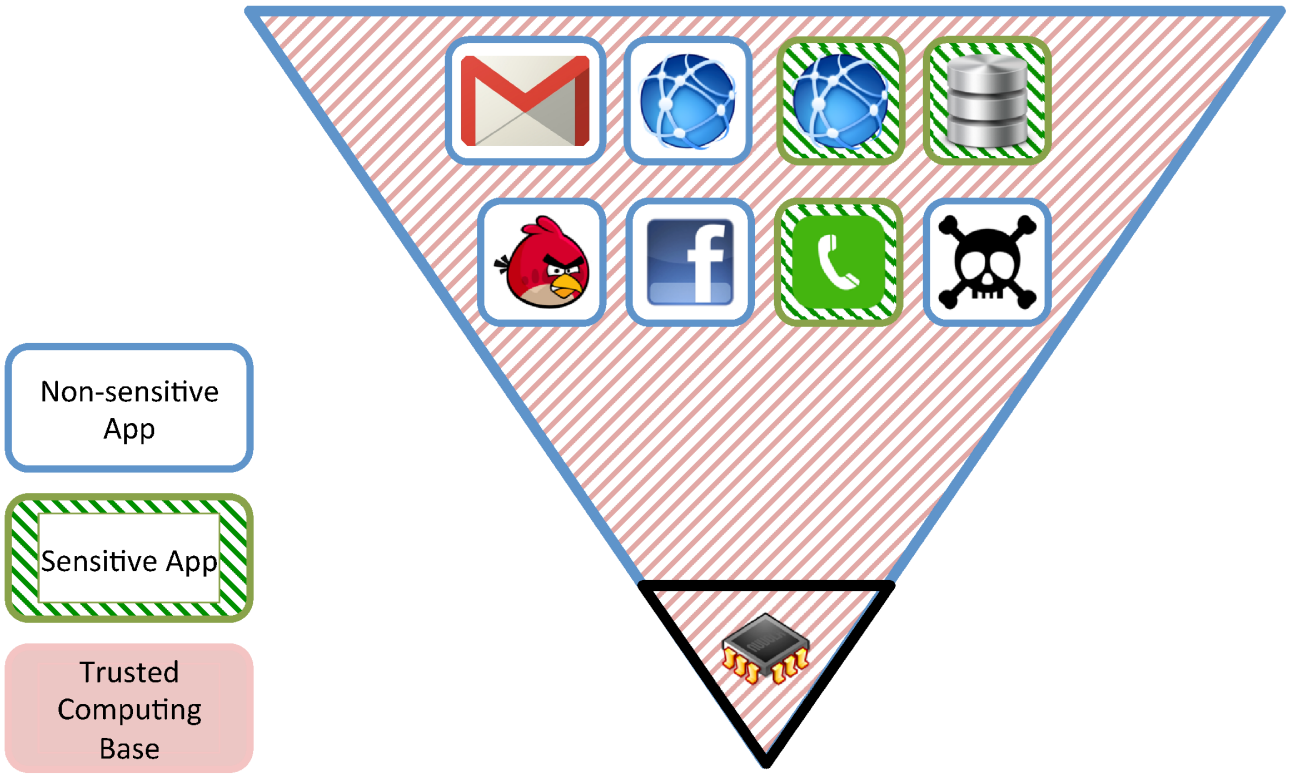
→ take advantage of the proof effort

Typical mobile device architecture



PROVE & RUN

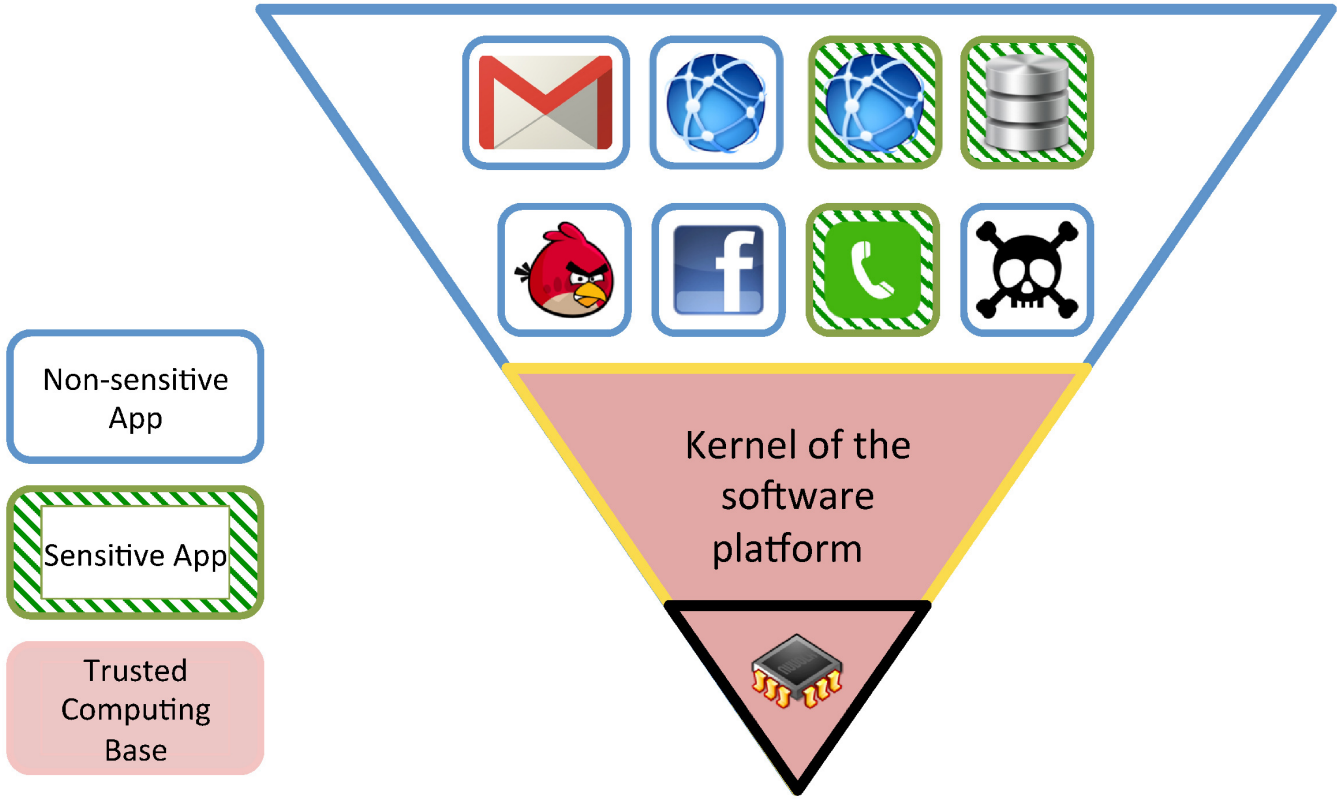
Typical mobile device architecture



- no assets are safe
- *Corporate Owned, Personally Enabled*

PROVE & RUN

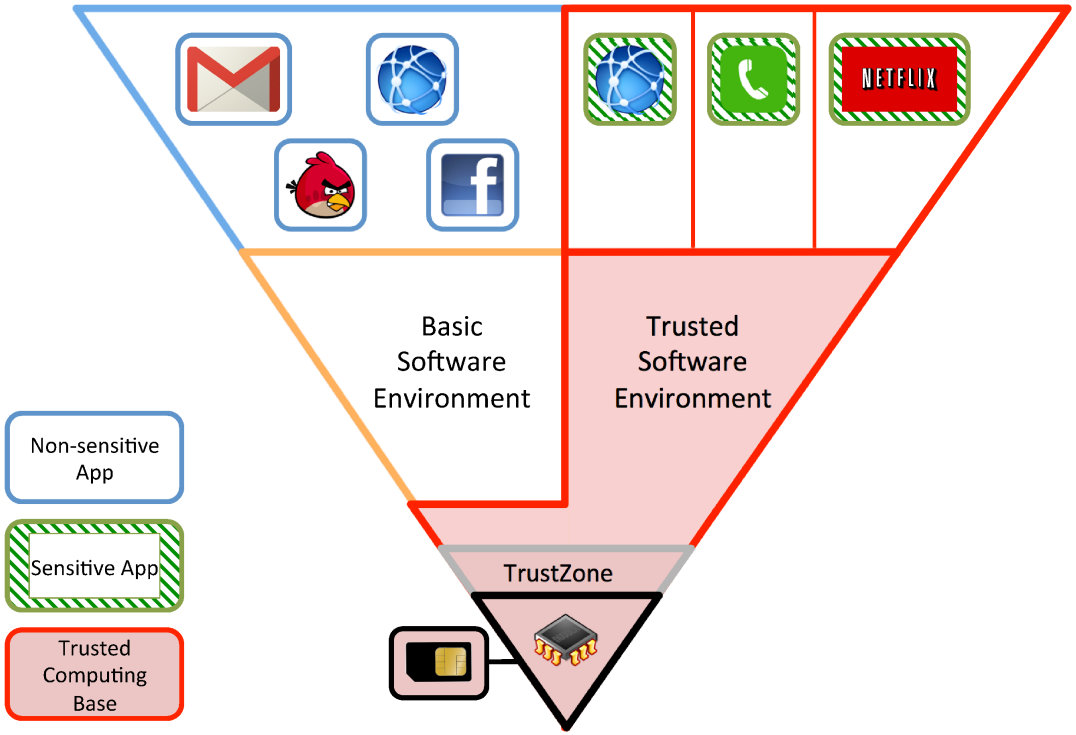
Securing the rich kernel?



