Formal Proof of a Secure OS Full Trusted Computing Base

Dominique Bolignano

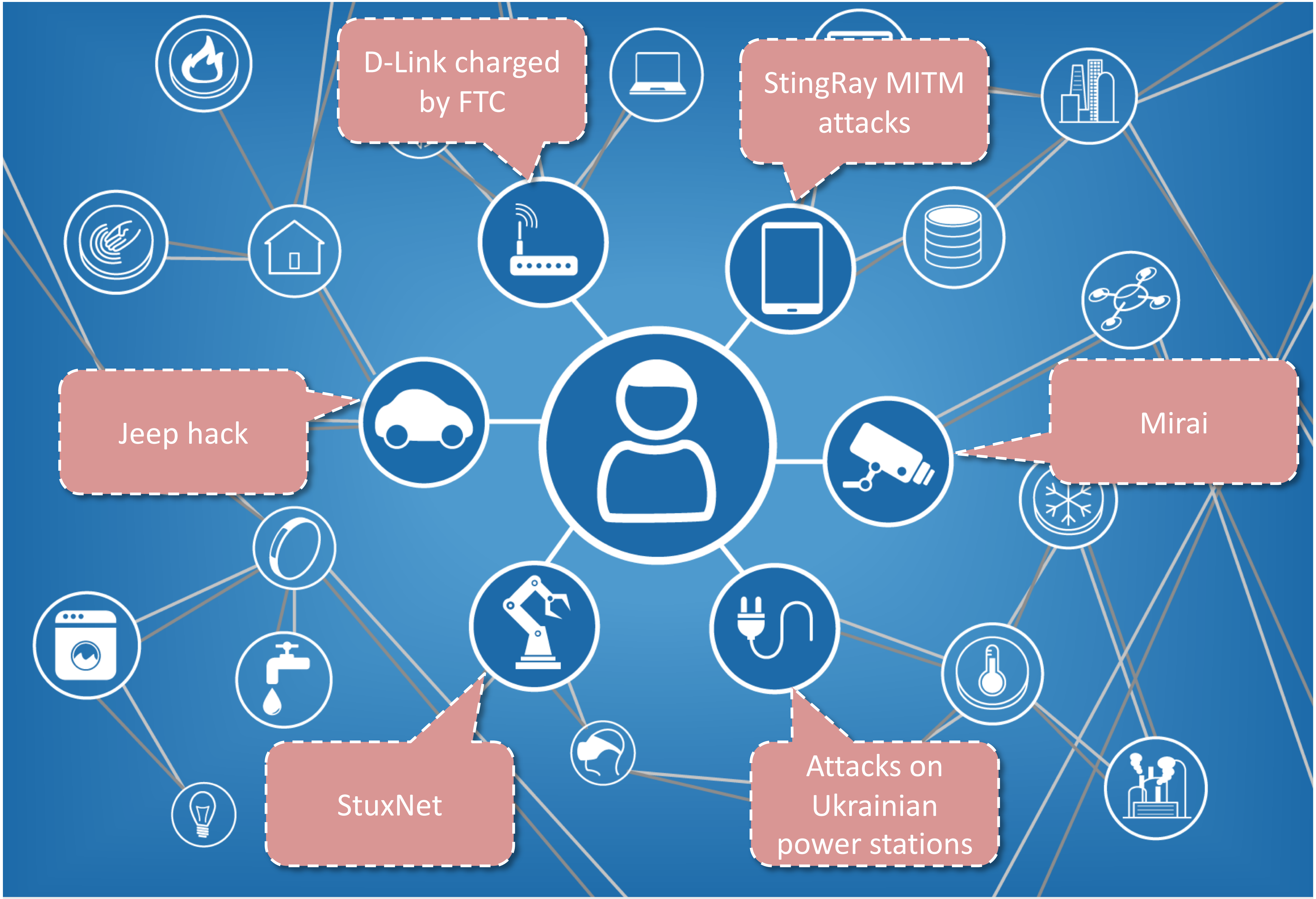
77, avenue Niel, 75017 Paris, France

contact@provenrun.com

Context and Objectives

Context

- Context :
  - Use of formal methods for security during a few decades, in particular in the context of Trusted Logic.
  - First Common Criteria EAL7 certification achieved for a smartcard OS, a Java Card environment and hardware. Formal verification of a bytecode verifier and its linking phase, etc. Deployed in billions.
  - First TEE (secure OS). Also deployed in billions.
- ProvenCore project started in 2009.
  - Development of a formally proven secure OS (proved down to the code),
  - Certifiable at the highest levels of security,


D-Link charged by FTC
StingRay MITM attacks
Jeep hack
Mirai
StuxNet
Attacks on Ukrainian power stations

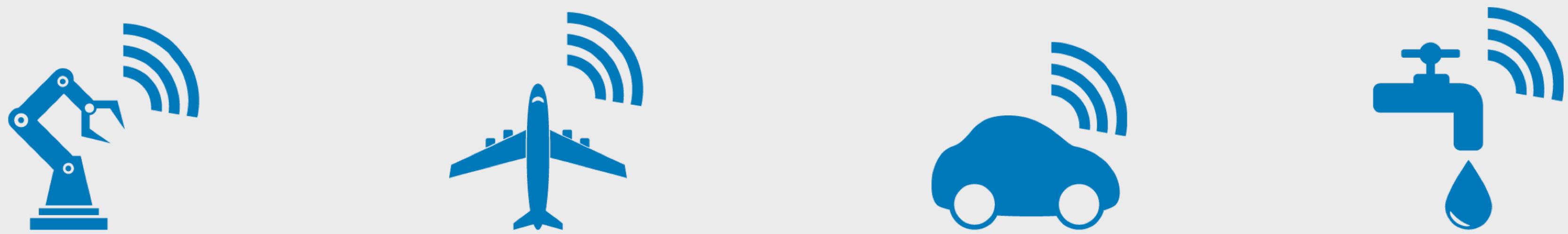
Prove & Run's answer to the challenge

- **Two critical off-the-shelf software components:**
  - ***ProvenCore*** : microkernel proven and certified for security (ARM Cortex A, Risc V, ARM Cortex M).
  - ***ProvenViso[illegible]*** [illegible]ecure hypervisor

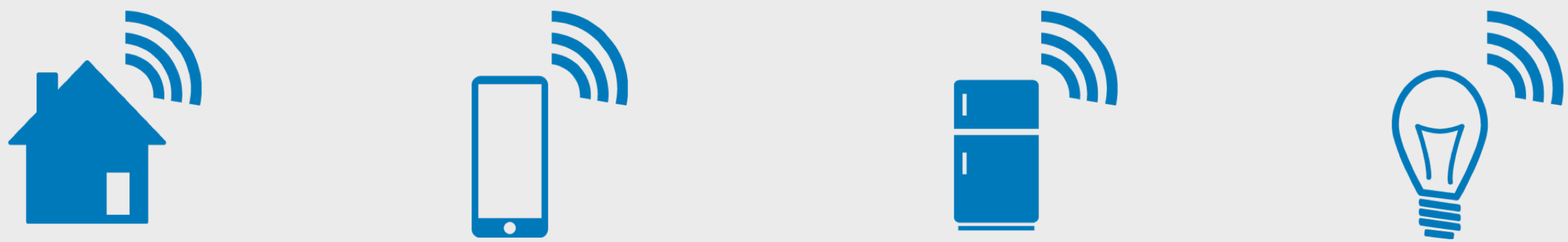
Designed to ha[illegible] TCB (Trusted Computing Base) that is <u>as close as pos[illegible]o zero-bug</u>.

Highest Level of Security ever reached
(CC EAL7 augmented) for Cortex A
World premiere

Prove & Run Value Proposition



We provide **cost effective off-the-shelf software** solutions that **dramatically improve the level of security** of your Connected Systems/Devices so as to protect them **against remote cyber-attacks**



Security certification Schemes

- Many security certification schemes exist, and many more are to come,
- The Common Criteria, which is an ISO standard is based on seven levels from EAL1 to EAL7,
- Security schemes generally have a correctness part and a robustness one.
- Correctness addresses the complete process (not only development).
- EAL7 requires in its correctness part the use formal methods down to the low level design.
- We have used formal methods down to the code.

Need for security certification

- **Exhaustive validation by trusted third party,**
  - You cannot usually check everything by yourselves,
  - You cannot take what the developer tells you for granted,
  - You need to compare the levels of security of different products,
- **Along an extensive certification scheme, i.e. The Common Criteria at its highest level :**
  - Important to not only cover the OS, but also its maintenance, its initialization, its installation, its provisionning, the associated organizational policies, etc.
  - Highest level needed because of the value at stake and the connectivity, i.e. highly profitable business models for hackers,

This is about Trust!

- **What would an organized attacker do with a given budget (automotive, avionics, ..) :**
  - **The objectives is to be able to resist to remote attacks that can be performed with a budget typically over 10 M€. (1 to 100)**
    - Identification phase of the attack
    - Exploitation phase of the attack,
- **The general architecture should be well balanced.**
- **Some of us spent most of their time, analyzing and improving security architectures in various areas (IoT, cloud, chip security architecture and firmware, etc.)**
- **ProvenCore is the main tool we used to secure such architectures.**

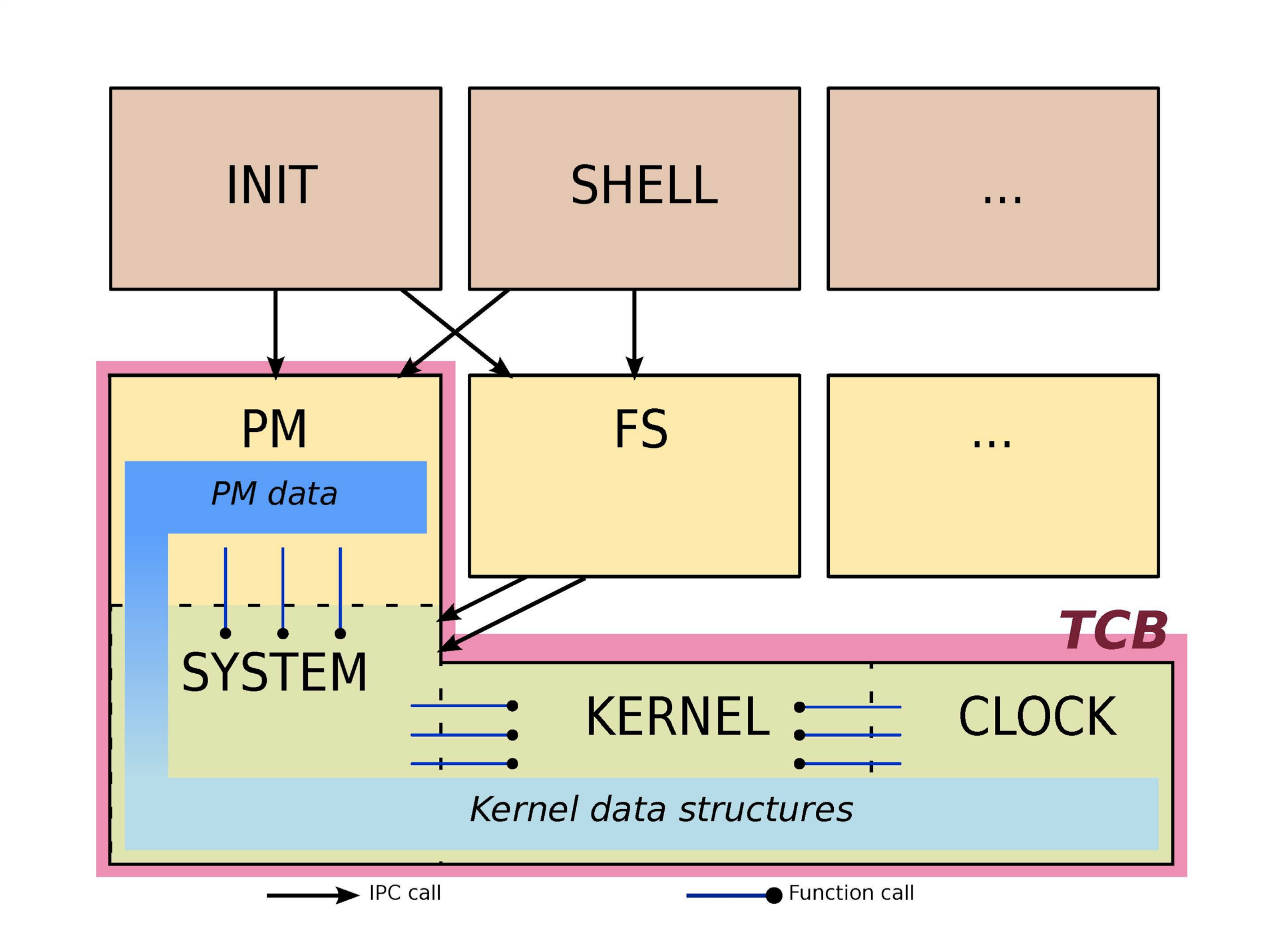
Our Strategy

- Maximize security level and more importantly the level of trust that can be achieved (for a given effort/budget),
- Use a very large base of use cases for defining functional and security requirements, and improving and assessing their adequacy.
- Stepwise approach.

