

# ProvenCore: Towards a Verified Isolation Micro-Kernel

Stéphane Lescuyer

Prove & Run

MILS Workshop, Amsterdam, 20/01/2015



**PROVE & RUN**

# Summary

## 1 ProvenCore

- Overview
- Features
- Policies

## 2 ProvenTools

- The Smart language
- Tool-chain
- C code generation

## 3 Proofs and properties

- Refinements
- Properties



# ProvenCore's objectives

## Objectives

- formally prove security properties of a  $\mu$ -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



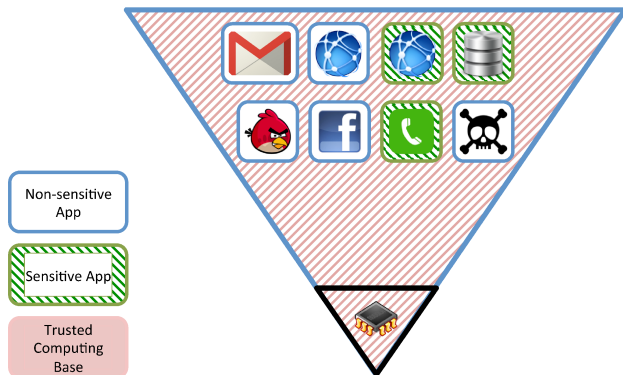
# ProvenCore's objectives

## Objectives

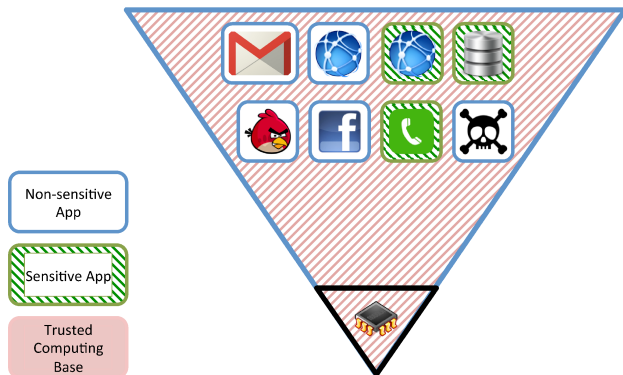
- formally prove security properties of a  $\mu$ -kernel
  - absence of runtime-errors
  - functional specifications
  - integrity / confidentiality
- using IDE and language developed at Prove & Run
- target high-level Common Criteria evaluation
- generate documentation from the specs
- generate executable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



# Typical mobile device architecture



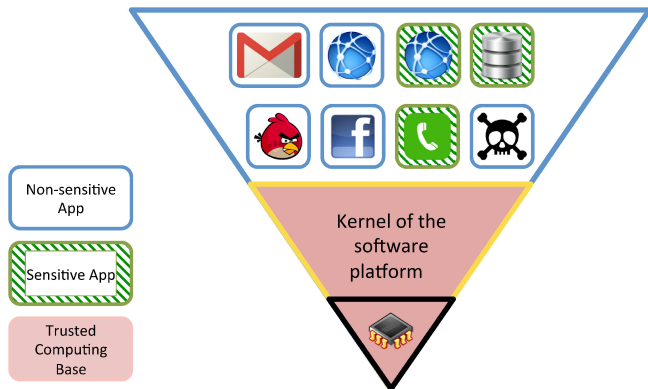
# Typical mobile device architecture



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



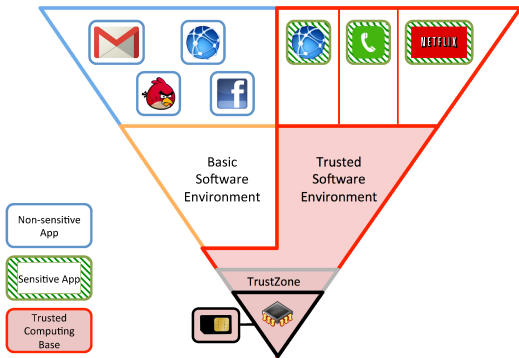
# Securing the rich kernel?