- theoretically possible...
- ...but in practice too complex, moving target

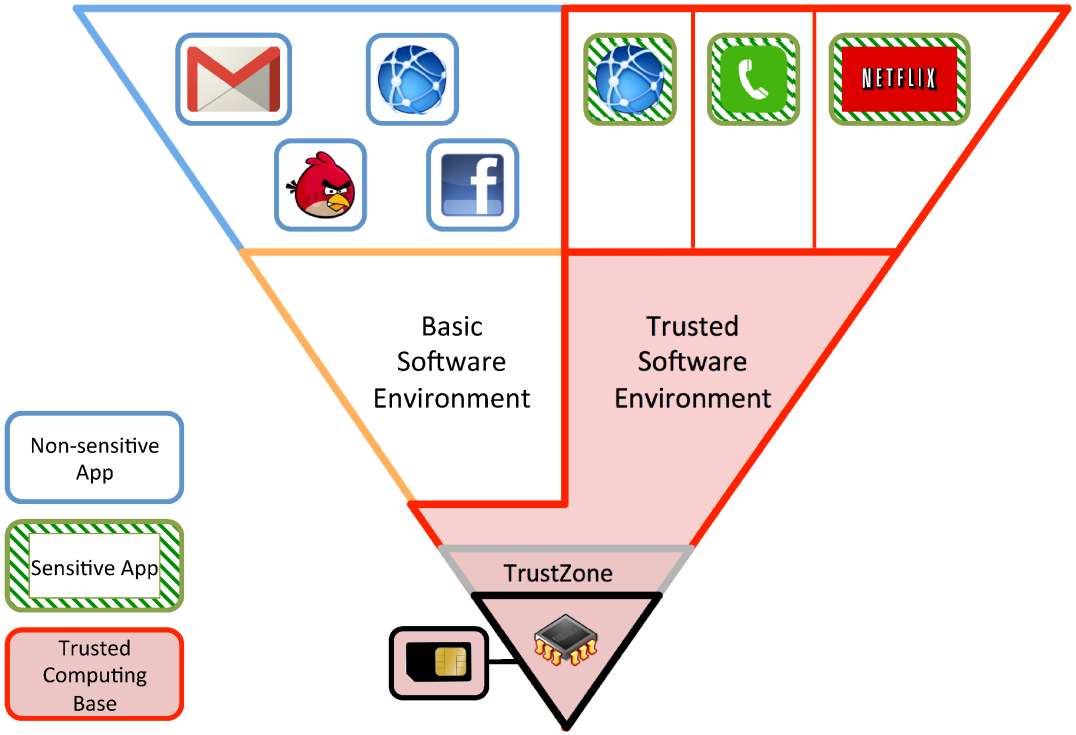
PROVE & RUN

TrustZone to the rescue



PROVE & RUN

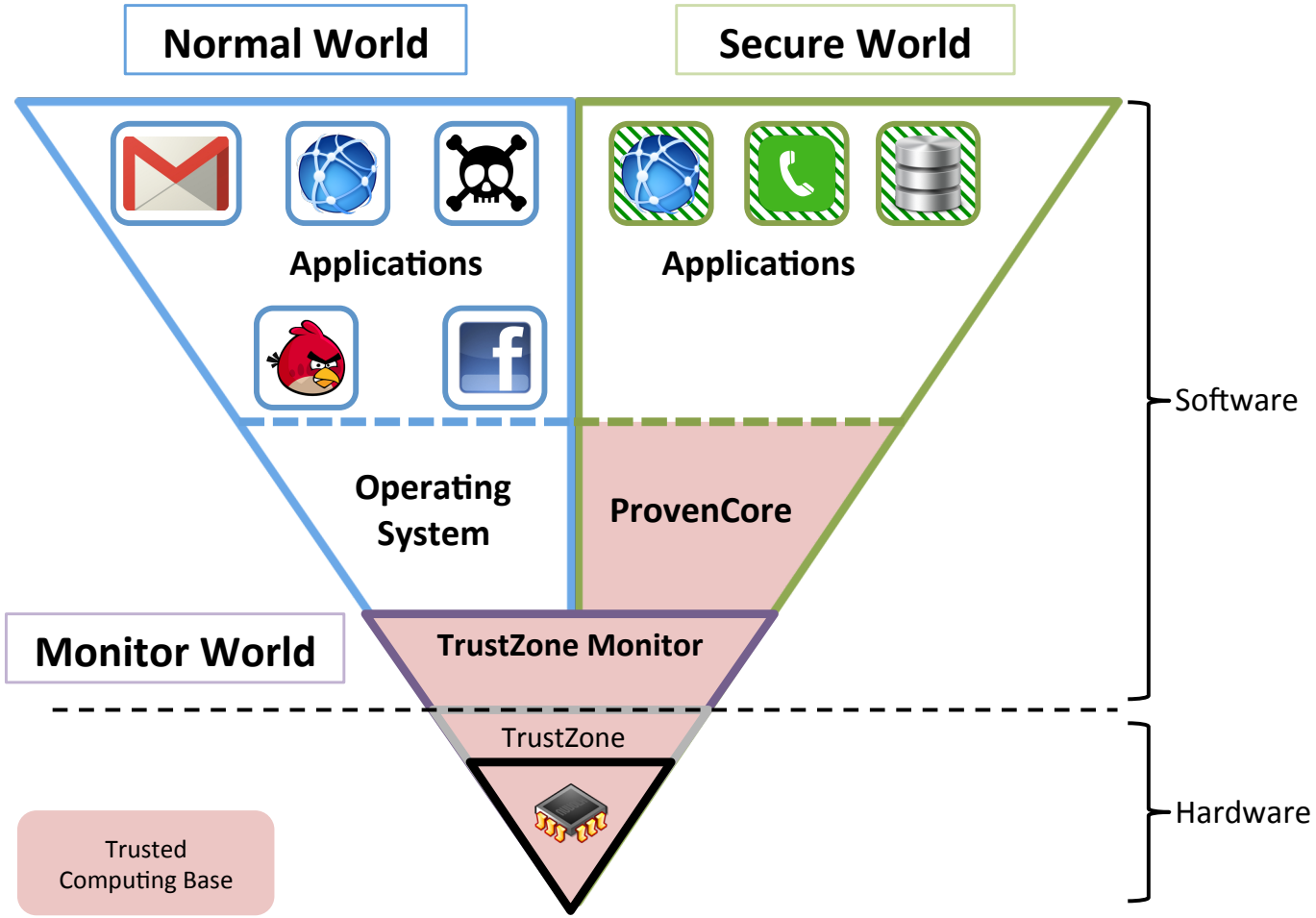
TrustZone to the rescue



→ the OS on the secure side can be constrained!

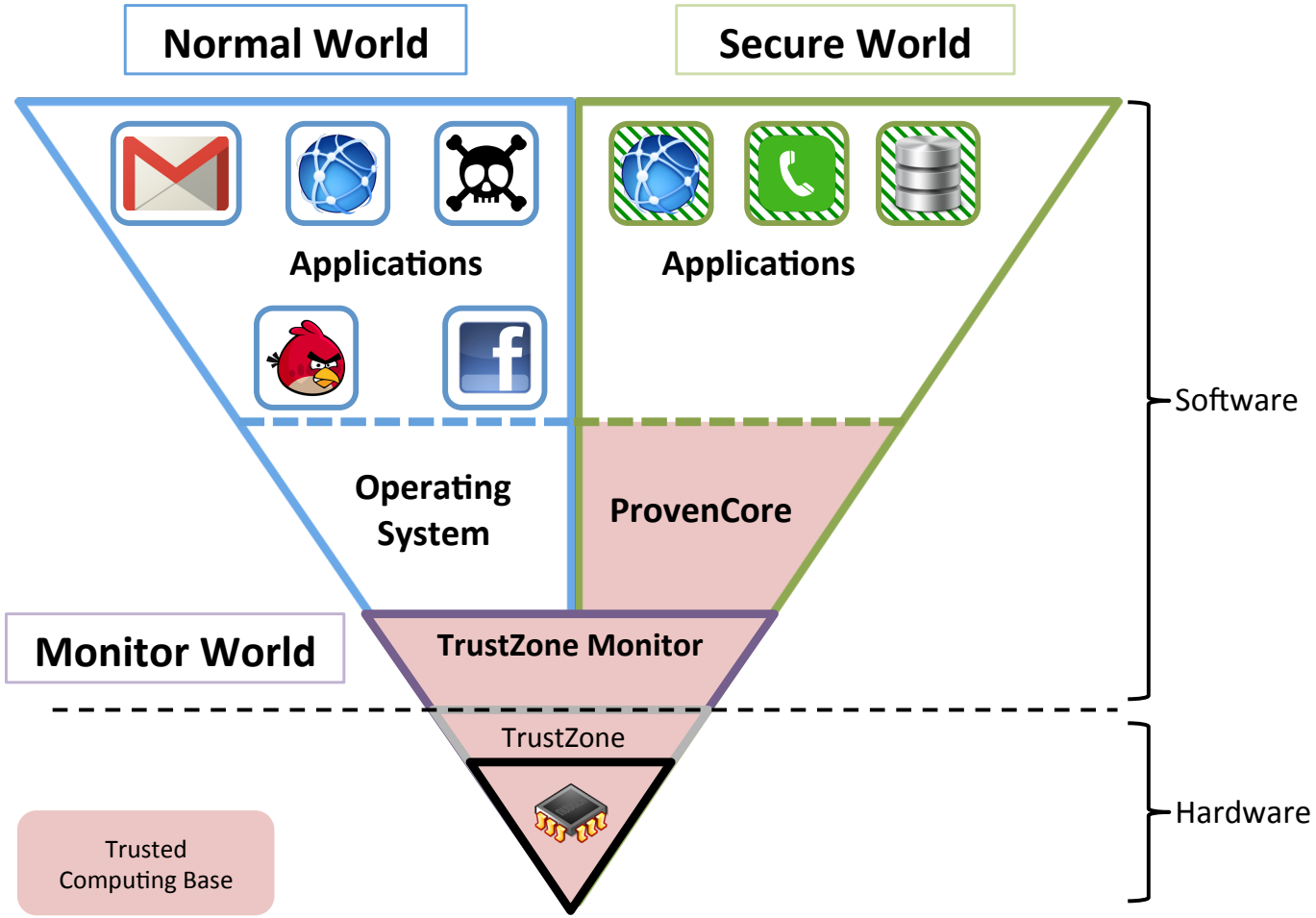
PROVE & RUN

ProvenCore as a secure world OS



PROVE & RUN

ProvenCore as a secure world OS



→ possible safe BYOD policy

→ possible to run a TEE on ProvenCore

PROVE & RUN

Minix heritage

ProvenCore is largely inspired by Minix 3.1

PROVE & RUN

Minix heritage

ProvenCore is largely inspired by Minix 3.1

Why Minix?