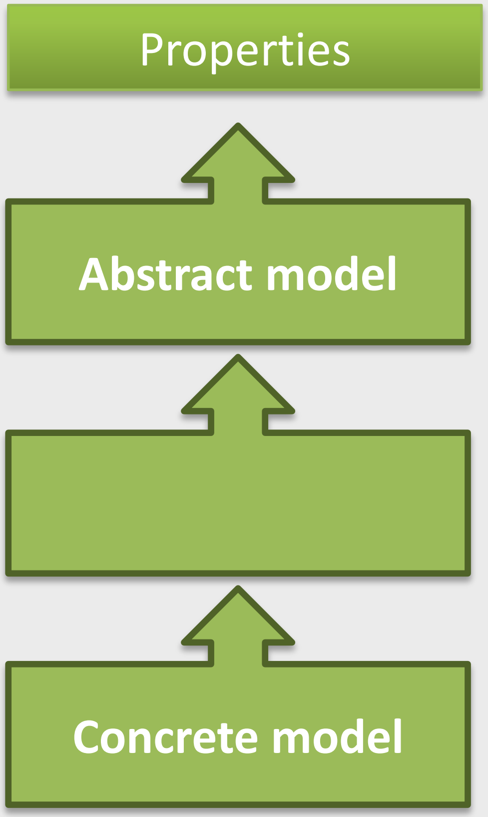
Need for specific functionalities

- **Access Control mechanisms between various applications (i.e. processes, so called Trusted Applications) themselves, between applications and peripherals,**
  - So as to enforce constraints on the flow of information to the applications themselves,
  - Static and enforced by ProvenCore based on simple access control matrix,
  - *Transferable tokens*.
- **High level APIs for applications,**
- **High level security services as applications (secure storage, cryptographic libraries, etc.).**

Trusted Computing Base

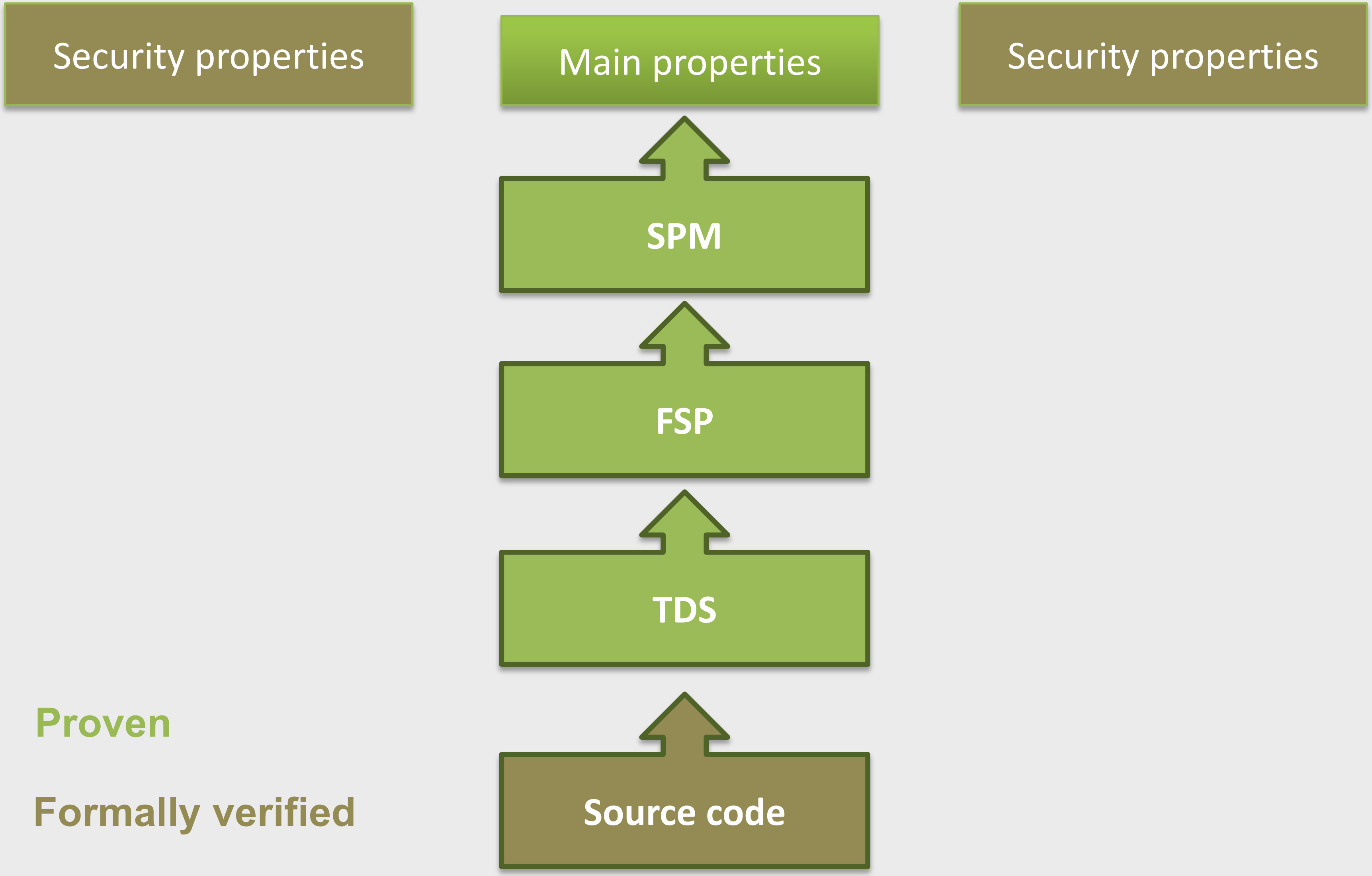


Refinement Proofs

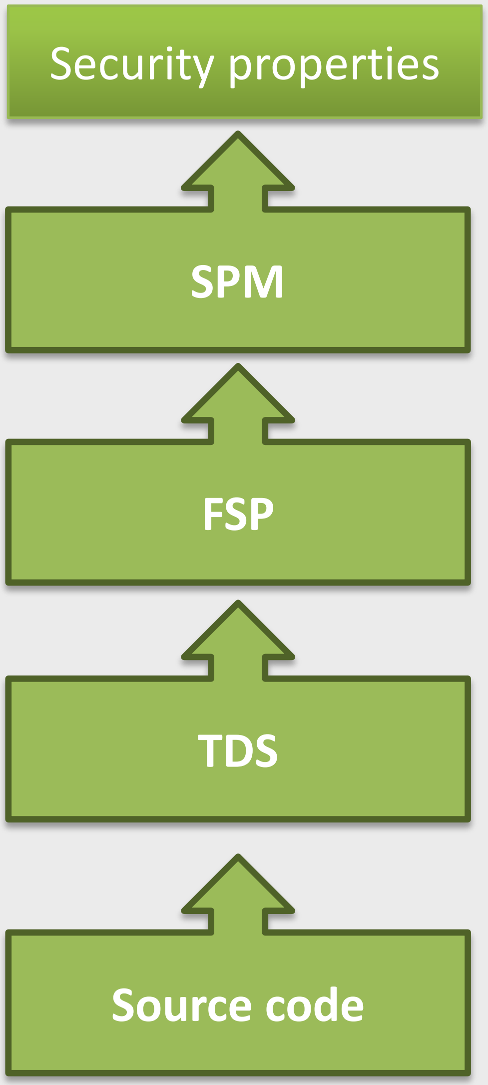


- One (or a few) abstract models
- Formal properties expressed at the highest level
- Properties should be as simple as possible to understand (see "ProvenCore: Towards a Verified Isolation Micro-Kernel", Stéphane Lescuyer, 10th HiPEAC Conference, 2015