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

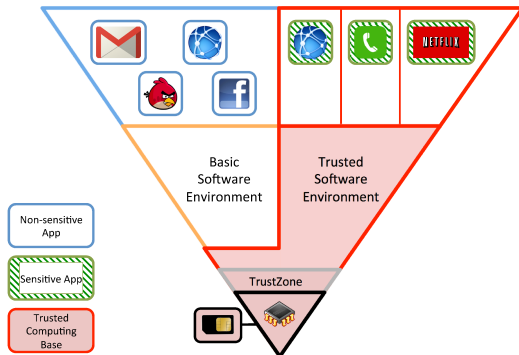


# TrustZone to the rescue





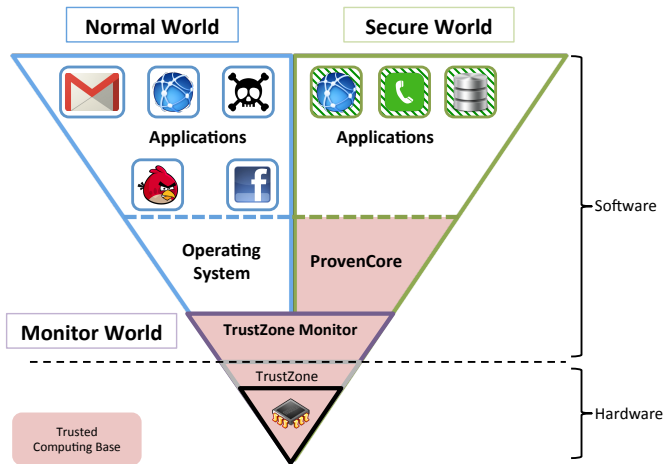
# TrustZone to the rescue!



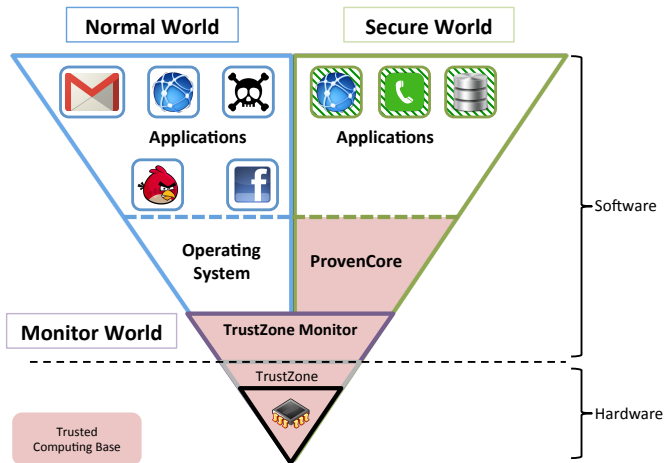
→ the OS on the secure side can be constrained!



# ProvenCore as a secure world OS



# ProvenCore as a secure world OS



- possible safe BYOD policy
- possible to run a TEE on ProvenCore



# Minix heritage

ProvenCore is largely inspired by Minix 3.1



# Minix heritage



ProvenCore is largely inspired by Minix 3.1

## Why Minix?

- $\mu$ -kernel
- well-documented
- mostly POSIX-compliant
- simple yet versatile enough
  - no real-time constraints



# Minix heritage



ProvenCore is largely inspired by Minix 3.1

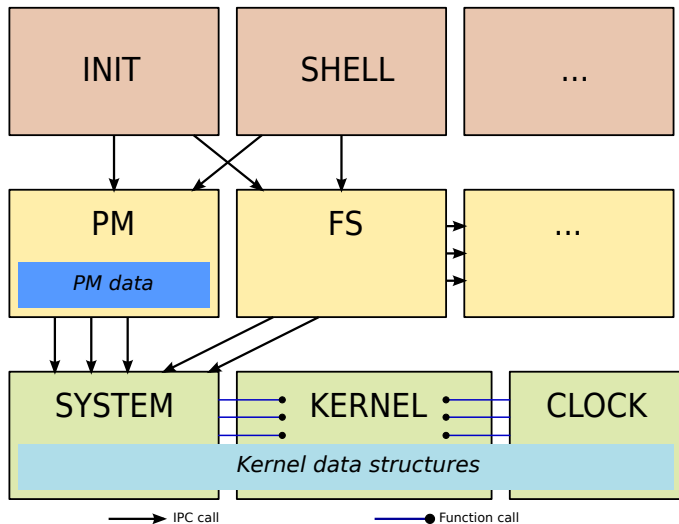
## Why Minix?