- μ-kernel
- well-documented
- mostly POSIX-compliant
- simple yet versatile enough
  - no real-time constraints

PROVE & RUN

Minix heritage

ProvenCore is largely inspired by Minix 3.1

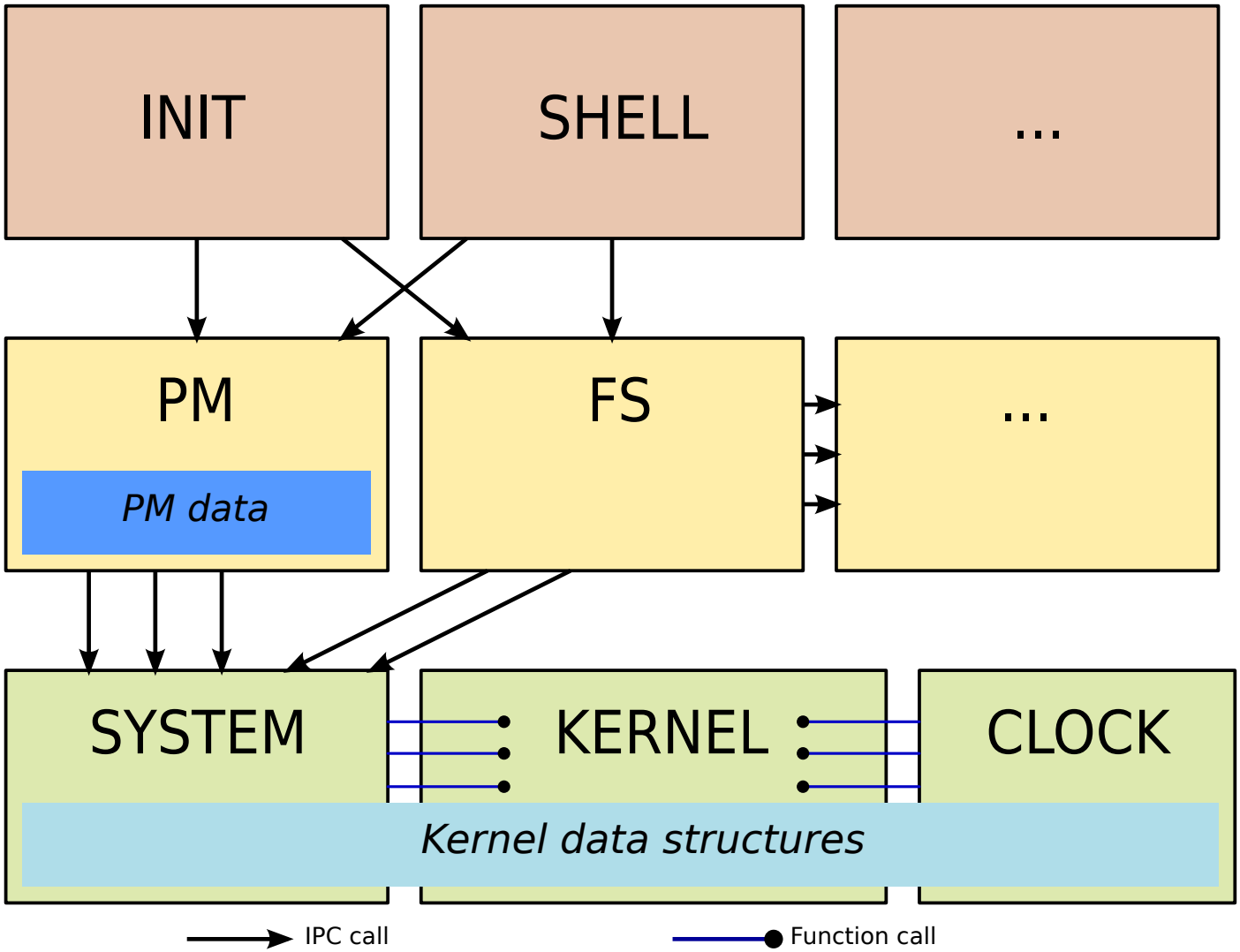
Why Minix?

- μ-kernel
- well-documented
- mostly POSIX-compliant
- simple yet versatile enough
  - no real-time constraints

Still...

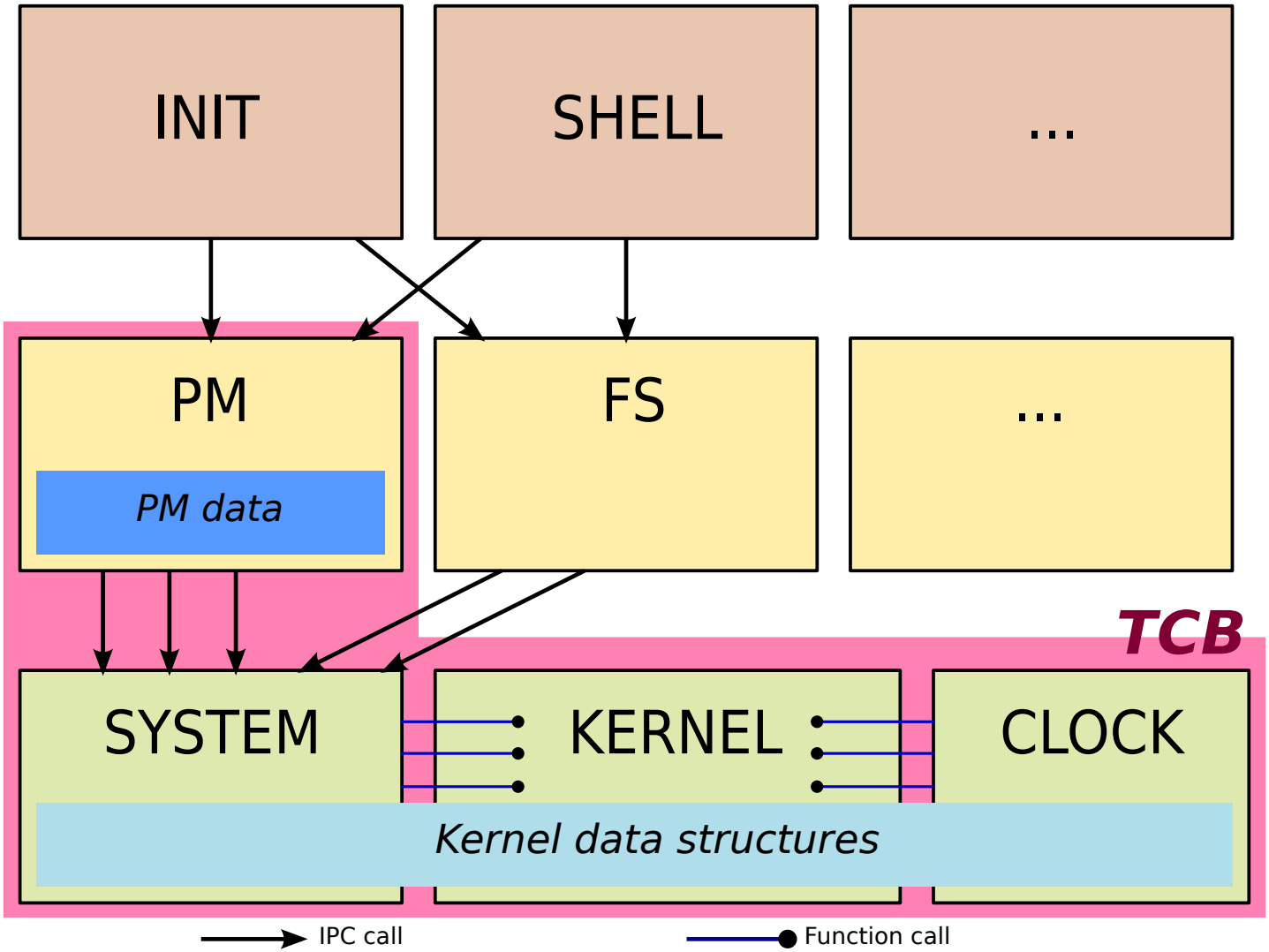
- we had to port it to ARM and MMU
- not a well-defined TCB

The TCB incident



PROVE & RUN

The TCB incident



PROVE & RUN

Towards sequentiality

Non-sequential TCB

- 4 different processes, 2 different address spaces
- (blocking) IPC communications inside the TCB
- asynchronous hardware interrupts in the TCB
- behaviour depends on the non-demotion of PM in the scheduler