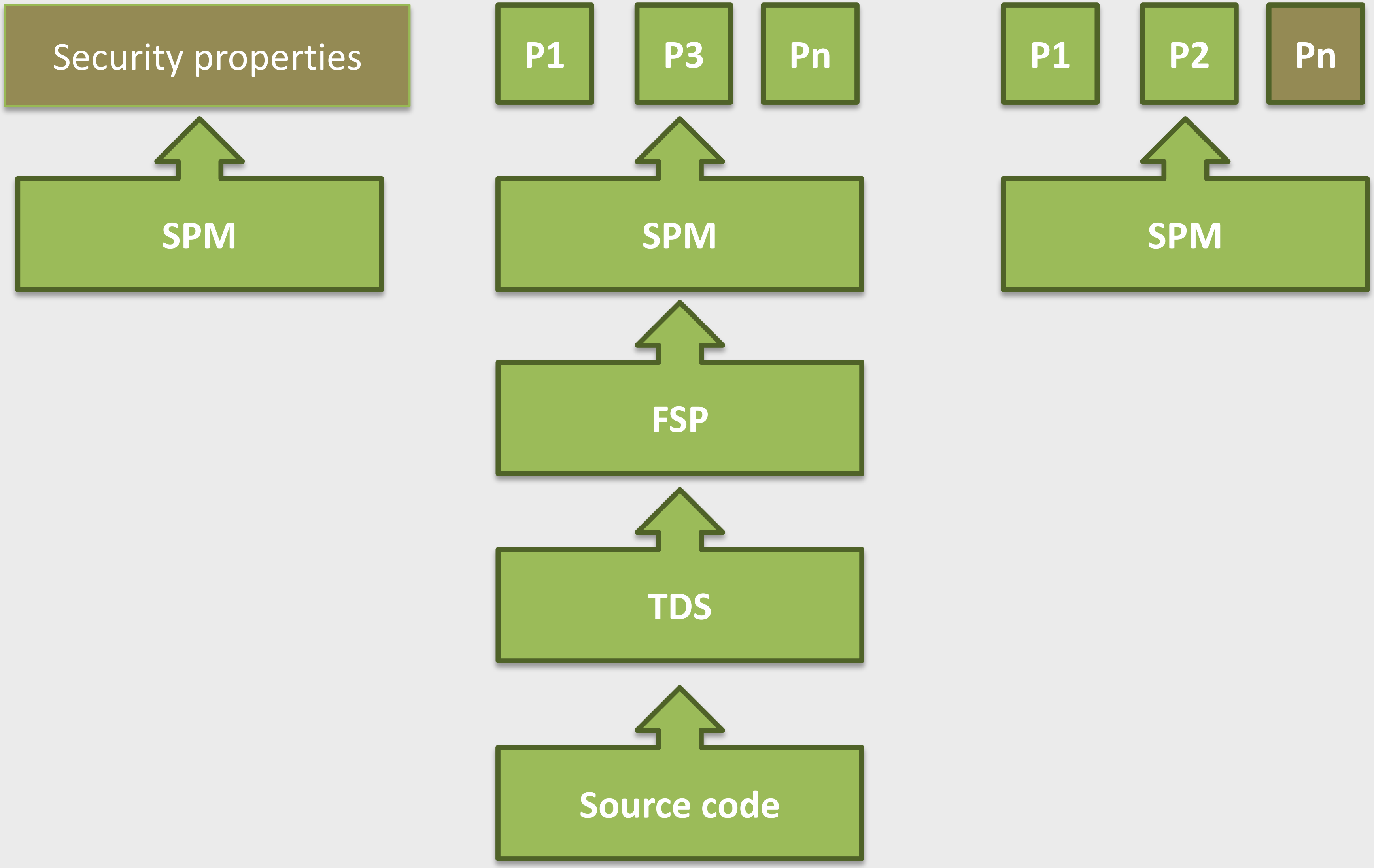
Refinement Proofs and Security Schemes



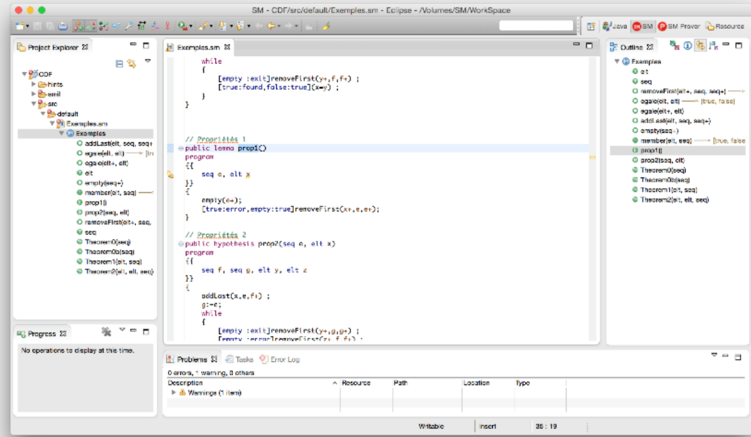
ProvenCore Case

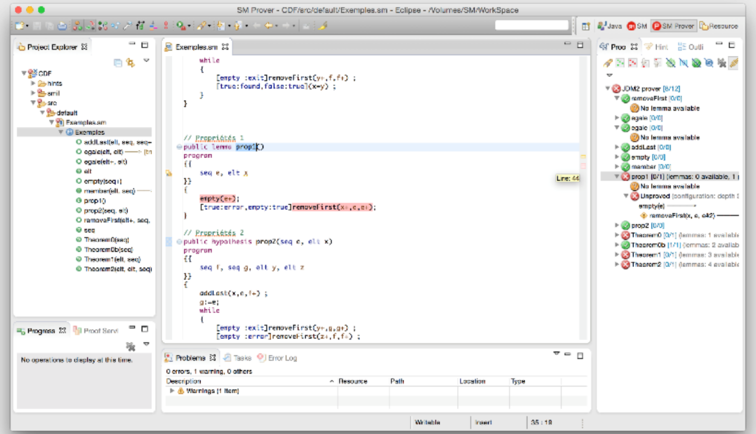


Modelling a microkernel: Properties



SMART development toolchain





Development environment

Prover: Eclipse plugin

P&R Intermediate Language: SMIL

Generator (source code and documentation): Eclipse plugin

Source Code

- Compilable
- C, Java, etc.

Certification Documentation Tests

- CC
- DO-178
- etc.

Automated

SMART language specificities

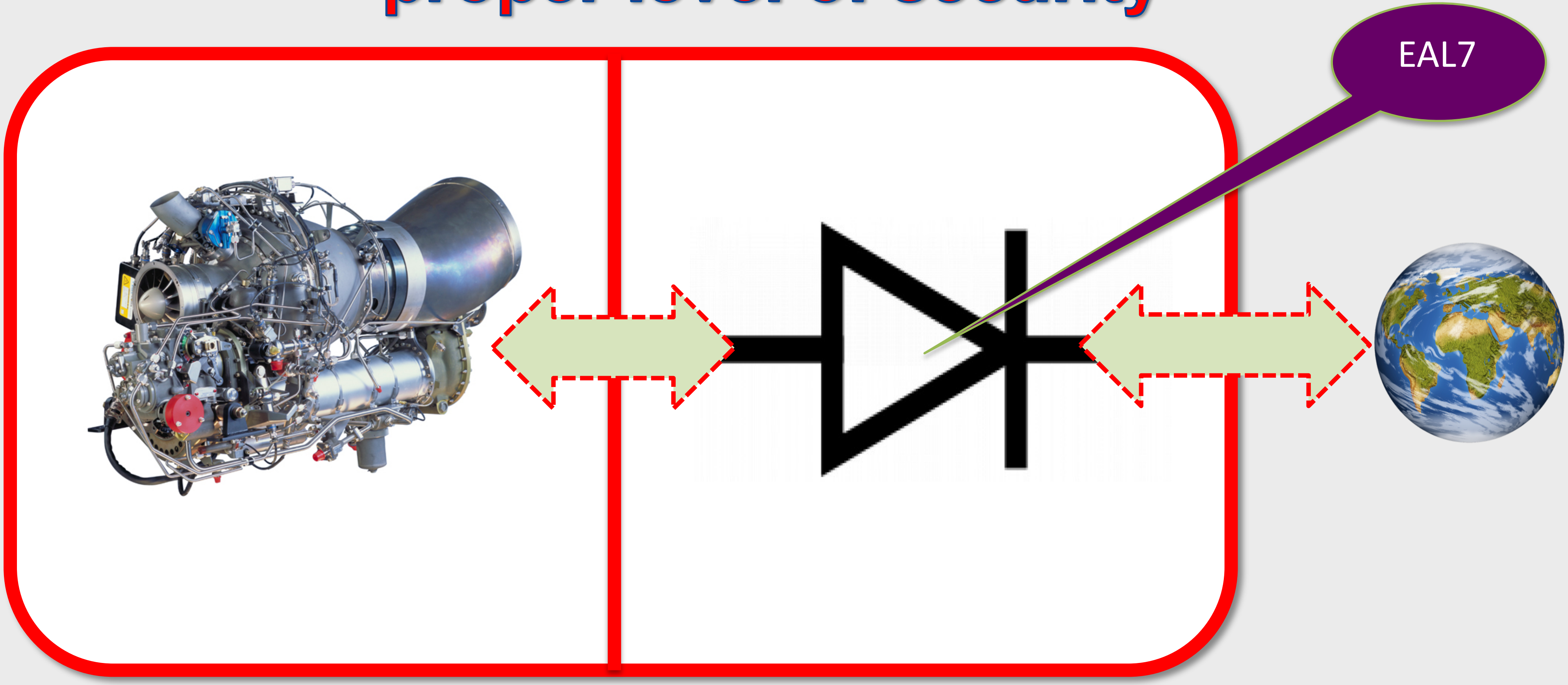
- Built to meet the identified requirements of applying formal methods at a large scale:
  - Usable by developers at high abstraction level but also and more importantly on the lowest levels
  - Allow developers to find (and rely on) paradigms that they usually use for either development, debugging or testing
  - Force developers to answer the right questions while coding and allow them to easily formalize those questions

- Language with a small and simple subset with the addition of clearly identified syntactic sugaring

Typical Use Case for ProvenCore

Filters / Applicative Firewalls

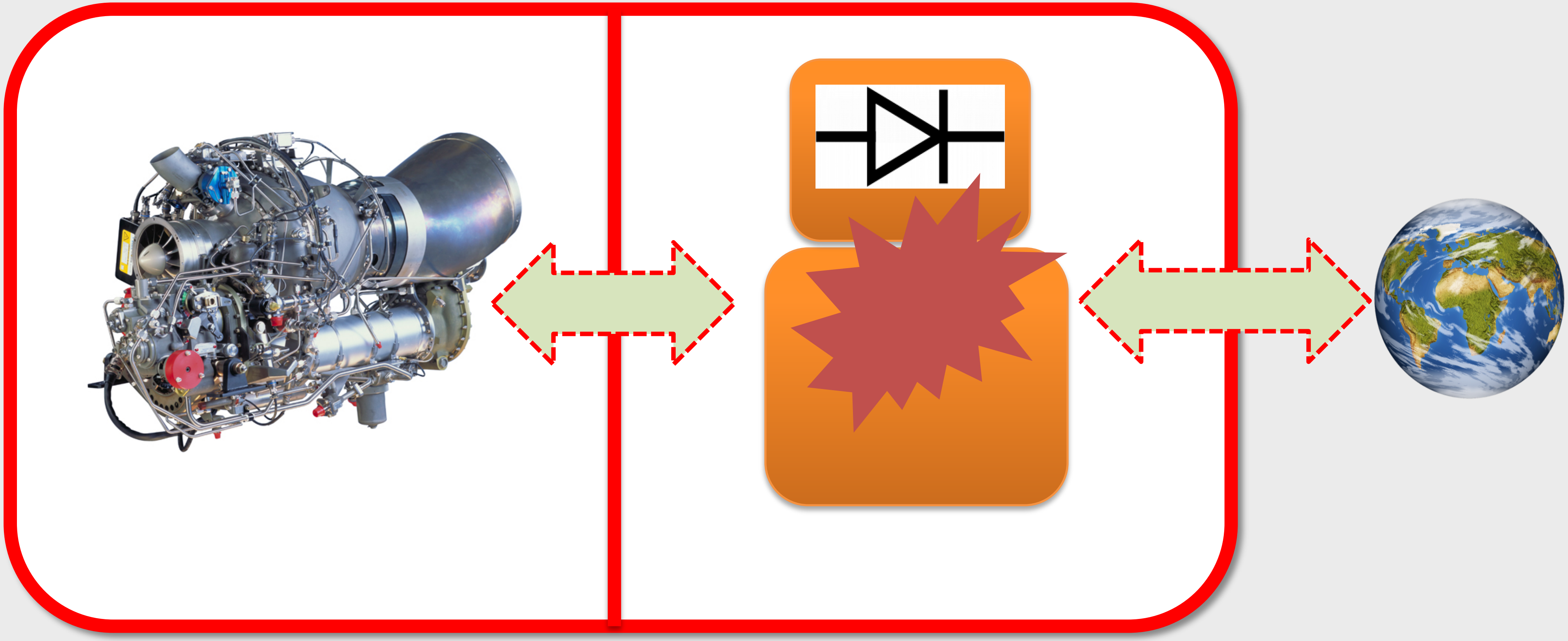
Only hardware based filter can reach proper level of security



Proven and Certified Secure Isolation

Filters / Applicative Firewalls

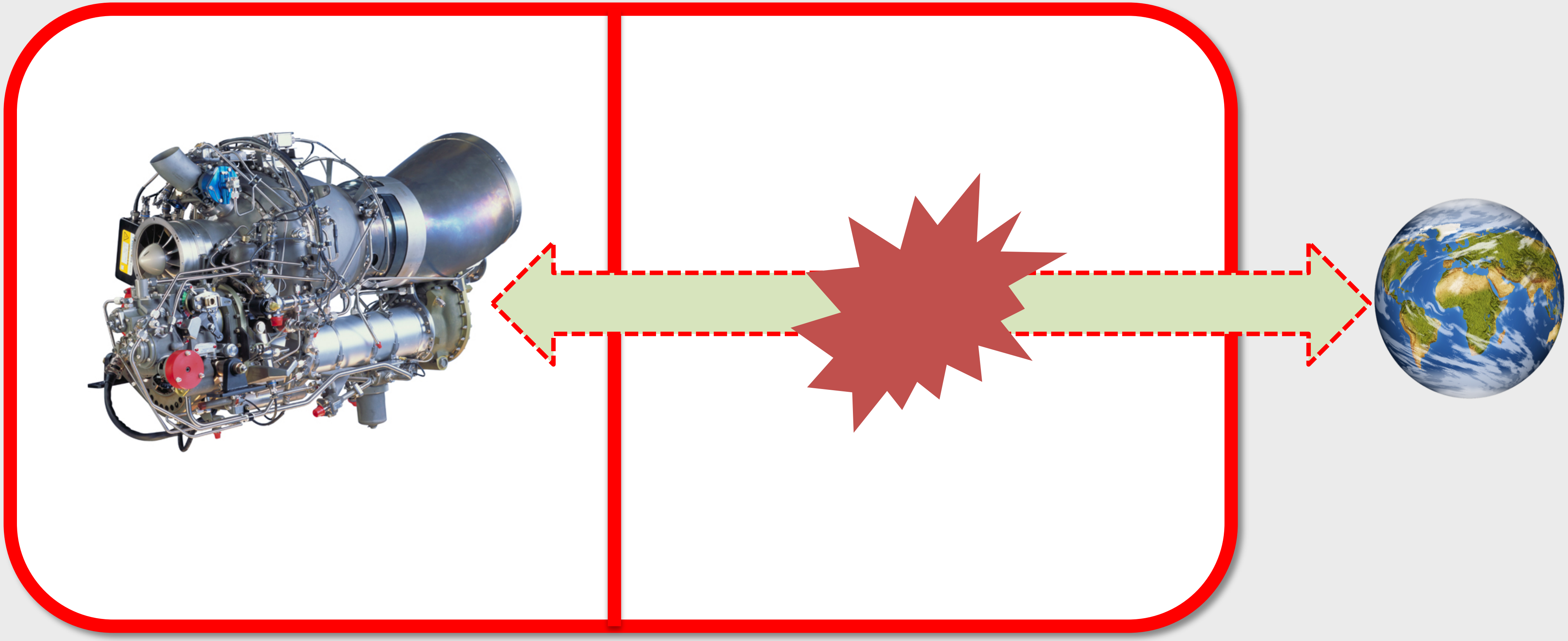
Only software can allow to achieve proper functionality …