- $\mu$ -kernel
- well-documented
- mostly POSIX-compliant
- simple yet versatile enough
  - no real-time constraints

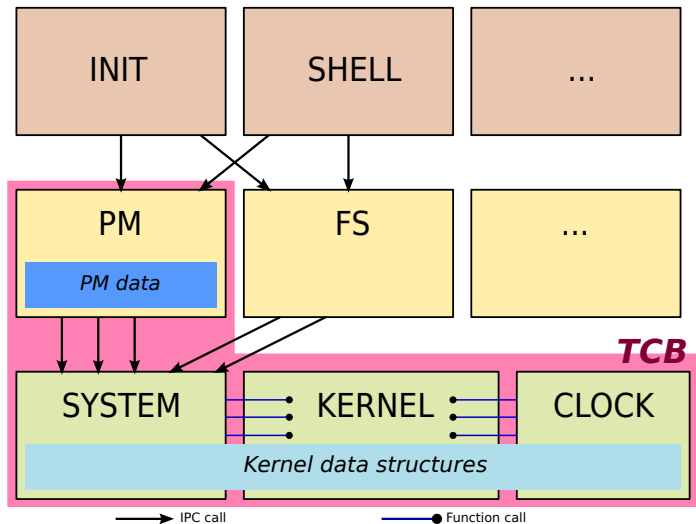
## Still...

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

# The TCB incident



# The TCB incident





# 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

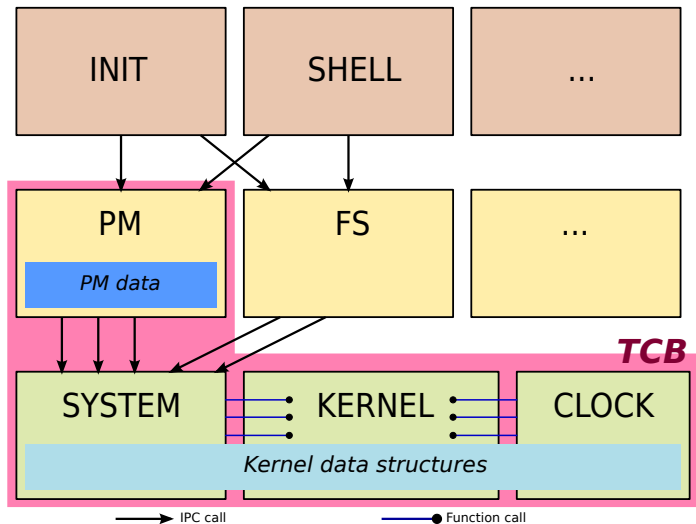
## Non-sequential TCB

- 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

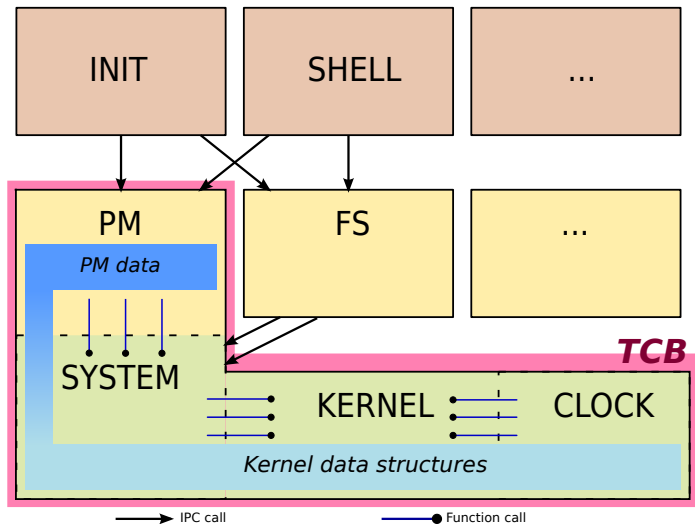
- formal reasoning on a concurrent TCB possible but extremely *hard*
- semi-formal reasoning possible, but unsatisfactory