Towards sequentiality

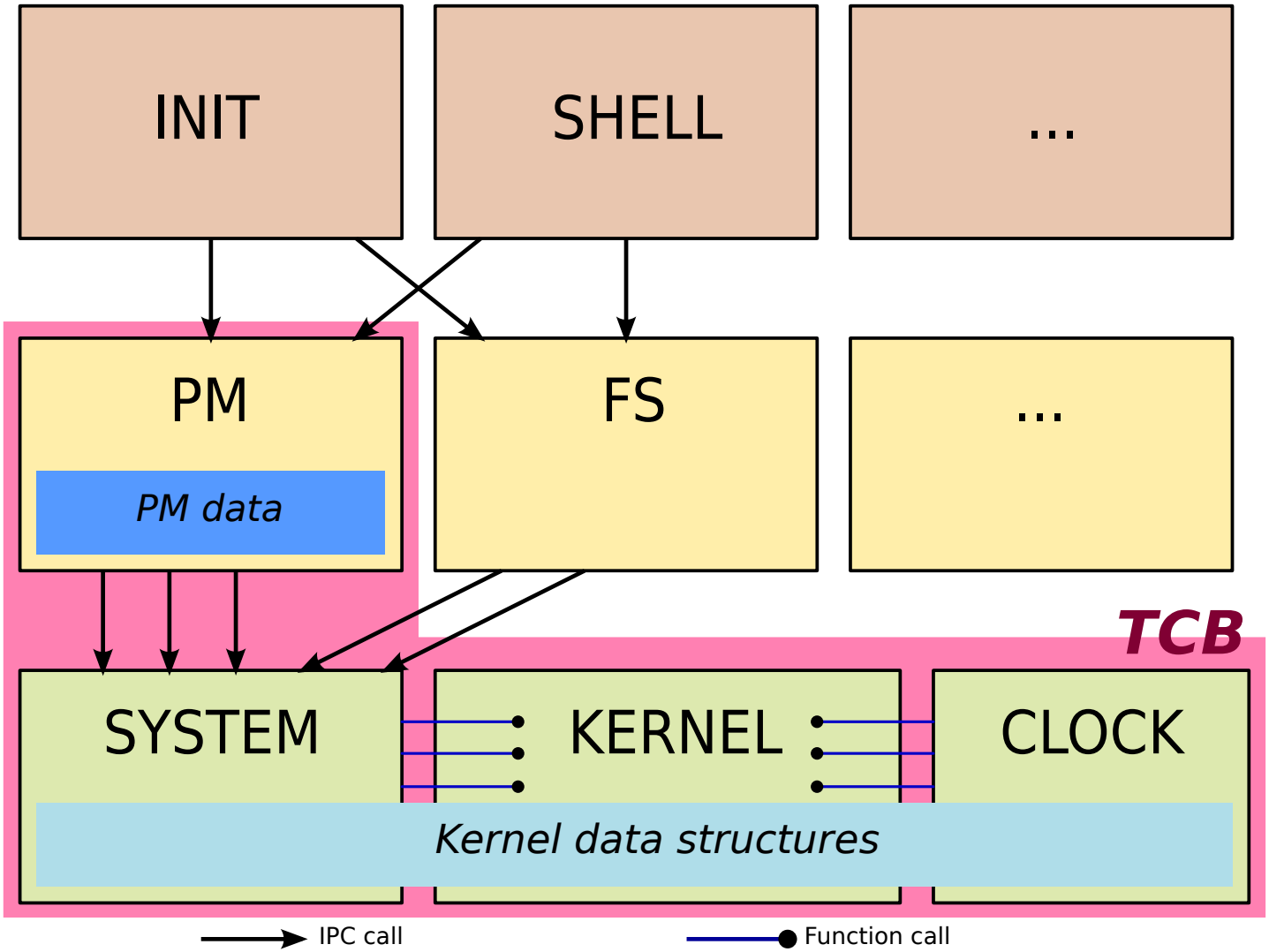
Non-sequential TCB

- 4 different processes, 2 different address spaces
- (blocking) IPC communications inside the TCB
- asynchronous hardware interrupts in the TCB
- behaviour depends on the non-demotion of PM in the scheduler

→ formal reasoning on a concurrent TCB possible but extremely *hard*

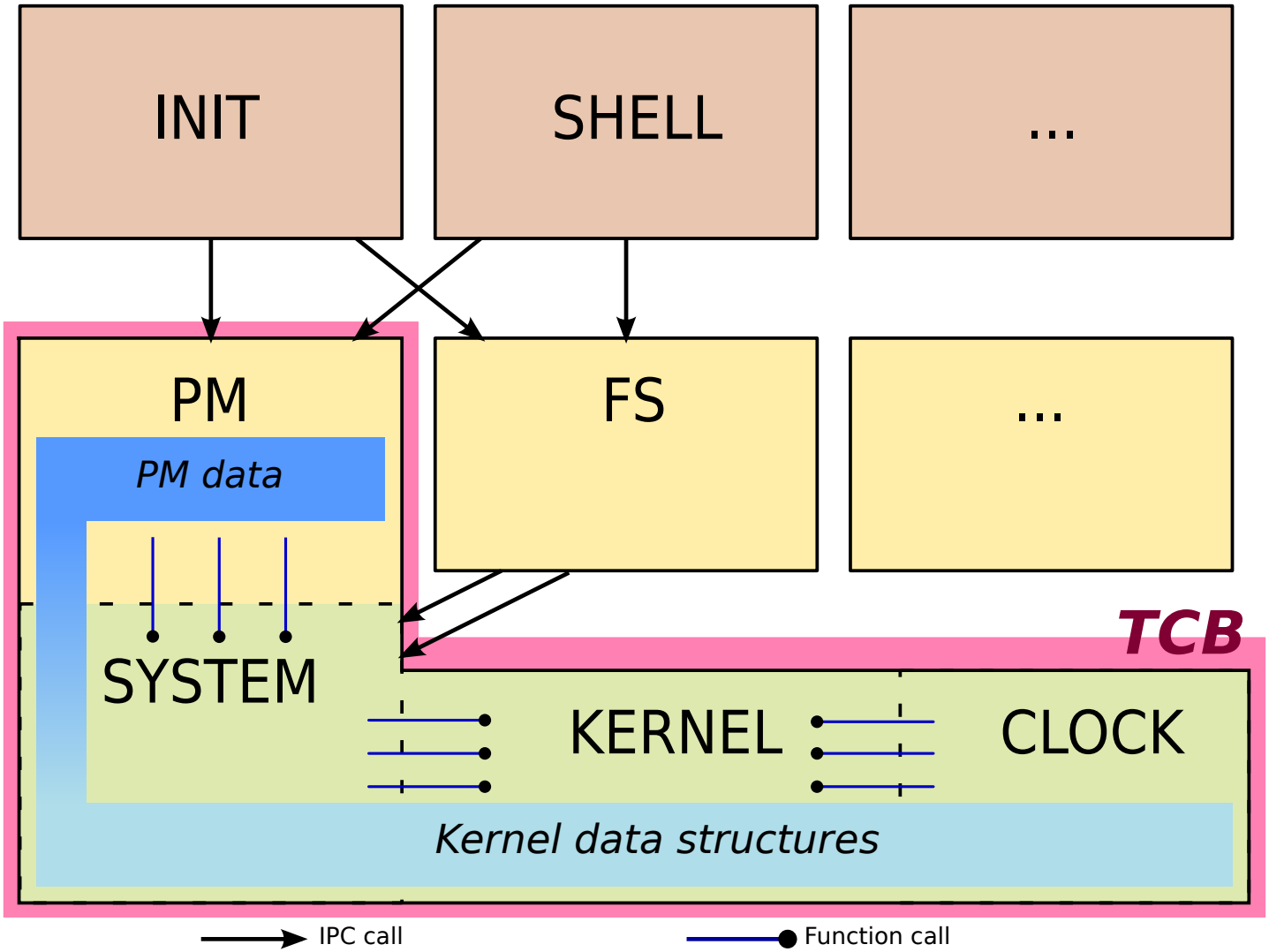
→ semi-formal reasoning possible, but unsatisfactory

Sequential TCB



PROVE & RUN

Sequential TCB



PROVE & RUN

ProvenCore's features

Process Management

- FORK
- EXIT
- EXEC of authorized codes

ProvenCore's features

Process Management

- FORK
- EXIT
- EXEC of authorized codes

IPC

- synchronous message-passing IPCs with timeouts
- asynchronous notifications

PROVE & RUN

ProvenCore's features

Process Management

- FORK
- EXIT
- EXEC of authorized codes

IPC

- synchronous message-passing IPCs with timeouts
- asynchronous notifications

Data transfers

- process-to-process data copies, guarded by authorizations
- safe shared memory system

PROVE & RUN

Security policies

- Resource allocation policy
- Access control policy
- Information flow policy

PROVE & RUN

Security policies

Resource allocation policy

time configurable scheduling priority bounds

RAM physical quotas can be put on binaries

Access control policy

Information flow policy

PROVE & RUN

Security policies

Resource allocation policy

Access control policy

- k-calls configurable bitmaps
- copy precise transferable R/W/RW authorities
- shm exactly one RW access at a time

Information flow policy

Security policies

Resource allocation policy

Access control policy

Information flow policy

- IPCs allowed calls configurable
- IPCs allowed endpoints configurable

High-level properties

Integrity

Resources (registers, data, code) of a process P can only be modified by another process Q provided P explicitly allowed it, and by the kernel following a request of P.

Confidentiality

Resources (registers, data, code) of a process P can only be read by another process Q provided P explicitly allowed it, and by the kernel following a request of P.