Proven and Certified Secure Isolation

Filters / Applicative Firewalls

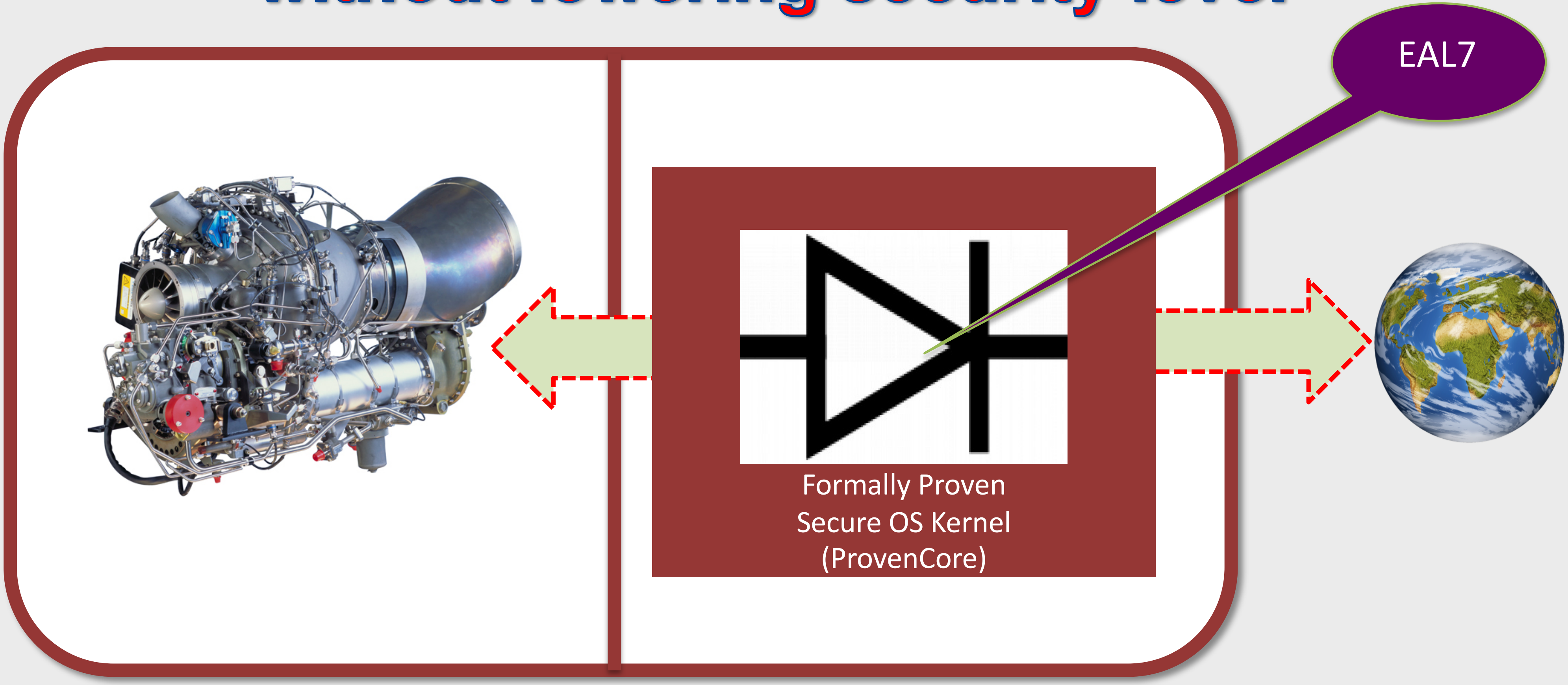
But not proper level of security



Proven and Certified Secure Isolation

Filters / Applicative Firewalls

First step : from hardware to software without lowering security level

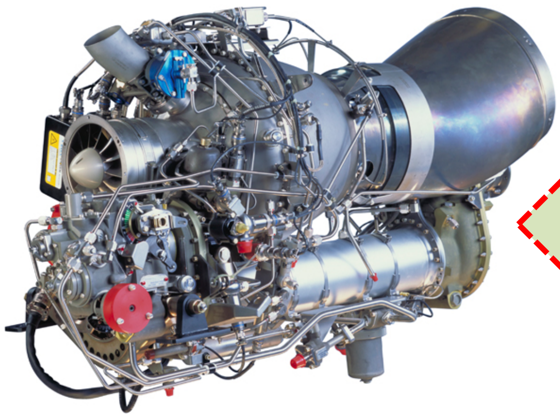


Proven and Certified Secure Isolation

Filters / Applicative Firewalls

Second step : tuning functionality without lowering security level

EAL7

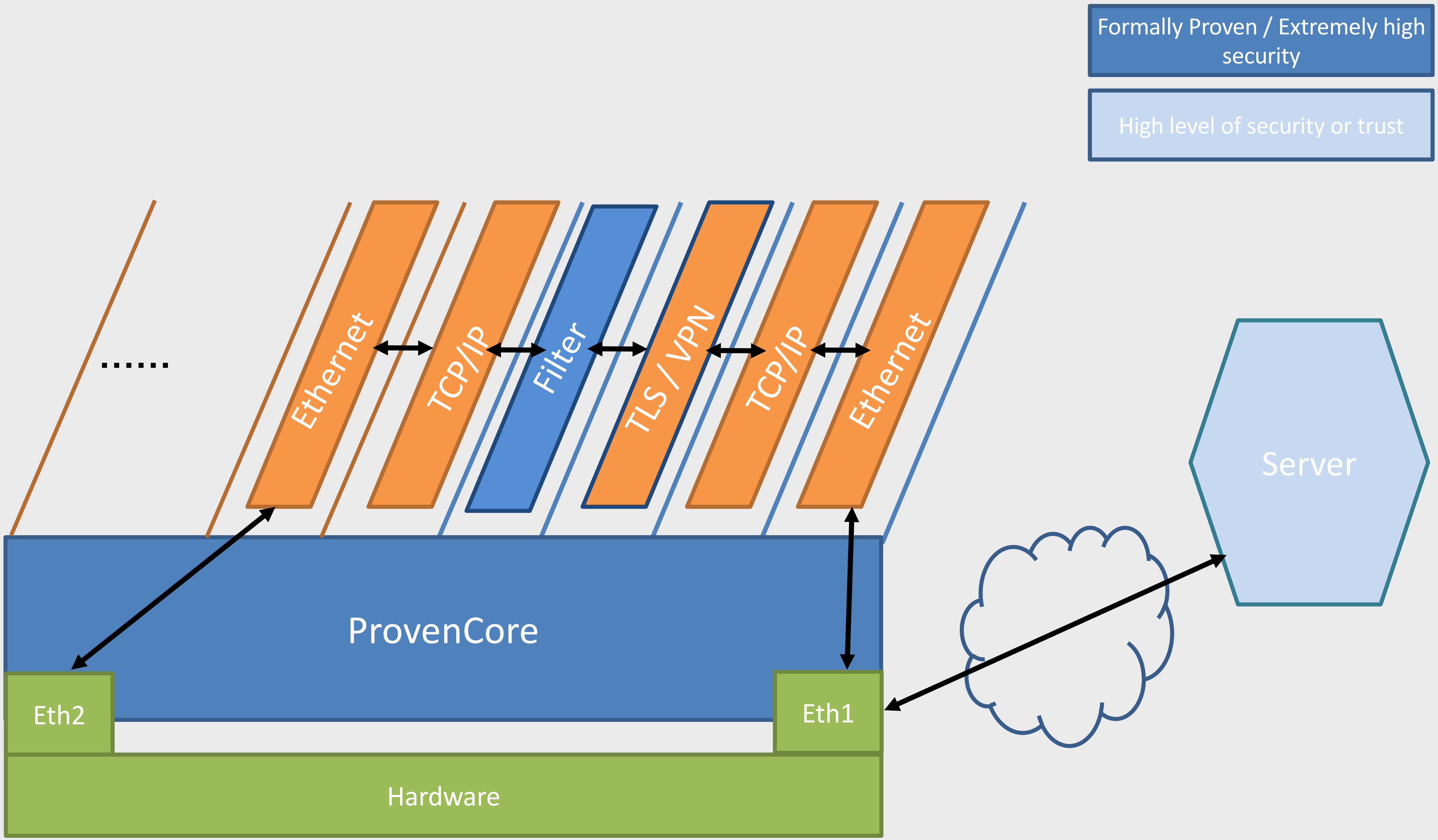


Security Policy Enforcement

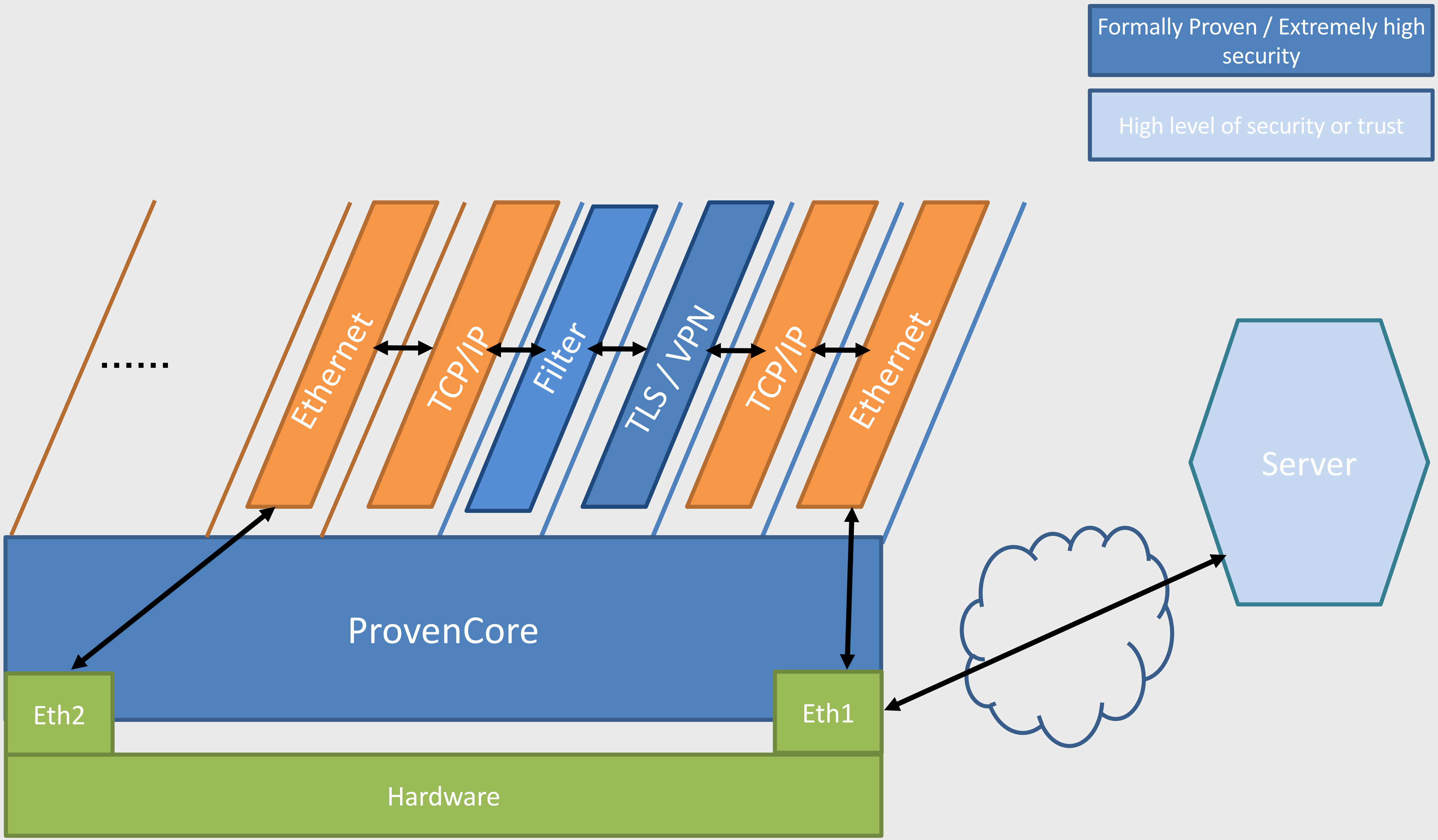
Formally Proven Secure OS Kernel (ProvenCore)