# Sequential TCB



# Sequential TCB



# 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



# 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**

# Security policies

Resource allocation policy

Access control policy

Information flow policy





# Security policies

## Resource allocation policy

**time** configurable scheduling priority bounds

**RAM** physical quotas can be put on binaries

## Access control policy

## Information flow policy



# 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 algebraic datatypes
- **pure**: only manipulate values



# Prove & Run's Smart language

## Smart

- first-order polymorphic functional language
- used **both** for models and specifications
- built-in algebraic 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



# 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 { ... }
```





# 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 { ... }
```

---

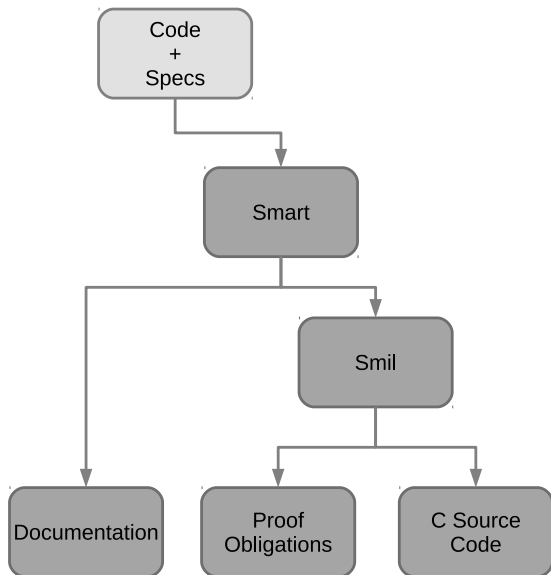
## No side-effect hidden

---

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

---

# Tool-chain overview



PROVE &amp; RUN

## Screenshots

The screenshot shows the ProvenCore IDE interface. The main editor displays the source code for a proof involving vectors. The code includes lemmas for successor, index, and ordinal, along with an inductive proof for the vector type. The right-hand pane shows the proof progress, listing 137 lemmas available and 137 unfolded. A red box highlights an 'Unproved' configuration with depth 3 and width 2, involving index\_spec and out\_of\_bounds lemmas.

```

begin(begin+ && ?index(ZERO, idx+) && idx = begin;
)
public lemma index_succ(nat n, index<S, N, A> _idx)
  // The index of the successor of (n) is the result of applying [[next]]
  // to the index of (n). In particular it succeeds if [[next]] succeeds.
  program { (index<S, N, A> idx, nat an, index<S, N, A> sid) {
    ?index(n, idx) =>
    ?next(idx, idx+) =>
    succ(n, sn+) =>
    ?index(sn, sid+) && idx = sid;
  }
}
public index_spec(nat n, index<S, N, A> idx+) -> [true, out_of_bounds]
  // An explicit predicate that iterates on indices, counting,
  // until it reaches (n) and returns the current index (idx), or
  // raises :out_of_bounds if it runs out of indices. It is
  // meant as an explicit implementation of the implicit [[index(-,+)]].
  program { (nat cur) {
    begin(idx+);
    cur := ZERO;
    while
    - inductive Inv(n, idx, cur)
    - {
      cur = n => return;
      next(idx, idx+);
      succ(cur, cur+);
    }
    raise out_of_bounds;
  }
}
public lemma index_spec_inv_ordinal(nat n, index<S, N, A> idx, nat cur)
  // At any iteration of the loop in [[index_spec]], (cur) is the ordinal
  // of the index (idx).
  program { (nat sid) {
    index_spec_inv(n, idx, cur) =>
    ordinal(idx, sid+) && cur = sid;
  } assuming std.peano
}
public lemma index_def(index<S, N, A> _idx, nat n)
  // [[index_spec]] is a correct alternative to [[index(-,+)]. In particular,
  // they fail at the same time.
  program { (index<S, N, A> idx1, index<S, N, A> idx2) {
    if ?index_spec(n, idx1); then
    ?index(n, idx2) && idx1 = idx2;
  }
  -> else
  -> !?index(n, idx2+);
  } assuming std.peano
}
public index_at(index<S, N, A> _idx, nat n) -> [true, false]
  // An inductive characterization of the fact that (idx) is the (n)-th index
  // of its vector type. It is equivalent to the fact that (n) is the
  // ordinal of (idx).
  // I am not convinced this is very useful tbh. -SL
  inductive
  case Begin: { (index<S, N, A> b) {
    - n = ZERO && begin(+)&& b = idx;
  }
}