Prove & Run's Smart language

Smart

- first-order polymorphic functional language
- used both for models and specifications
- built-in algeraic datatypes
- pure: only manipulate values

Prove & Run's Smart language

Smart

- first-order polymorphic functional language
- used both for models and specifications
- built-in algeraic datatypes
- pure: only manipulate values

Strong separation of data-flow & control-flow

- each *predicate* returns a number of outputs, as well as labels
- labels can typically be used for exceptional cases, or booleans
- only outputs associated to the returned label are created
- ignored labels lead to proof obligations

PROVE & RUN

Smart prototype examples

Labels as exceptions

```
public get(array<A> a, int idx, A v+) -> [true, oob]
program { ... }
```

Smart prototype examples

Labels as exceptions

```
public get(array<A> a, int idx, A v+) -> [true, oob]
program { ... }
```

Pure control-flow predicates

```
public is_slot_free(proc p) -> [true, false]
program { ... }
```

PROVE & RUN

Smart prototype examples

Labels as exceptions

```
public get(array<A> a, int idx, A v+) -> [true, oob]
program { ... }
```

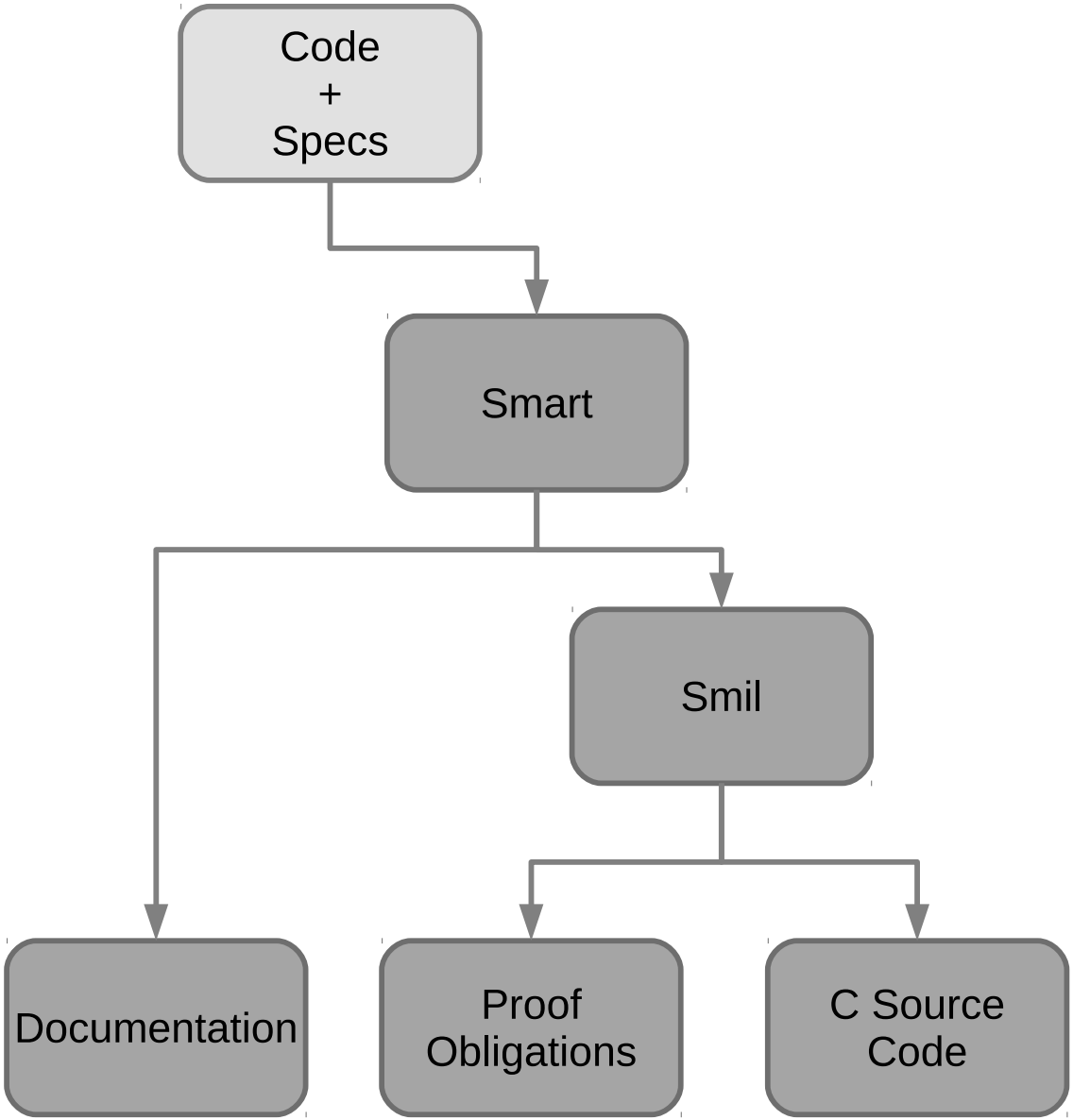
Pure control-flow predicates

```
public is_slot_free(proc p) -> [true, false]
program { ... }
```

No side-effect hidden

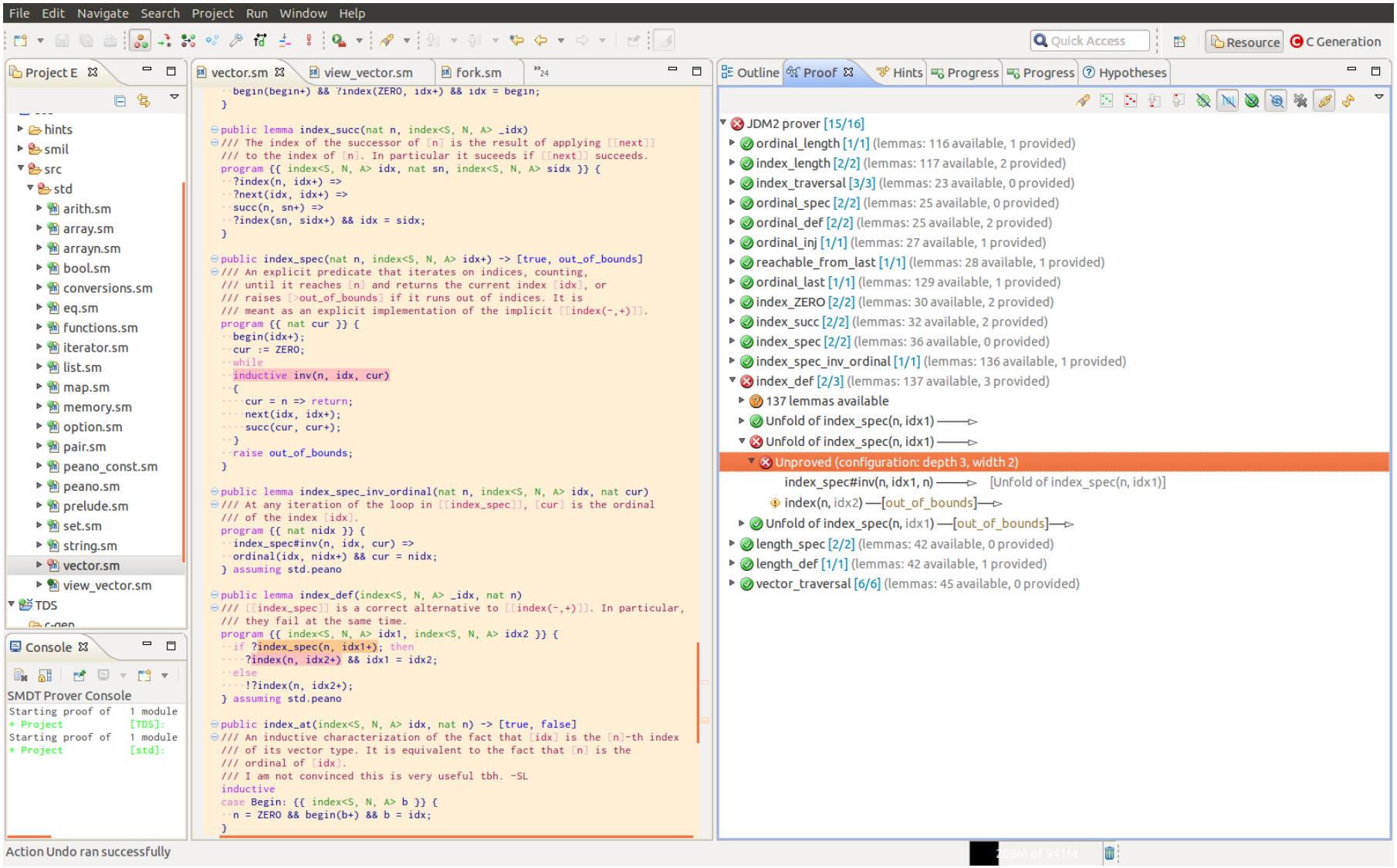
```
public fetch_irq(gic gic0, nat irq+, gic gic+)
implicit program
```