Proven and Certified Secure Isolation

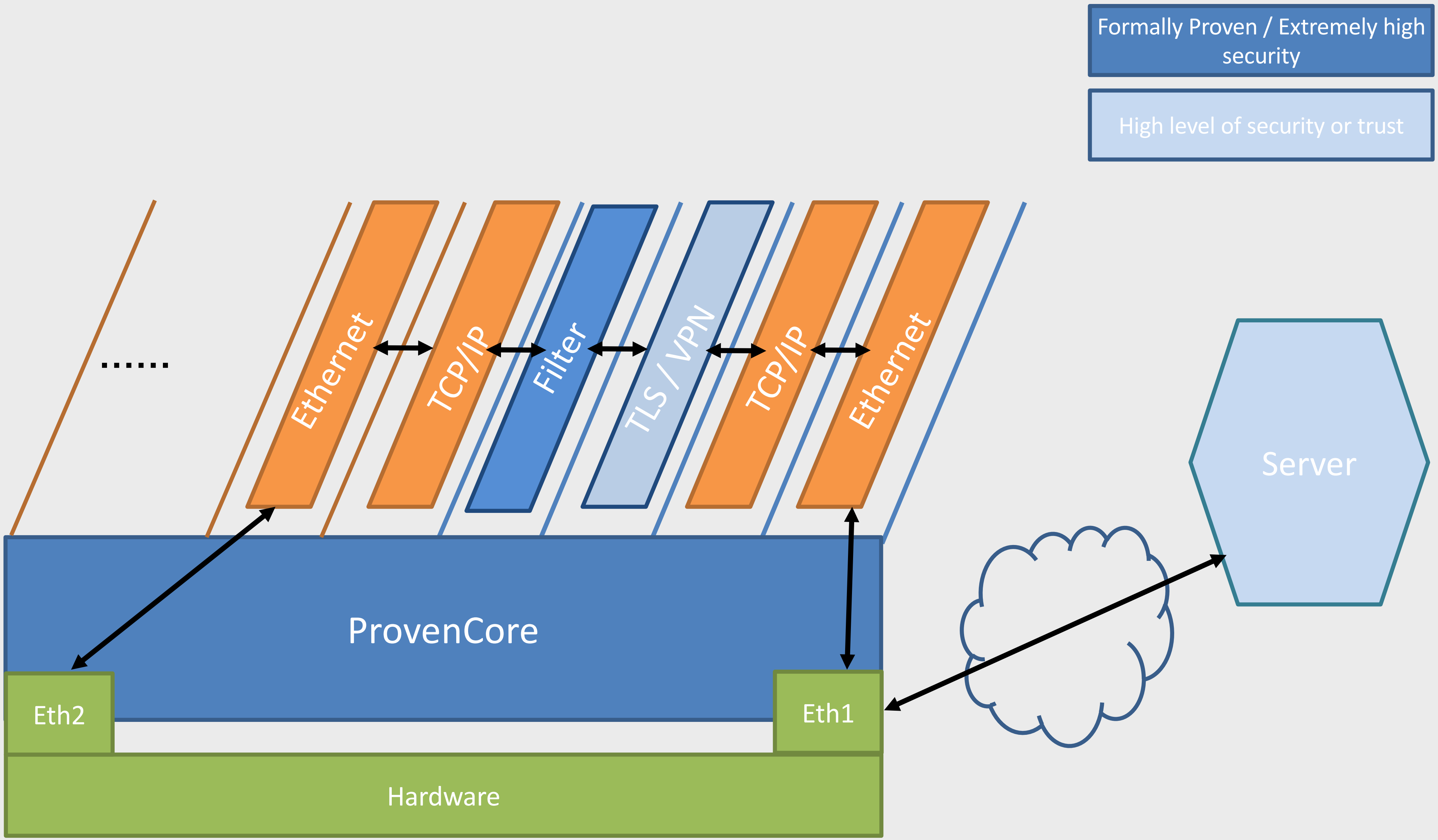
Gateway with secured Filtering



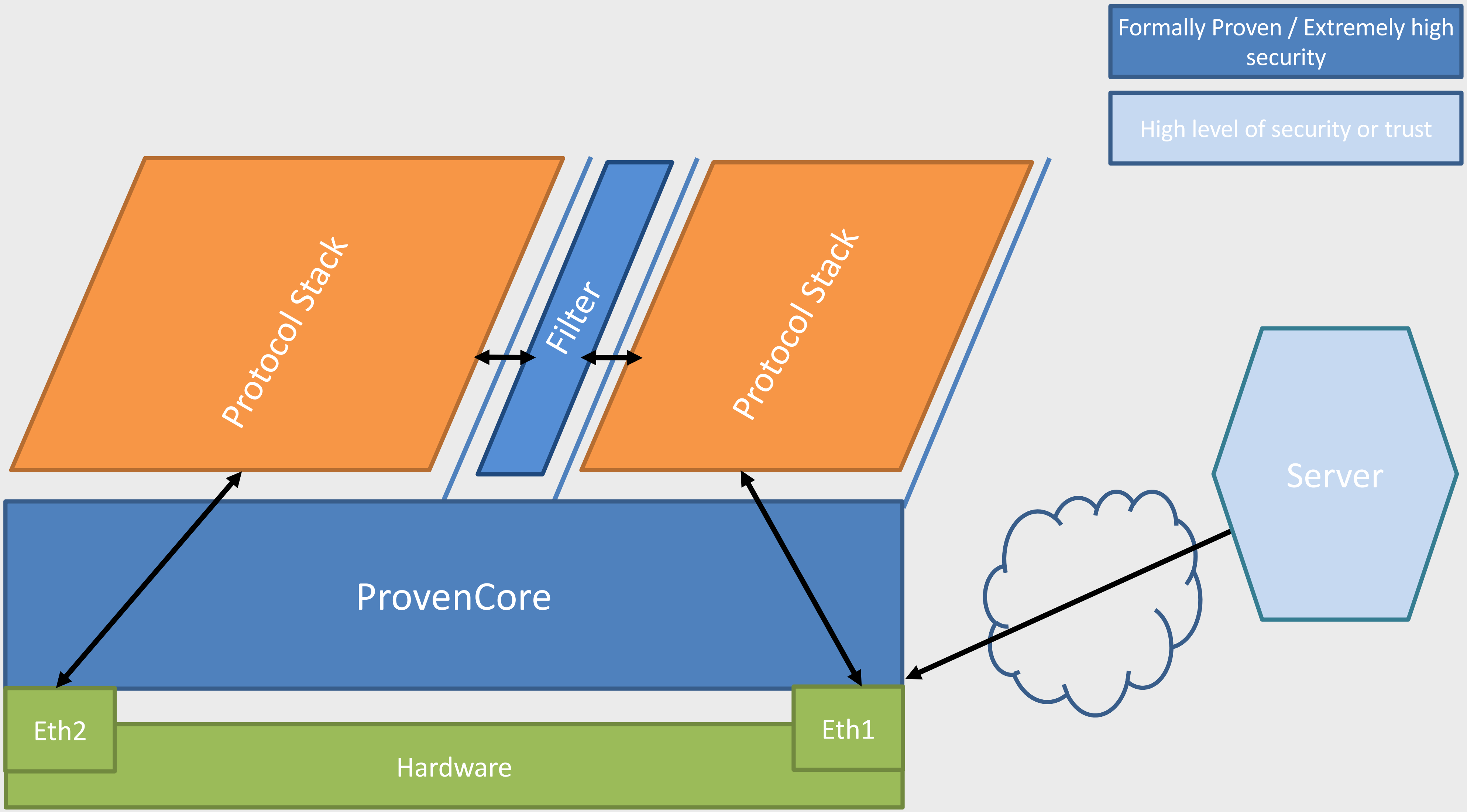
Gateway with secured Filtering



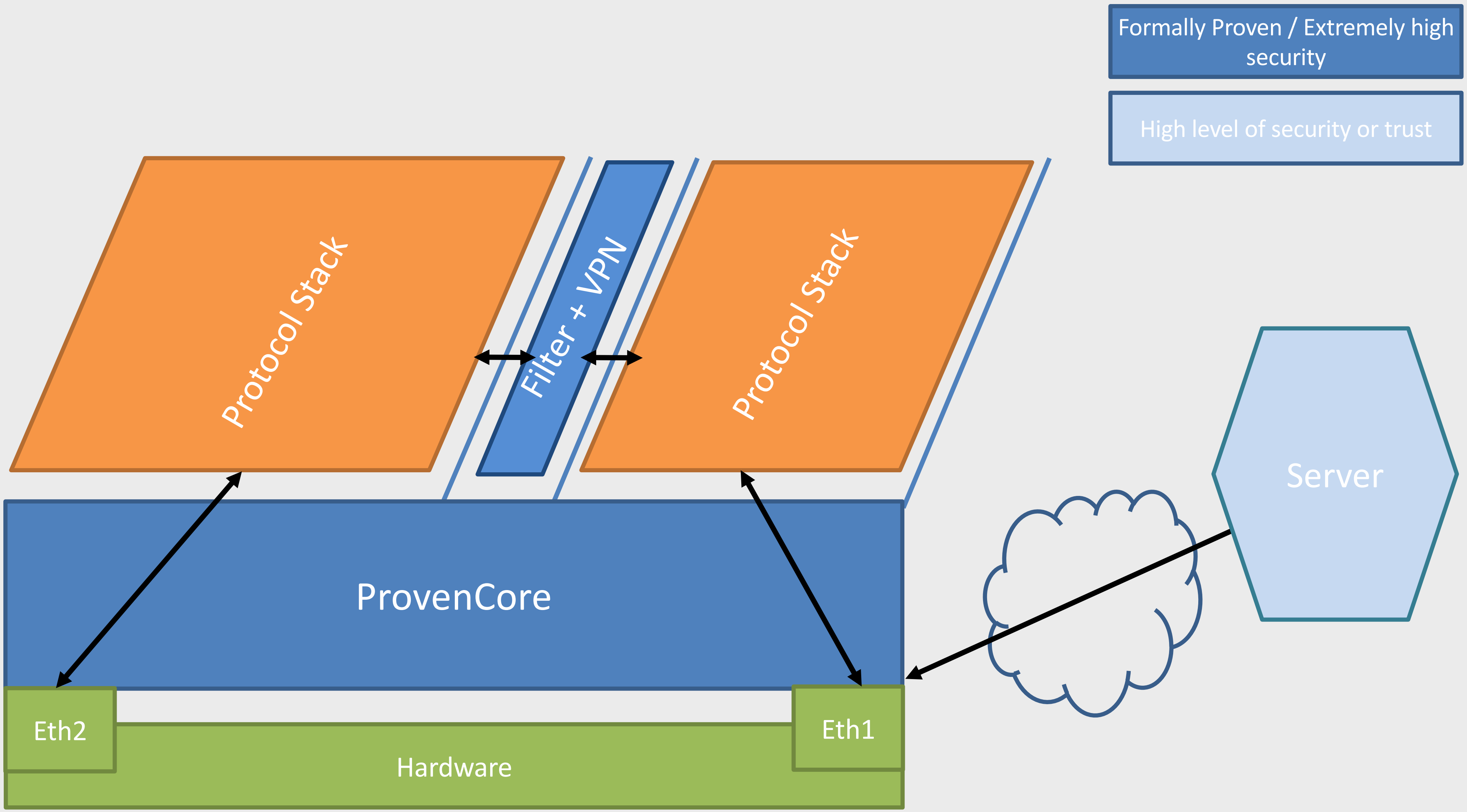
Gateway with secured Filtering



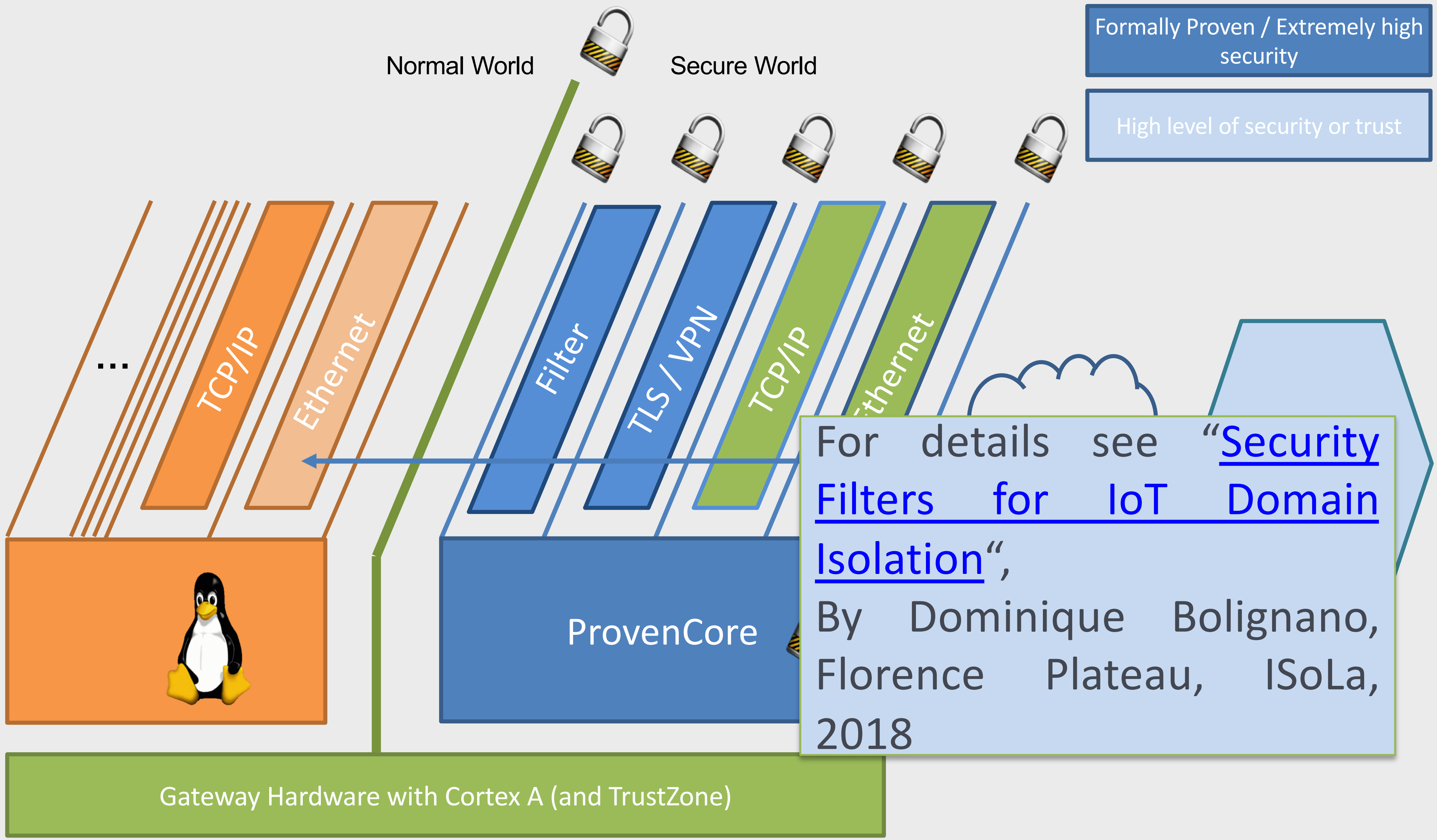
Gateway with secured Filtering



Gateway with secured Filtering



Gateway with secured Filtering



For details see "Security Filters for IoT Domain Isolation",
By Dominique Bolignano, Florence Plateau, ISoLa, 2018

Smart Language

SMART language specificities

- Functional (values manipulation) but with an imperative style
- Can be used at different abstraction levels
- Functions are partial
- Possible to associate proof obligations with logical paths in the execution graph
  - Proofs displayed as symbolic debugging
  - Makes certification easier and brings trust
- Properties can be expressed as tests
  - Invariants can also be expressed as programs or tests
- Possibility to use models/programs for proving

SMART language specificities

- Functional (values manipulation) but with an imperative style
- Can be used at different abstraction levels
- Functions are partial
- Possible to associate proof obligations with logical paths in the execution graph
  - Proofs displayed as symbolic debugging
  - Makes certification easier and brings trust
- Properties can be expressed as tests
  - Invariants can also be expressed as programs or tests
- Possibility to use models/programs for proving

Predicates

```
public member(elt x, seq e) -> [true, false]
program {{ seq f, elt y }}
{
    f := e;
    while
    {
        ?removeFirst(y+, f, f+);
        if x = y; then return true;
    }
}
```

Implicit predicates

```
public equals(elt x+, elt y)
implicit program

public removeFirst(elt x+, seq e, seq f+) -> [true, empty]
implicit program

public equals(elt x, elt y) -> [true, false]
implicit program
```

Handling by case / control structure

```
public member(elt x, seq e) -> [true, false]
program {{ seq f, elt y }}
{
    f := e;
    while
    {
        ?removeFirst(y+, f, f+);
        if x = y; then return true;
    }
}
```

Handling by case / control structure

```