```

The right-hand pane shows the proof progress:

- JDM2 prover [15/16]
- ordinal\_length [1/1] (lemmas: 116 available, 1 provided)
- index\_length [2/2] (lemmas: 117 available, 2 provided)
- index\_traversal [3/3] (lemmas: 23 available, 0 provided)
- ordinal\_spec [2/2] (lemmas: 25 available, 0 provided)
- ordinal\_def [2/2] (lemmas: 25 available, 2 provided)
- ordinal\_inj [1/1] (lemmas: 27 available, 1 provided)
- reachable\_from\_last [1/1] (lemmas: 28 available, 1 provided)
- ordinal\_last [1/1] (lemmas: 129 available, 1 provided)
- index\_ZERO [2/2] (lemmas: 30 available, 2 provided)
- index\_succ [2/2] (lemmas: 32 available, 2 provided)
- index\_spec [2/2] (lemmas: 36 available, 0 provided)
- index\_spec\_inv\_ordinal [1/1] (lemmas: 136 available, 1 provided)
- index\_def [2/3] (lemmas: 137 available, 3 provided)
- 137 lemmas available
- Unfold of index\_spec(n, idx1) ->
- Unfold of index\_spec(n, idx1) ->
- Unproved (configuration: depth 3, width 2)
  - index\_spec\_inv(n, idx1, n) -> [unfold of index\_spec(n, idx1)]
  - index(n, idx2) -> [out\_of\_bounds] ->
  - Unfold of index\_spec(n, idx1) -> [out\_of\_bounds] ->
  - length\_spec [2/2] (lemmas: 42 available, 0 provided)
  - length\_def [1/1] (lemmas: 42 available, 1 provided)
  - vector\_traversal [6/6] (lemmas: 45 available, 0 provided)

The bottom left corner shows the SMDT Prover Console:

```

Starting proof of 1 module
+ Project [TDS]:
Starting proof of 1 module
+ Project [std]:

```

The bottom left corner also shows the status: Action Undo ran successfully

## Screenshots

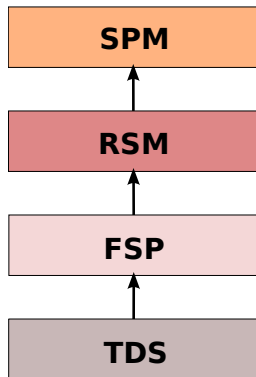
The screenshot displays the ProvenCore IDE interface. The main editor shows the source code for the `vector` module, including functions like `index_succ`, `index_spec`, `index_spec_inv_ordinal`, `index_def`, and `index_at`. The code is annotated with comments and uses the `std` namespace for standard operations.

The right-hand pane shows the documentation for the `vector` module, which describes it as an abstract type of efficient contiguous arrays. It details the `index` and `get` methods, explaining their behavior and constraints. The `index` method returns the element at a given index, while `get` sets the element at a given index to a specific value.