Tool-chain overview

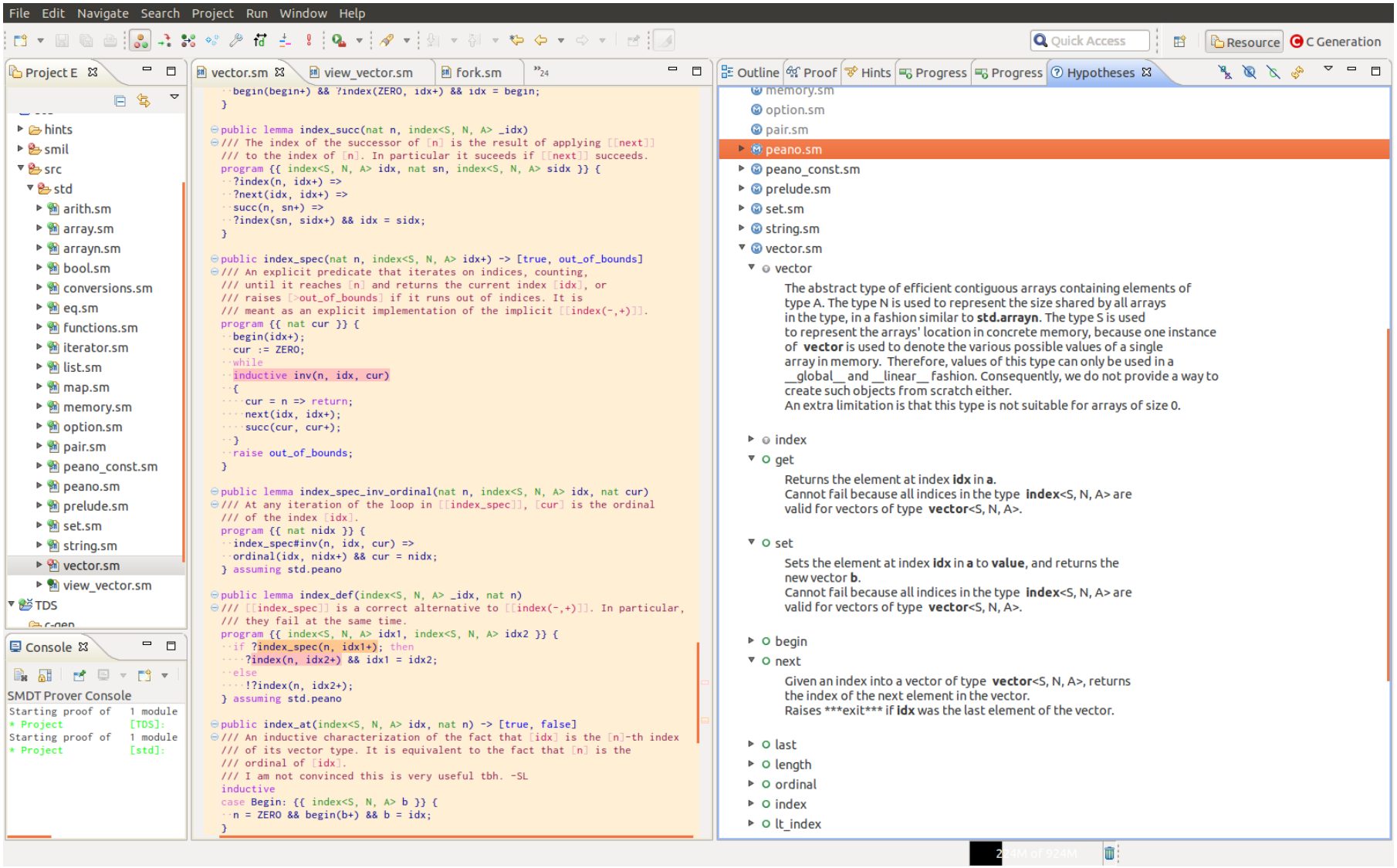


PROVE & RUN

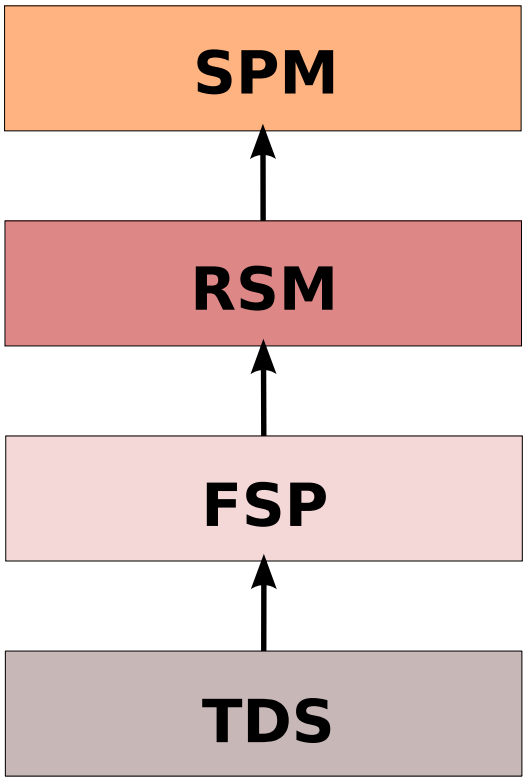
Screenshots



Screenshots

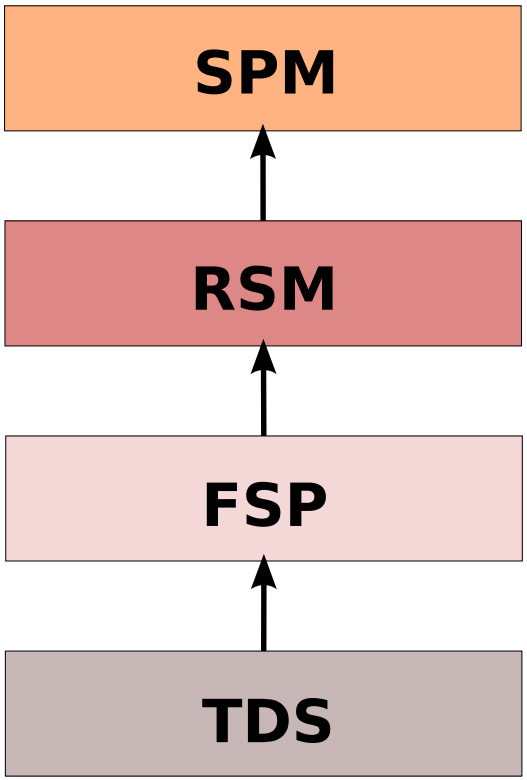


Refinements



PROVE & RUN

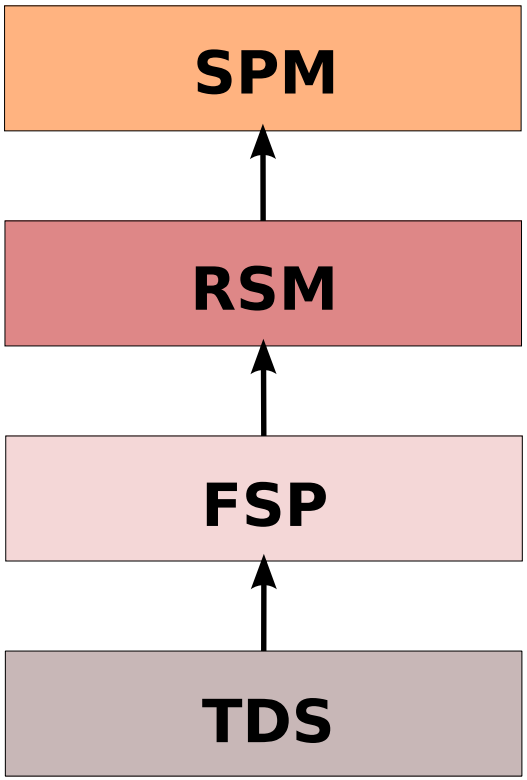
Refinements



TDS Target Design

PROVE & RUN

Refinements

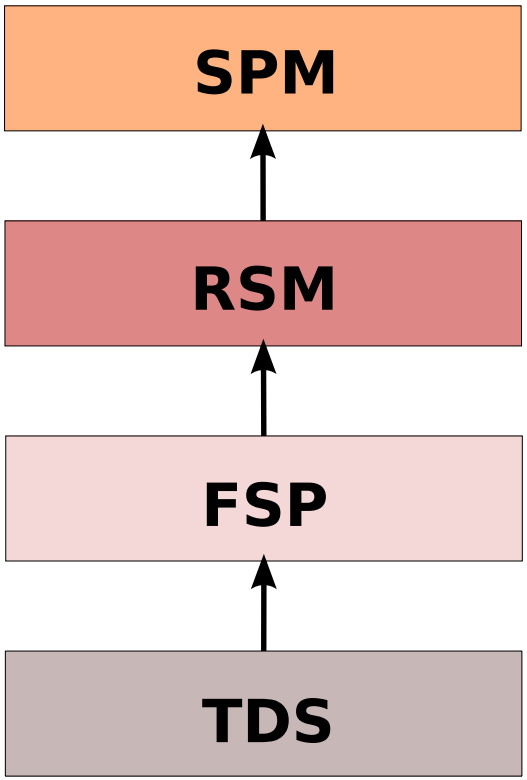


FSP Functional Specifications

TDS Target Design

PROVE & RUN

Refinements



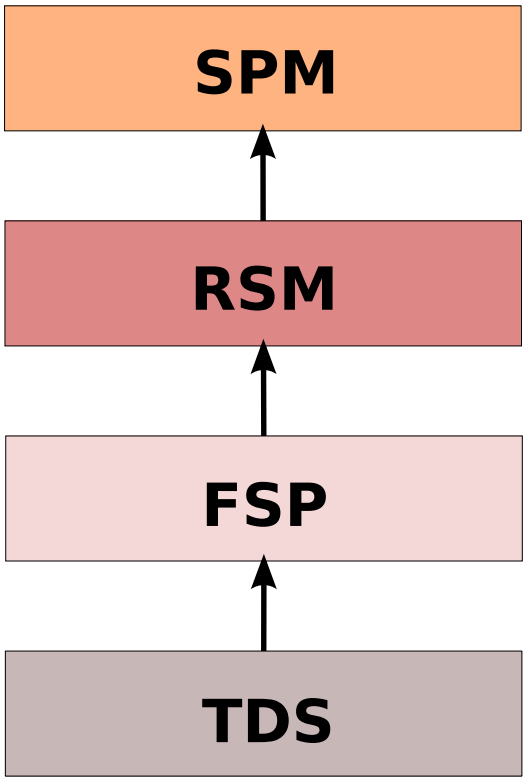
RSM Refined Security Model

FSP Functional Specifications

TDS Target Design

PROVE & RUN

Refinements



SPM Security Policy Model

RSM Refined Security Model

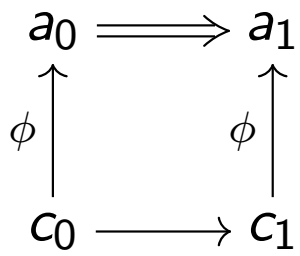
FSP Functional Specifications

TDS Target Design

PROVE & RUN

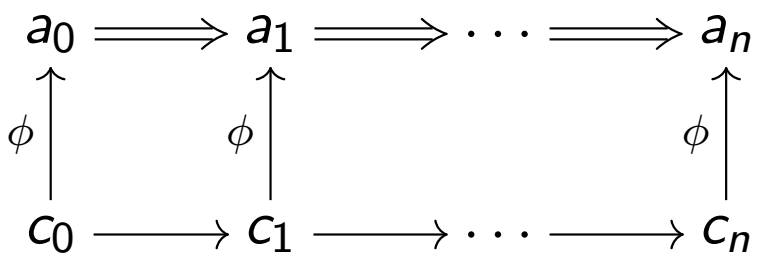
Forward Simulation

ϕ a view from concrete states to abstract states



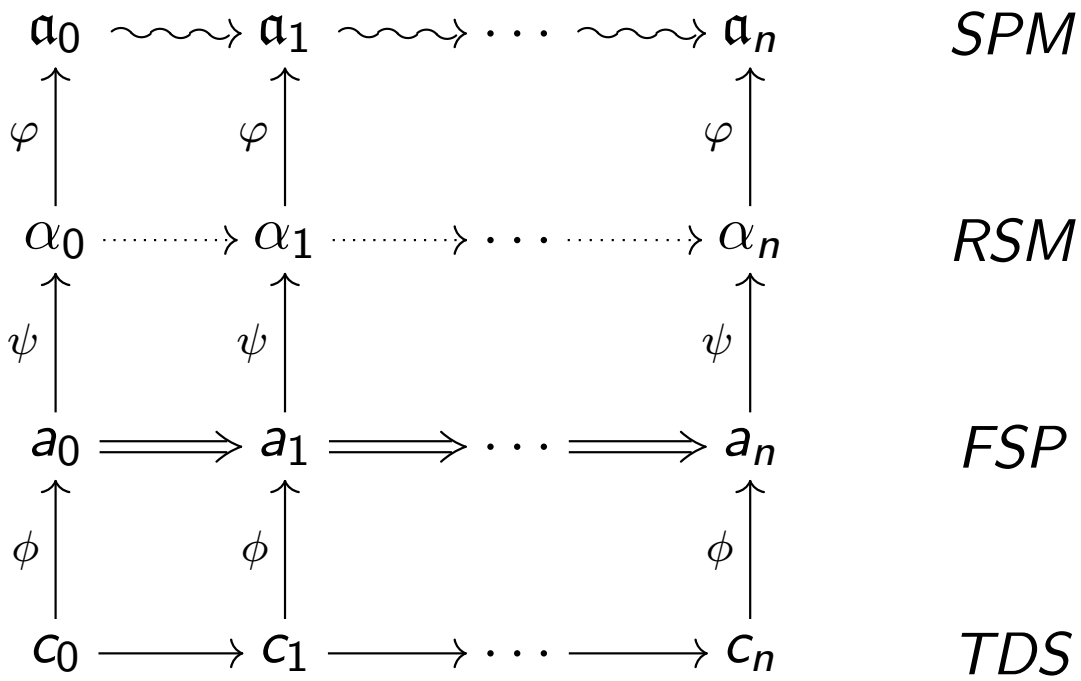
Forward Simulation

ϕ a view from concrete states to abstract states



Forward Simulation

ϕ a view from concrete states to abstract states



PROVE & RUN

FSP characteristics

FSP characteristics

- functional code simulating the TDS
- simplified data structures
- simplified algorithms
- linearized address spaces

FSP characteristics