public member(elt x, seq e) -> [true,false]
program {{ seq f, elt y }}
[found:true]
{
    f :=e ;
    while
    {
        [empty :false]removeFirst(y+,f,f+) ;
        [true:found,false:true](x=y) ;
    }
}
```

Handling by case / control structure

```
public member(elt x, seq e) -> [true,false]
program {{ seq f, elt y }}
[found:true]
{
    f :=e ;
    while
    {
        [empty :false]removeFirst(y+,f,f+) ;
        [true:found,false:true](x=y) ;
    }
}
```

Data / Control separation

```
public member(elt x, seq e) -> [true, false]
program {{ seq f, elt y }}
{
    f := e;
    while
    {
        ?removeFirst(y+, f, f+);
        if x = y; then return true;
    }
}
```

Impossible cases / Associated local properties

// program

```
public lemma propx(elt x,seq e)
program
{
 empty(e) => !member(x,e);
}
```

// equivalent code:

```
public lemma propxa(elt x,seq e)
program
[OK:true]
{
     [false:OK]empty(e);
     [true:error,false:true]member(x,e);
}
```

Impossible cases / Associated local properties

// code chunks

// Property (1):

```
{
 empty(e+) => !?removeFirst(_, e, _);
}
```

// equivalent code:

```
{
    empty(e+);
    [true:error,empty:true]removeFirst(_x+,e,e+);
}
```

SMART language specificities

- Functional (values manipulation) but with an imperative style
- Can be used at different abstraction levels
- Functions are partial
- Possible to associate proof obligations with logical paths in the execution graph
  - Proofs displayed as symbolic debugging
  - Makes certification easier and brings trust
- Properties can be expressed as tests
  - Invariants can also be expressed as programs or tests
- Possibility to use models/programs for proving

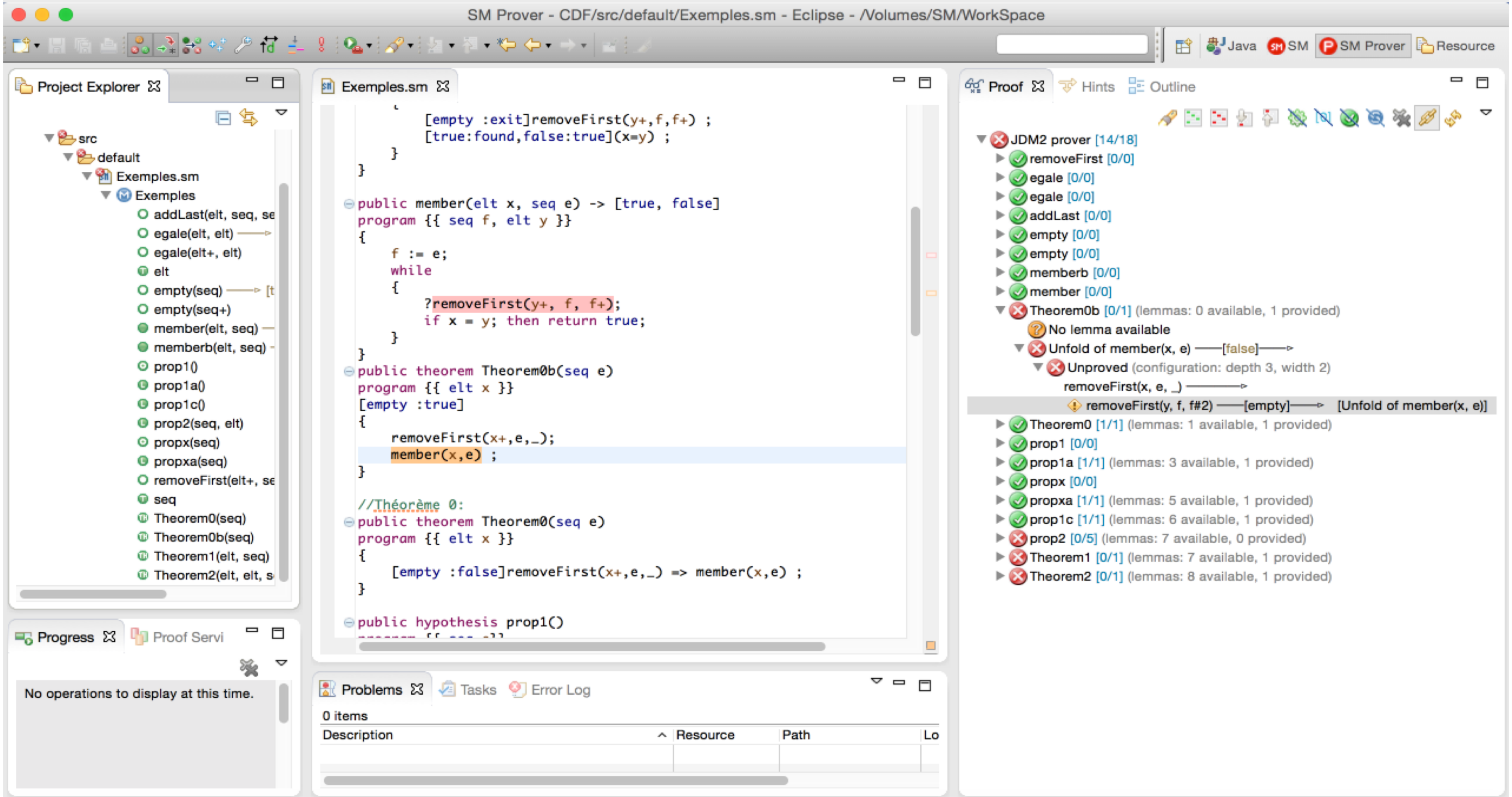
SMART language specificities

- Functional (values manipulation) but with an imperative style
- Can be used at different abstraction levels
- Functions are partial
- Possible to associate proof obligations with logical paths in the execution graph
  - Proofs displayed as symbolic debugging
  - Makes certification easier and brings trust
- Properties can be expressed as tests
  - Invariants can also be expressed as programs or tests
- Possibility to use models/programs for proving