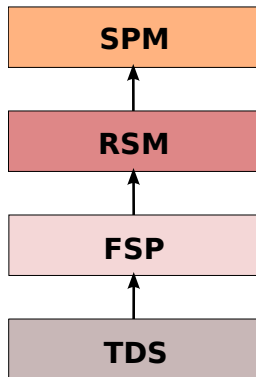
The left-hand pane shows the project structure, including various modules like `hints`, `smil`, `src`, `std`, `arith.sm`, `array.sm`, `array.sm`, `bool.sm`, `conversions.sm`, `eq.sm`, `functions.sm`, `iteratior.sm`, `list.sm`, `map.sm`, `memory.sm`, `option.sm`, `pair.sm`, `peano_const.sm`, `peano.sm`, `prelude.sm`, `set.sm`, `string.sm`, `vector.sm`, and `vector_vector.sm`. The `vector_vector.sm` module is currently selected.

The bottom-left pane shows the SMDT Prover Console, which displays the progress of the proof process, including the starting of a proof for the `vector` module and the project.

# Refinements



# Refinements

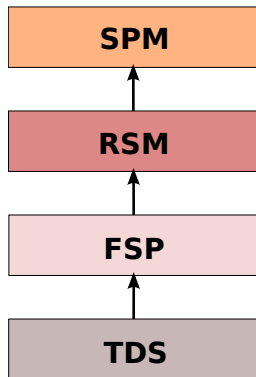


TDS Target Design



PROVE & RUN

# Refinements

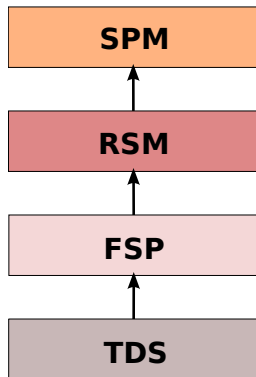


FSP Functional Specifications

TDS Target Design



# Refinements



**RSM** Refined Security Model

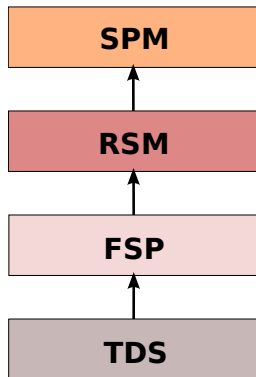
**FSP** Functional Specifications

**TDS** Target Design





# Refinements



**SPM** Security Policy Model

**RSM** Refined Security Model

**FSP** Functional Specifications

**TDS** Target Design



# Forward Simulation

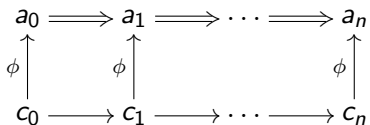
$\phi$  a **view** from concrete states to abstract states

$$\begin{array}{ccc} a_0 & \Longrightarrow & a_1 \\ \uparrow \phi & & \uparrow \phi \\ c_0 & \longrightarrow & c_1 \end{array}$$