FSP characteristics

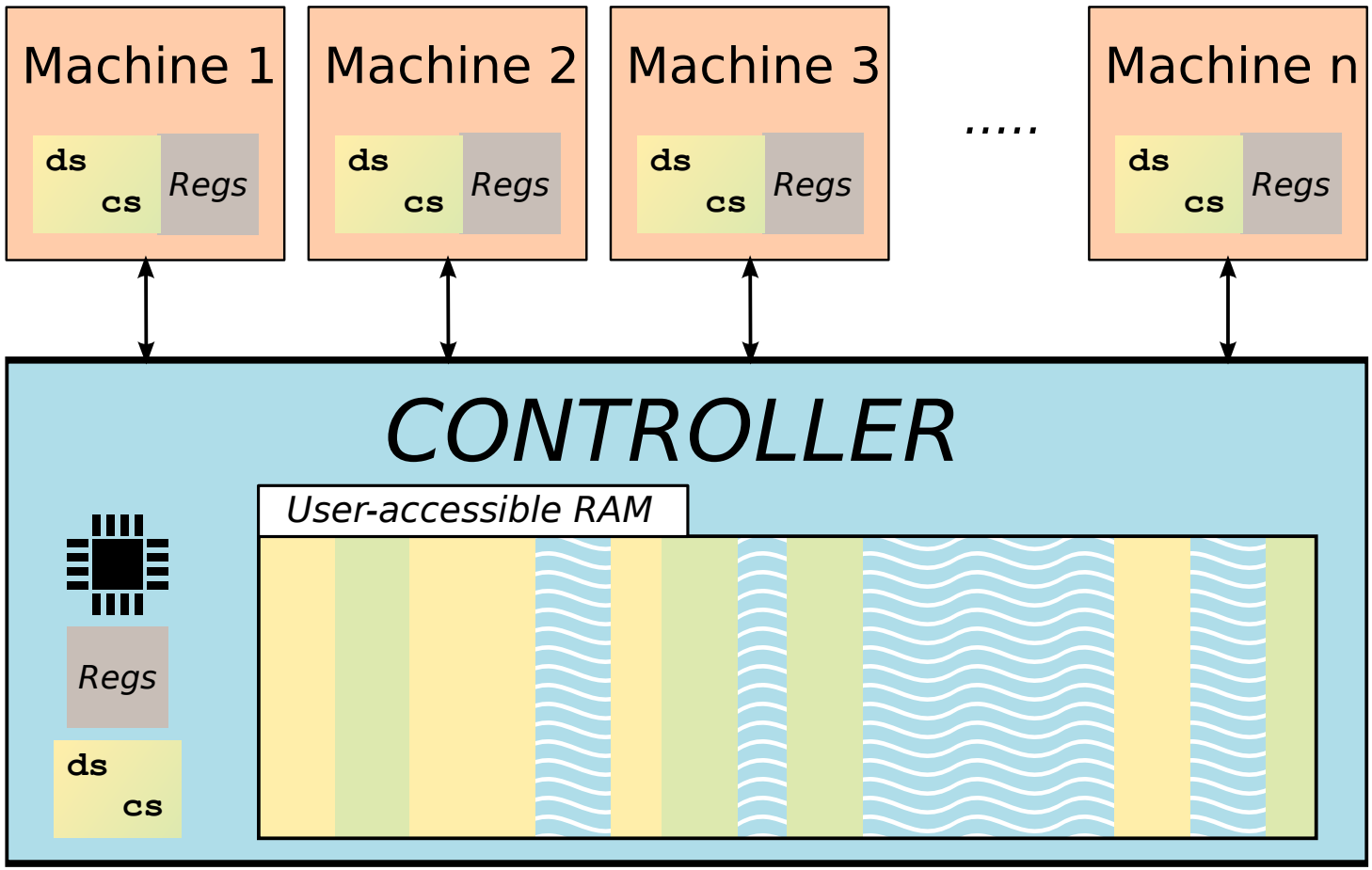
- functional code simulating the TDS
- simplified data structures
- simplified algorithms
- linearized address spaces

→ structural invariants captured

→ functional invariants easier to describe

→ no more bad pointers, overflows, etc

Schematic view of RSM



PROVE & RUN

SPM characteristics

Security model characteristics

- non-deterministic transition system
  - → uses *predictions* to simulate external ND
- all processes have their own resources
- generalization of non-critical heuristics

SPM characteristics

Security model characteristics

- non-deterministic transition system
  - → uses *predictions* to simulate external ND
- all processes have their own resources
- generalization of non-critical heuristics

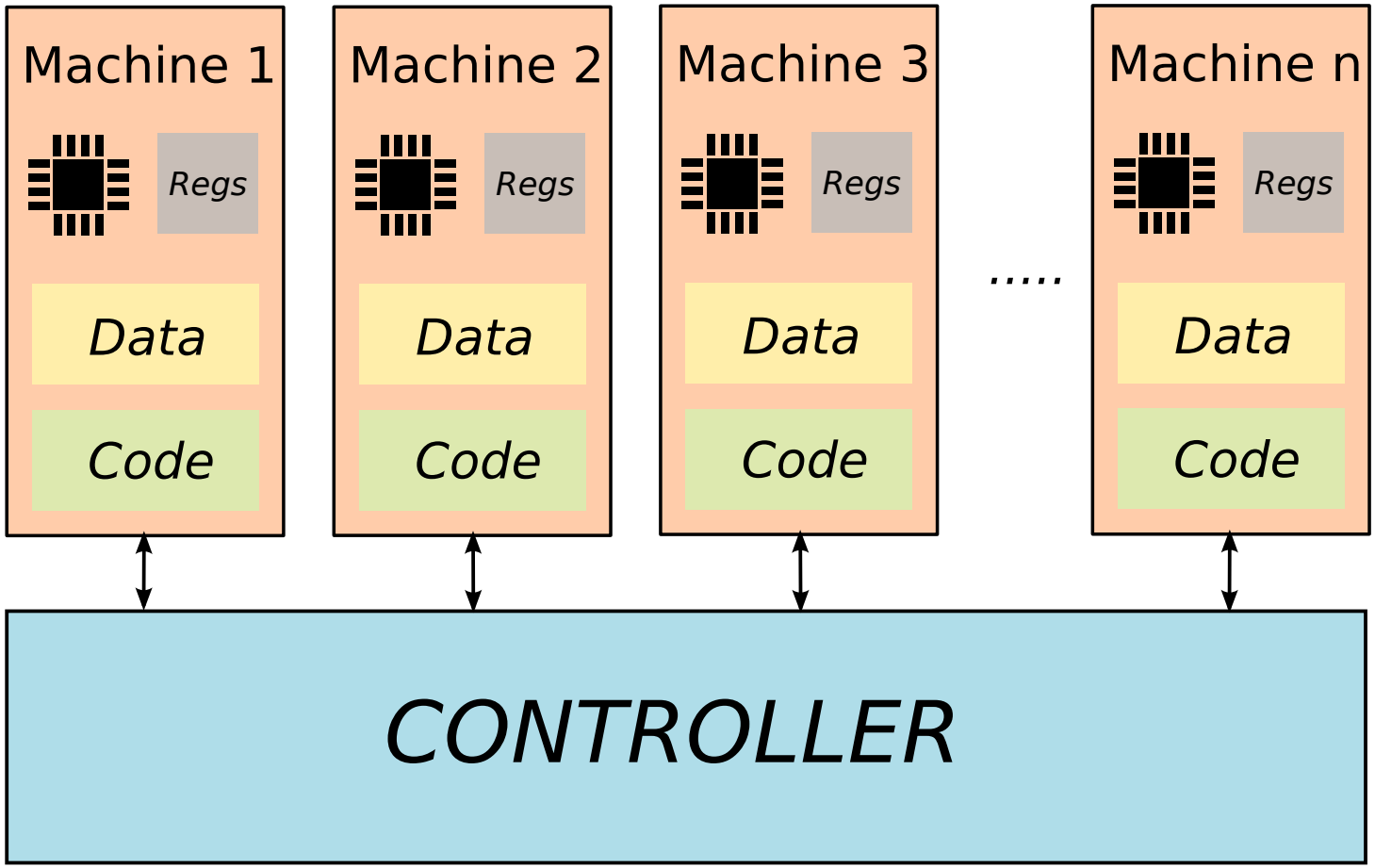
→ functional invariants captured

→ processes are isolated by construction

→ proof and expression of security properties

PROVE & RUN

Schematic view of the SPM



PROVE & RUN

Conclusion

Prove & Run methodology

- one unique language for all levels of abstraction
- formal refinements between all levels, in Smart
- automatic generation of C code from TDS
- integrated into the Eclipse IDE