SMART language specificities

- Functional (values manipulation) but with an imperative style
- Can be used at different abstraction levels
- Functions are partial
- Possible to associate proof obligations with logical paths in the execution graph
  - Proofs displayed as symbolic debugging
  - Makes certification easier and brings trust
- Properties can be expressed as tests
  - Invariants can also be expressed as programs or tests
- Possibility to use models/programs for proving

A more concrete example



A more concrete example

```
public theorem Theorem0b(seq e)
program {{ elt x }}
[empty :true]
{
    removeFirst(x+,e,_);
    member(x,e) ;
}
```

- memberb [0/0]
- Theorem0b [0/1] (lemmas: 0 available, 1 provided)
- Theorem0 [1/1] (lemmas: 1 available, 1 provided)
- prop1 [0/0]
- prop1a [1/1] (lemmas: 3 available, 1 provided)
- propx [0/0]
- propxa [1/1] (lemmas: 5 available, 1 provided)

A more concrete example

```
public theorem Theorem0b(seq e)
program {{ elt x }}
[empty :true]
{
    removeFirst(x+,e,_);
    member(x,e) ;
}
```

- memberb [0/0]
- Theorem0b [0/1] (lemmas: 0 available, 1 provided)
  - No lemma available
  - Unproved (configuration: depth 3, width 2)
    - removeFirst(x, e, _) ——▹
    - member(x, e) ——[false]——▹
- Theorem0 [1/1] (lemmas: 1 available, 1 provided)

A more concrete example

```
public member(elt x, seq e) -> [true, false]
program {{ seq f, elt y }}
{
    f := e;
    while
    {
        ?removeFirst(y+, f, f+);
        if x = y; then return true;
    }
}
public theorem Theorem0b(seq e)
program {{ elt x }}
[empty :true]
{
    removeFirst(x+,e,_);
    member(x,e) ;
}
```

```
▸ egale [0/0]
▸ addLast [0/0]
▸ empty [0/0]
▸ empty [0/0]
▸ memberb [0/0]
▸ member [0/0]
▾ Theorem0b [0/1] (lemmas: 0 available, 1 provided)
    No lemma available
  ▾ Unfold of member(x, e) ——[false]——▹
    ▾ Unproved (configuration: depth 3, width 2)
        removeFirst(x, e, _) ————▹
          removeFirst(y, f, f#2) ——[empty]——▹ [Unfold of member(x, e)]
▸ Theorem0 [1/1] (lemmas: 1 available, 1 provided)
▸ prop1 [0/0]
▸ prop1a [1/1] (lemmas: 3 available, 1 provided)
▸ propx [0/0]
```

SMART language specificities

- Functional (values manipulation) but with a imperative style
- Can be used at different abstraction levels
- Functions are partial
- Possible to associate proof obligations with logical paths in the execution graph
  - Proofs pictured and guided as symbolic debugging
  - Makes certification easier
- Properties can be expressed as tests
  - Invariants can also be expressed as programs or tests
- Possibility to use models/programs for proving

A more concrete example

// code chunks // Property (2)

```
{
    addLast(x,e,f+) ;
    g:=e;
    while
    {
        [empty :exit]removeFirst(y+,g,g+) ;
        [empty :error]removeFirst(z+,f,f+) ;
        [false:error](y=z) ;
    }
    [empty :error]removeFirst(z+,f,f+) ;
    [false:error](x=z) ;
    [true:error, empty:true]removeFirst(z+,f,f+) ;
}
```

A more concrete example

```
// Propriétés 2
public lemma prop2(seq e, elt x)
program {{ seq f, seq g, elt y, elt z }}
{
    addLast(x,e,f+) ;
    g:=e;
    while
    {
        [empty :exit]removeFirst(y+,g,g+) ;
        [empty :error]removeFirst(z+,f,f+) ;
        [false:error](y=z) ;
    }
    [empty :error]removeFirst(z+,f,f+) ;
    [false:error](x=z) ;
    [true:error, empty:true]removeFirst(z+,f,f+) ;
}
```

A more concrete example

```
public hypothesis prop1()
program {{ seq e}}
{
 empty(e+) => !?removeFirst(_, e, _);
}

public hypothesis prop2(seq e, elt x)
program {{ seq f, seq g, elt y, elt z }}
{
    addLast(x,e,f+) ;
    g:=e;
    while
    {
        [empty :exit]removeFirst(y+,g,g+) ;
        [empty :error]removeFirst(z+,f,f+) ;
        [false:error](y=z) ;
    }
    [empty :error]removeFirst(z+,f,f+) ;
    [false:error](x=z) ;
    [true:error, empty:true]removeFirst(z+,f,f+) ;
}
```

Example of properties to be proven

- Theorem0: removeFirst(x+,e,f+) => member(x,e) ;
- Theorem1: addLast(x,e,f+) => member(x,f) ;
- Theorem2: member(x,e) => addLast(y,e,f+) => member(x,f) ;

```
//Théorème 0:
public theorem Theorem0(seq e)
program {{ elt x }}
{
    [empty :false]removeFirst(x+,e,_) => member(x,e) ;
}


//Théorème 1:
public theorem Theorem1(elt x,seq e)
program {{ seq f }}
{
    addLast(x,e,f+) => member(x,f) ;
}

// Théorème 2:
public theorem Theorem2(elt x, elt y, seq e)
program {{  seq f }}
{
    member(x,e) => addLast(y,e,f+) => member(x,f) ;
}
```

Intermediate language: SMIL

```
//Théorème 0:
public theorem Theorem0(seq e)
program {{ elt x }}
{
    [empty :false]removeFirst(x+,e,_) => member(x,e) ;
}

public theorem Theorem0b(seq e)
program {{ elt x }}
[empty :true]
{
    removeFirst(x+,e,_);
    member(x,e) ;
}
```

Indexed variables. Logical traces. Proof structures. Congruence

```
// Theorem0 : Unfold
[empty:true]
{
        removeFirst(x+,e,_);
        {
                f :=e ;
                while
                {
                        [empty :error]removeFirst(y+,f,f+) ;
                        [true:empty,false :true](x=y) ;
                }
        }
}
```

Indexed variables. Logical traces. Proof structures. Congruence

```
// Theorem0 : Unfold
[empty:true]
{
        removeFirst(x+,e,_);
        {
                f :=e ;
                while
                {
                        [empty :error]removeFirst(y+,f,f+) ;
                        [true:empty,false :true](x=y) ;
                }
        }
}
// b* -> [b] b*
```

Indexed variables. Logical traces. Proof structures. Congruence

```
// Theorem0 : Unfold + Unrolled
[empty:true]
{
        removeFirst(x+,e,_);
        {
                f :=e ;
                [empty:error]removeFirst(y+,f,f+) ;
                [true:empty, false:true](x=y) ;
                while
                {
                        [empty :error]removeFirst(y+,f,f+) ;
                        [true:OK, false:true](x=y) ;
                }
        }
}
```

Indexed variables. Logical traces. Proof structures. Congruence

Logical trace:

- [true]removeFirst(x1+,e1,g1+) ;
- [true]equals(f1+,e1);
- [empty:error]removeFirst(y1+,f,f+) ;

Transitive closing of congruence:

- [true]removeFirst(x1+,e1,g1+) ;
- [true]equals(e1+,e1);
- [empty:error]removeFirst(x1+,e1,g1+) ;

Indexed variables. Logical traces. Proof structures. Congruence

Logical trace:

- [true]removeFirst(x1+,e1,g1+) ;
- [true]equals(f1+,e1);
- [true]removeFirst(y1+,f,f+) ;
- [false :true](x1=y1) ;

Transitive closing of congruence:

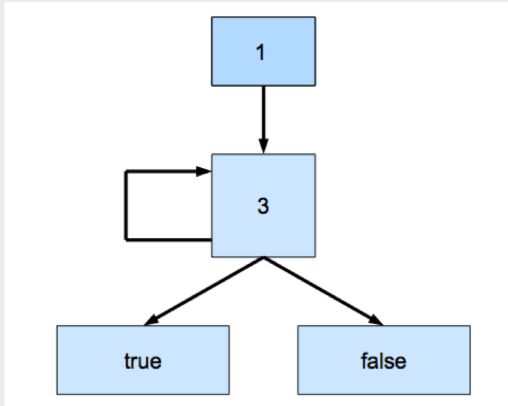
- [true]removeFirst(x1+,e1,g1+) ;
- [true]equals(e1+,e1);
- [true]removeFirst(x1+,e1,g1+) ;
- [false :error](x1=x1) ;

Indexed variables. Logical traces. Proof structures. Congruence

```
// Theorem0 : Unfold + Unrolled
[empty:true]
{
        removeFirst(x+,e,_);
        {
                f :=e ;
                [empty:error]removeFirst(y+,f,f+) ;
                [true:empty, false:error](x=y) ;
                while
                {
                        [empty :error]removeFirst(y+,f,f+) ;
                        [true:OK, false:true](x=y) ;
                }
        }
}
```

Composition with Synchronization

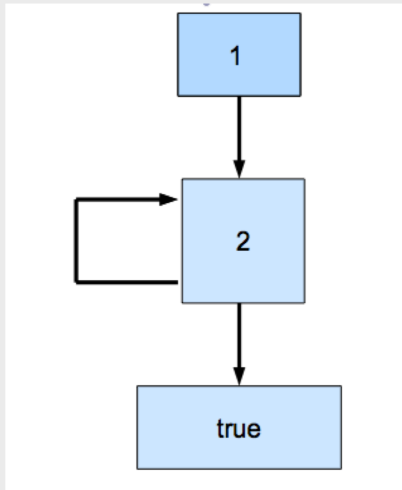
```
// Theorem1 Unfold
[OK:true]
{
1:          [empty :OK]addLast(x,e,f+);
            {
2:                    g:=f ;
                      while
                      {
3:                              [empty:false]removeFirst(y+,g,g+);
4:                              [true:OK, false:true](x=y);
            }
            }
}
```