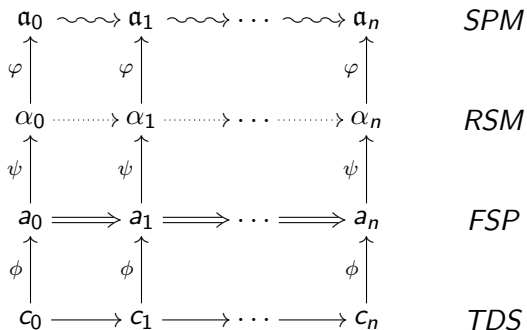
# Forward Simulation

$\phi$  a **view** from concrete states to abstract states



# Forward Simulation

$\phi$  a **view** from concrete states to abstract states



# FSP characteristics

## FSP characteristics

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



# FSP characteristics

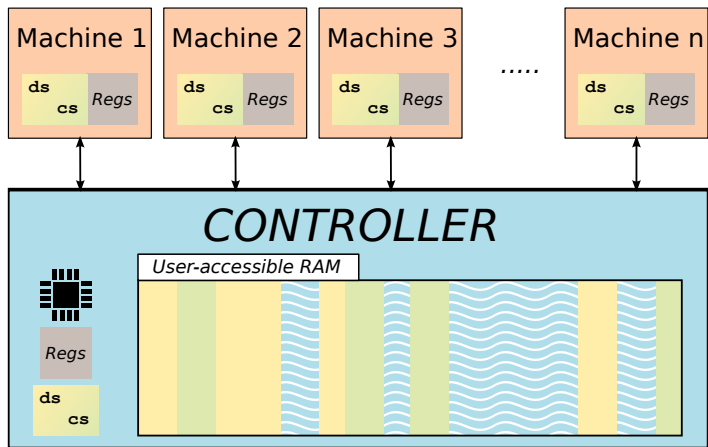
## FSP characteristics

- 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



# 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

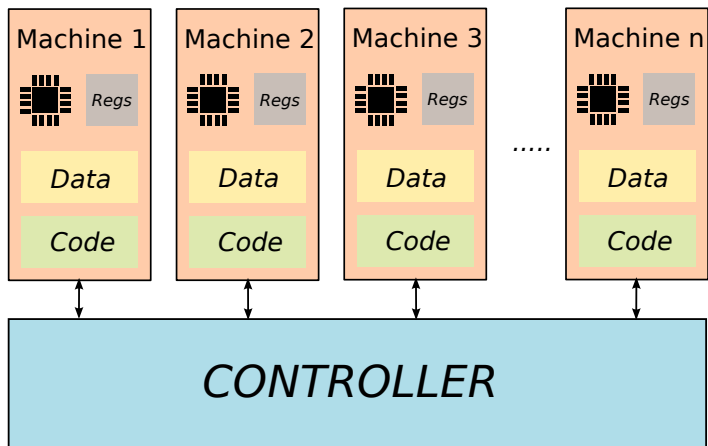
## Security model characteristics

- 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



# Schematic view of the SPM



# 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

- $\mu$ -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



# Appendices



PROVE & RUN

# In-place updates, example 1

---

```
@Ghost  
type ram = ...
```

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



# 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 $\rightarrow$ FSP (1/2)

TDS

---

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

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

---



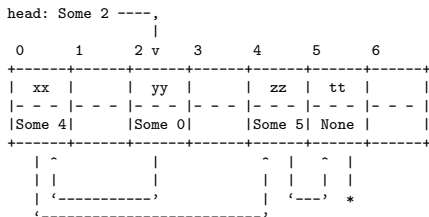
PROVE & RUN

# Abstraction example TDS $\rightarrow$ 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;  
}
```

```
head: Some 2 ----,  
      |  
0     1     2 v  3     4     5     6  
+-----+-----+-----+-----+-----+-----+  
|  xx  |     |  yy  |     |  zz  |  tt  |     |  
| - - -| - - -| - - -| - - -| - - -| - - -| - - -|  
|Some 4|     |Some 0|     |Some 5| None |     |  
+-----+-----+-----+-----+-----+-----+  
      | ^ |     |     | ^ | ^ | ^ | |
      | |     |     | | | | |  
      | '-----' |     | | | | |  
      |-----' |     | | | | |  
      |-----' |     | | | | |  
      |-----' |     | | | | |
```

## Invariants

- all indices valid
- no cycles

PROVE & RUN

## Abstraction example TDS $\rightarrow$ FSP (2/2)

FSP

---

```
struct state {  
  list<t> cells;  
  t current;  
}  
  
type list<A> =  
| Nil  
| Cons(A a, list<A> l);
```

---



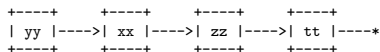
PROVE & RUN

## Abstraction example TDS $\rightarrow$ FSP (2/2)

FSP

---

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



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

---



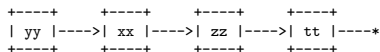
PROVE & RUN



# Abstraction example TDS $\rightarrow$ 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  $\Rightarrow$  sorted in TDS

# 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  $t$ .



# 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,  $t \sim_i t'$ .