ProvenCore

- μ-kernel for secure world on TrustZone aware devices
- formally established integrity and confidentiality
- can host a TEE as a user service
- higher-level models can be reused for other isolation kernels

PROVE & RUN

Appendices

In-place updates, example 1

```
@Ghost
type ram = ...

@Global
@Linear
struct glo {
   ...
   ram ram;
   ...
}
```

PROVE & RUN

In-place updates, example 1

```
@Ghost
type ram = ...

@Global
@Linear
struct glo {
   ...
   ram ram;
   ...
}

private enqueue(global now, int rp, global after+)
```

In-place updates, example 1

```
@Ghost
type ram = ...

@Global
@Linear
struct glo {
   ...
   ram ram;
   ...
}

private enqueue(global now, int rp, global after+)
```

→ `void sm_pred_enqueue(int sm_loc_rp);`

In-place updates, example 2

```
@MustAlias(``a=b'')
public set(array<int> a, int idx, int v,
           array<int> b+) -> [true, out_of_bounds]

... {
  ...
  set(a, i, v, b+);
  set(b, j, w, c+);
  ...
  get(a, j, x+);   // Not OK
}
```

In-place updates, example 2

```
@MustAlias(``a=b'')
public set(array<int> a, int idx, int v,
           array<int> b+) -> [true, out_of_bounds]

... {
  ...
  set(a, i, v, b+);
  set(b, j, w, c+);
  ...
  get(a, j, x+);   // Not OK
}
```

Non-linearity

→ *a* is read after modification of *a* in *b*

→ impossible to update *a* in-place and alias *a* and *b*

→ the analysis produces an error or a copy

In-place updates, example 2

```
... {
   ...
   set(a, i, v, b+);
   set(b, j, w, c+);
   ...
   get(c, j, x+);   // OK
}
```

In-place updates, example 2

```
... {
   ...
   set(a, i, v, b+);
   set(b, j, w, c+);
   ...
   get(c, j, x+);   // OK
}
```

```
set_result_t set(int *a, int idx, int v);

... {
    set(&a, i, v);
    set(&a, j, w);
    ...
    get(&a, j, x+);
}
```

Abstraction example TDS → FSP (1/2)

TDS

```
struct state {
  array<cell> data;
  option<int> head;
  t current;
}

struct cell {
  t value;
  option<int> next;
}
```

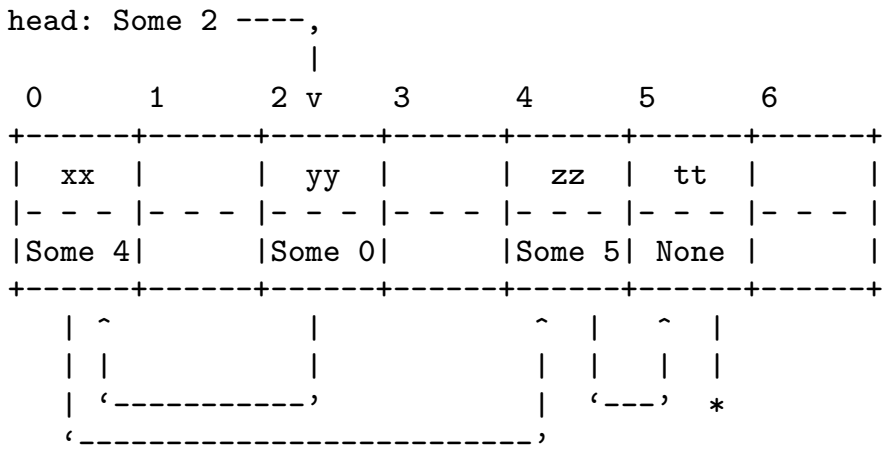
PROVE & RUN

Abstraction example TDS → FSP (1/2)

TDS

```
struct state {
  array<cell> data;
  option<int> head;
  t current;
}

struct cell {
  t value;
  option<int> next;
}
```



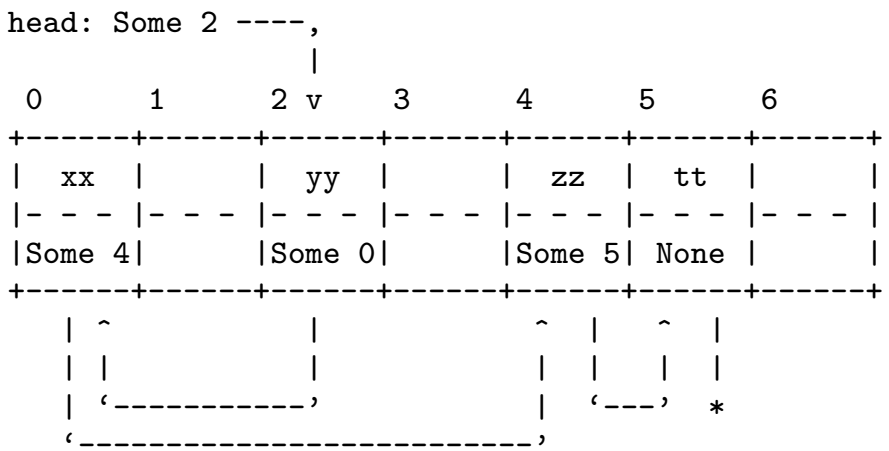
PROVE & RUN

Abstraction example TDS → FSP (1/2)

TDS

```
struct state {
  array<cell> data;
  option<int> head;
  t current;
}

struct cell {
  t value;
  option<int> next;
}
```



Invariants

- all indices valid
- no cycles

PROVE & RUN

Abstraction example TDS → FSP (2/2)

FSP

```
struct state {
  list<t> cells;
  t current;
}

type list<A> =
| Nil
| Cons(A a, list<A> l);
```

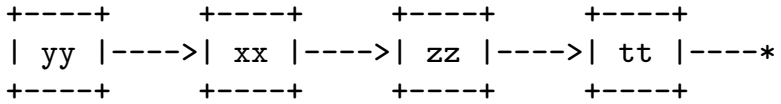
PROVE & RUN

Abstraction example TDS → FSP (2/2)

FSP

```
struct state {
  list<t> cells;
  t current;
}

type list<A> =
| Nil
| Cons(A a, list<A> l);
```



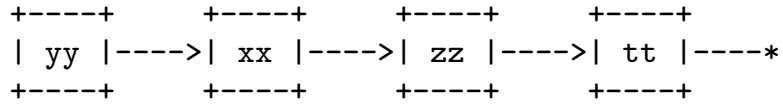
PROVE & RUN

Abstraction example TDS → FSP (2/2)

FSP

```
struct state {
  list<t> cells;
  t current;
}

type list<A> =
| Nil
| Cons(A a, list<A> l);
```



Properties

- the list is sorted
- insertion/deletion/... preserve the sortedness
- sorted in FSP ⇒ sorted in TDS

PROVE & RUN

Expression of high-level properties

Integrity

Let s be an SPM state, $\bar{\pi}$ some predictions, and t such that $s, \bar{\pi} \rightarrow^* t$, et i the index of a machine in s that never runs during these transitions. Then,

(i) s_i and t_i have identical codes

(ii) s_i and t_i have identical registers unless i is unblocked between s and t

(iii) the value at address p in the memory of s_i is the same as in t_i, unless p was writable in s_i, which happens when:

- s_i is waiting to receive a message and p belongs to the range where the message should be received
- s_i has a write memory permission in its memory that covers p

Expression of high-level properties

Integrity

Let s be an SPM state, $\bar{\pi}$ some predictions, and t such that $s, \bar{\pi} \rightarrow^* t$, et i the index of a machine in s that never runs during these transitions. Then,

(i) s_i and t_i have identical codes

(ii) s_i and t_i have identical registers unless i is unblocked between s and t

(iii) the value at address p in the memory of s_i is the same as in t_i, unless p was writable in s_i, which happens when:

- s_i is waiting to receive a message and p belongs to the range where the message should be received
- s_i has a write memory permission in its memory that covers p

Corollary

A machine without write permissions which has been preempted finds its registers, code and data completely unchanged when it is rescheduled.

Expression of high-level properties

Similarity

Two SPM states s and t are similar modulo the machine i, $s \sim_i t$, if all machines except i are identical between s and t.

PROVE & RUN

Expression of high-level properties

Similarity

Two SPM states s and t are similar modulo the machine i, $s \sim_i t$, if all machines except i are identical between s and t.

Confidentiality

Let s and s' be two states similar modulo i, and t and t' such that:

- $s, \bar{\pi} \rightarrow^* t$ and $s', \bar{\pi} \rightarrow^* t'$
- i does not run between s et t
- s_i has no data readable from another machine, i.e.
  - s_i is not blocked trying to send a message
  - s_i has no read permissions on its memory

Then, $t \sim_i t'$.

PROVE & RUN