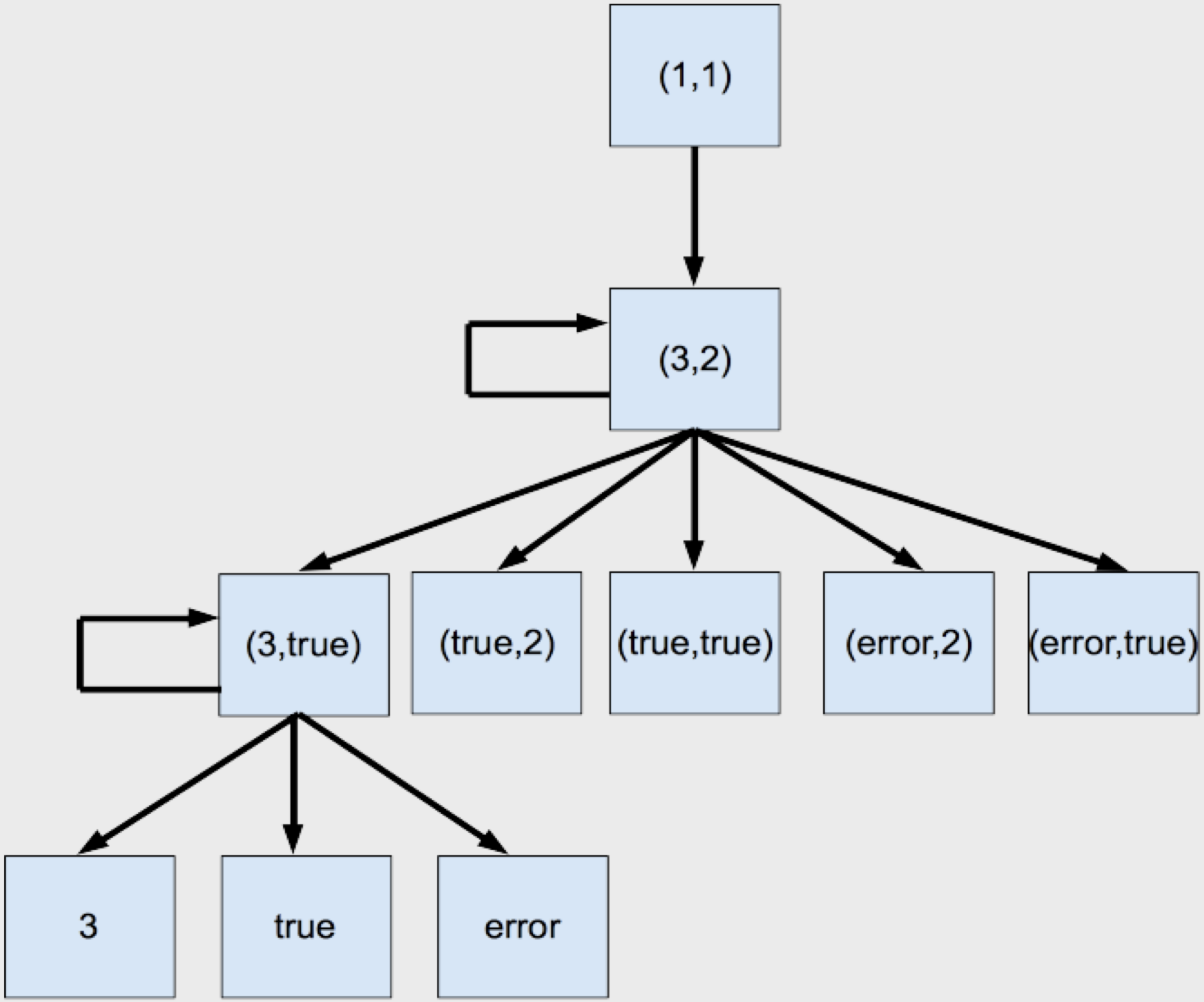
Rappel Theorem1: addLast(x,e,f+) ; => member(x,f) ;

Composition with Synchronization

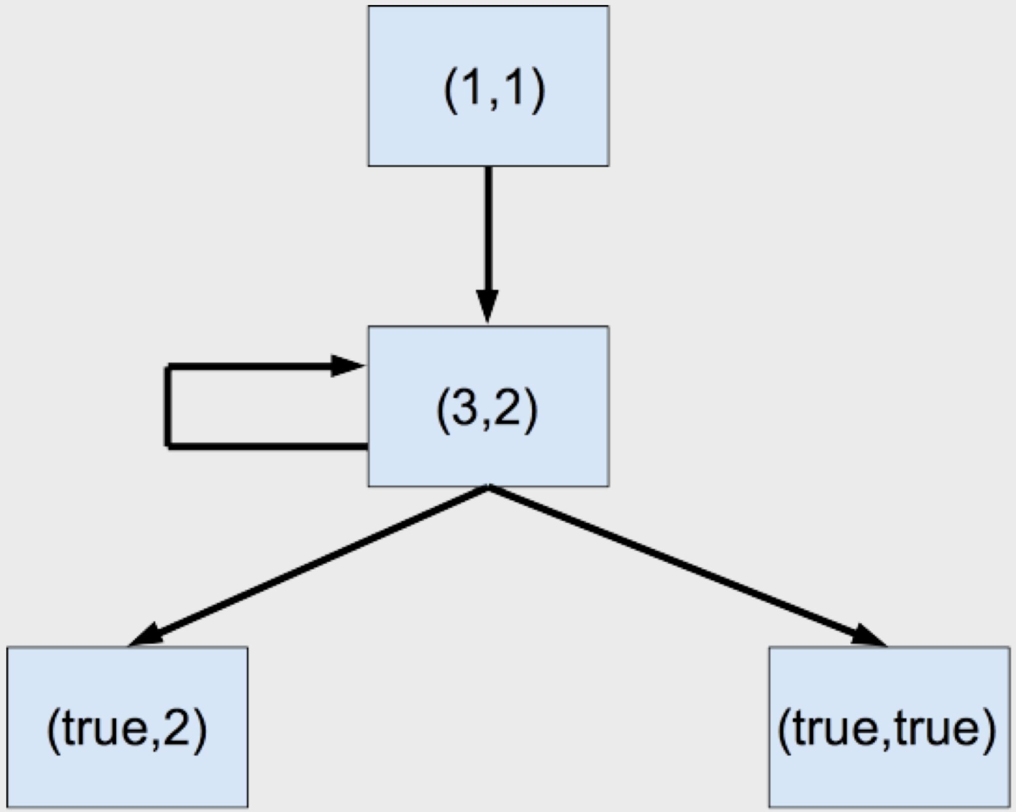
```
// code chunks  // Property (2)
{
1:      addLast(x,e,f+) ;
        while
        {
2:              [empty :exit]removeFirst(y+,e,e+) ;
3:              [empty :error]removeFirst(z+,f,f+) ;
4:              [false:error](y=z) ;
        }
5:      [empty :error]removeFirst(z+,f,f+) ;
6:      [false:error](x=z) ;
7:      [true:error,empty :true]removeFirst(z+,f,f+) ;
}
```



Composition Theorem1Unfold Axiom2

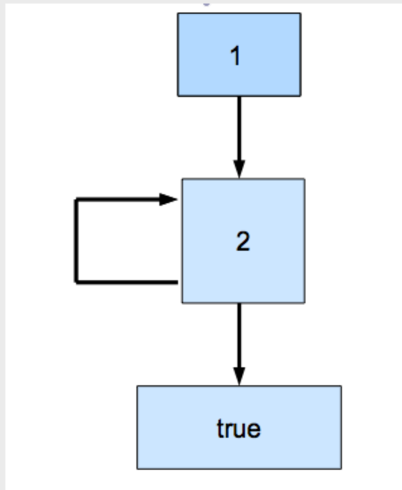


Simplification Composition Theorem1Unfold Axiom2 (propagation congruence)



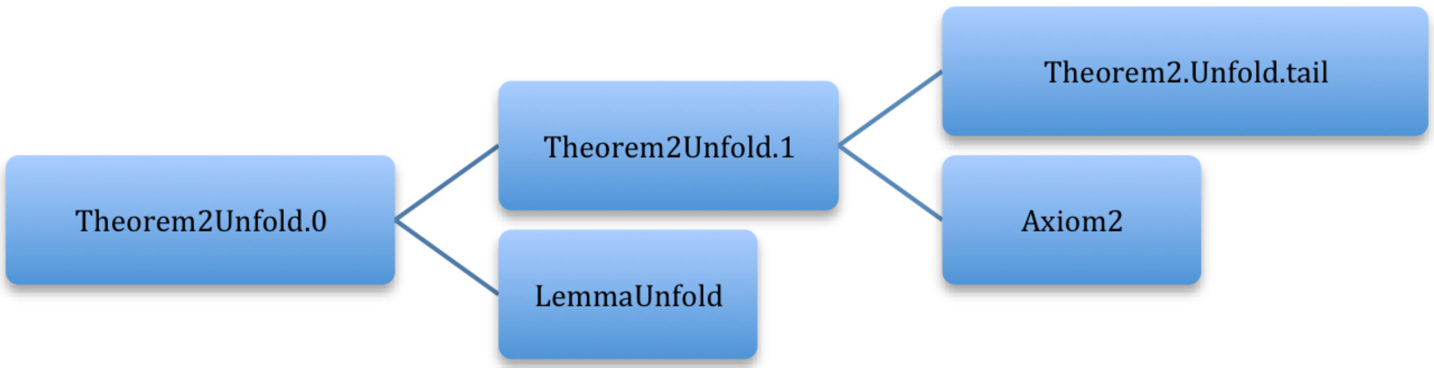
Composition with Synchronization

```
// code chunks  // Property (2)
{
1:      addLast(x,e,f+) ;
        while
        {
2:              [empty :exit]removeFirst(y+,e,e+) ;
3:              [empty :error]removeFirst(z+,f,f+) ;
                [false:error](y=z) ;
        }
4:      [empty :error]removeFirst(z+,f,f+) ;
5:      [false:error](x=z) ;
6:      [true:error,empty :true]removeFirst(z+,f,f+) ;
}
```



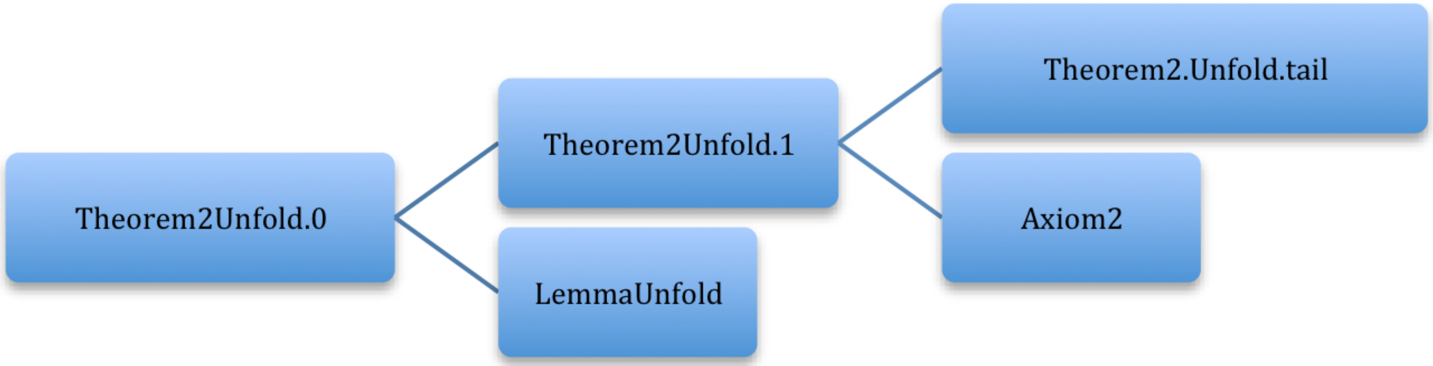
Theorem2: member(x,e); addLast(y,e,f+); => member(x,f) ;

New example of (double) composition for theorem 2 proof



LemmaUnfold : member(x,e) => member(x,e) ;

New example of (double) composition for theorem 2 proof



LemmaUnfold : [false:true]member(x,e) ;

Further Documentation

- “Inferring Frame Conditions with Static Correlation Analysis“, Oana Andreescu, Thomas Jensen, Stéphane Lescuyer, Benoît Montagu, POPL, 2019
- “Security Filters for IoT Domain Isolation“, Dominique Bolignano, Florence Plateau, ISoLa, 2018
- “Formally Proven and Certified Off-The-Shelf Software Components“, Dominique Bolignano, C&SAR, 2016
- “Proven Security for the Internet of Things“, Dominique Bolignano, Embedded Conference 2016
- “ProvenCore: Towards a Verified Isolation Micro-Kernel“, Stéphane Lescuyer, 10th HiPEAC Conference, 2015

Conclusions / Future Work

- Applicable to a very large range of market segments and situations
- Everything doesn’t need to be modelled nor proven (hypotheses, resistance to physical attacks, properties appropriateness, unsuitable architectures, human chain, etc.)
- More features to be added,
- Enlarging the scope of evaluation to hardware is planned,
- Some optimizations to be done if required,
- Other kernels to be handled,