Toward a methodology for Unified Verification of HW/SW Co-designs

Building a bridge between two worlds

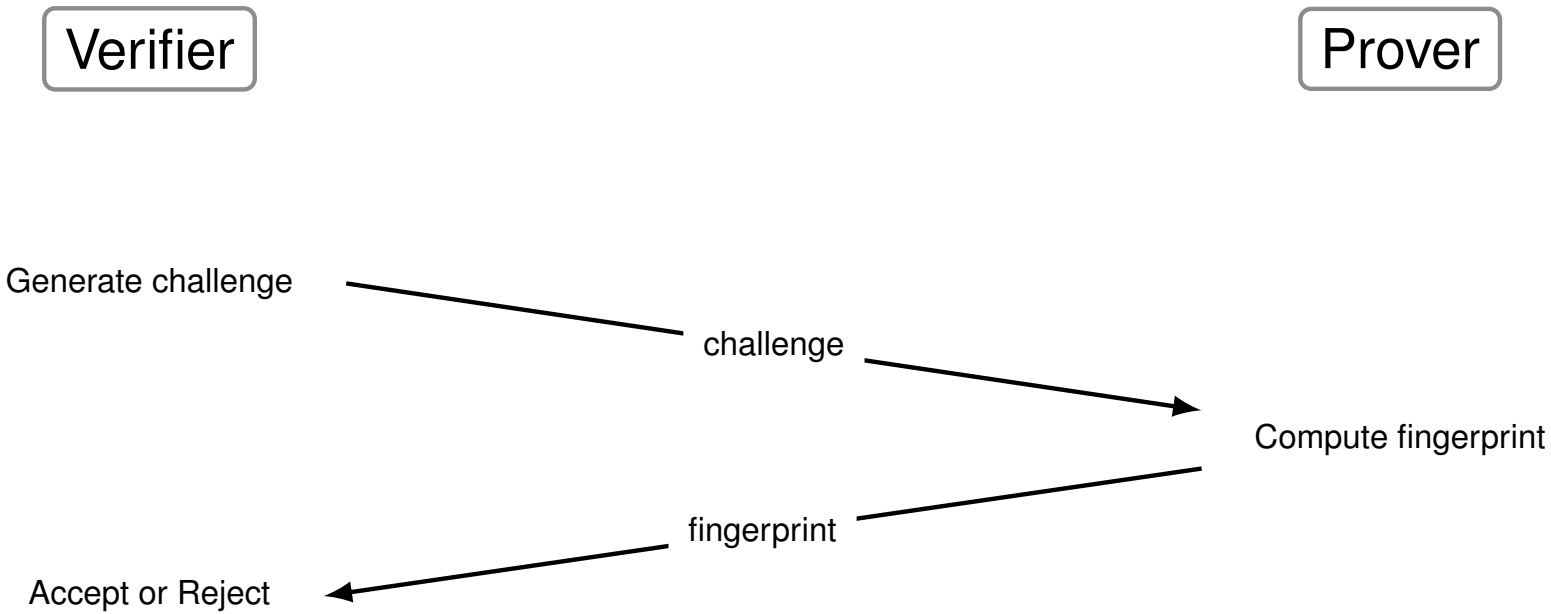
Florian Lugou <florian.lugou@telecom-paristech.fr>
Ludovic Apvrille <ludovic.apvrille@telecom-paristech.fr>
Aurélien Francillon <aurelien.francillon@eurecom.fr>

Contents

Contents

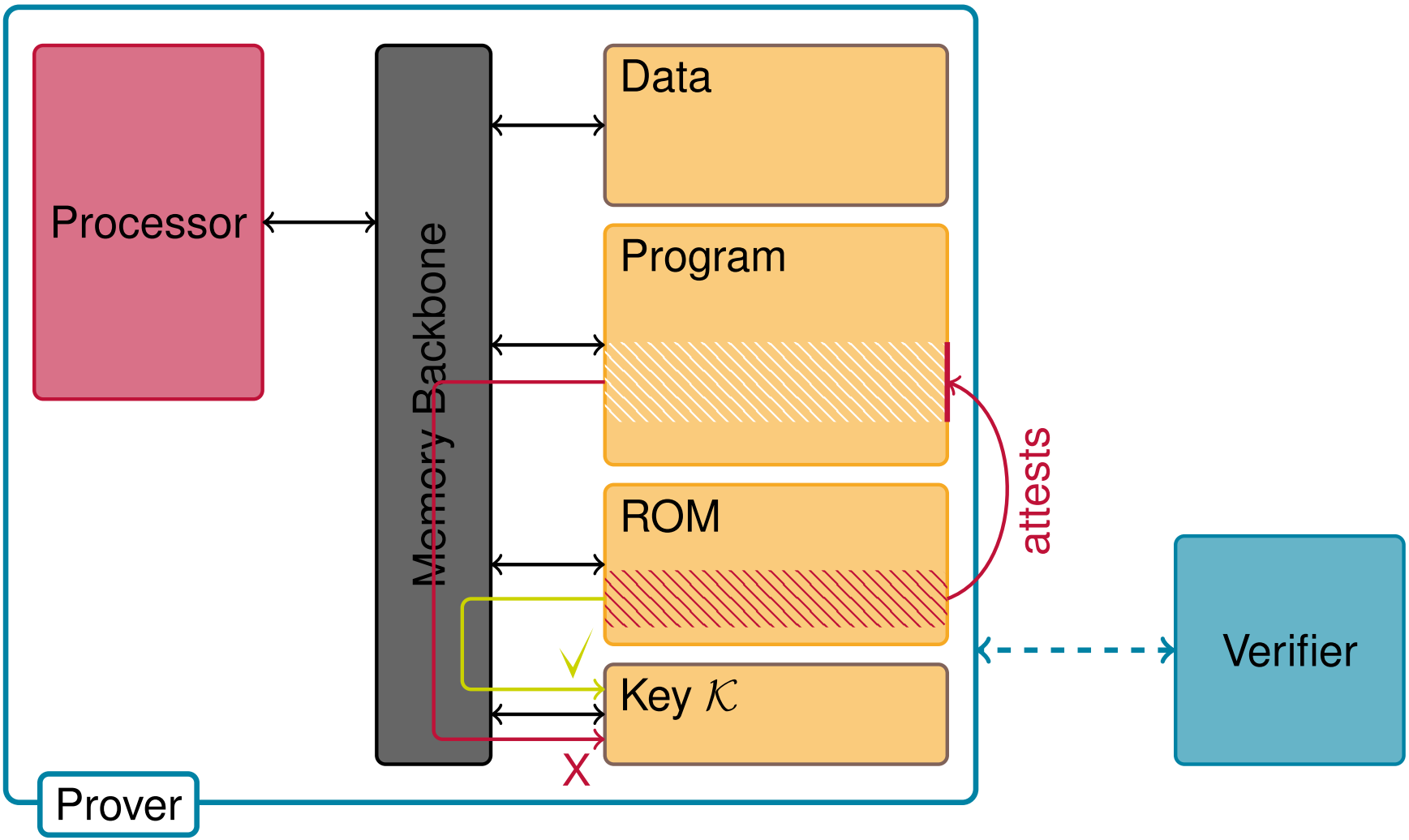
SMART
remote attestation

Secure and Minimal Architecture for (Establishing a Dynamic) Root of Trust



SMART

a HW/SW co-design



SMART

Bringing formal guarantees

We are here interested in **SW-level attacks** (no side channel attack, etc.).

Formal verification of SMART raises **challenges**:

- **Security of the scheme** depends on secrecy of $\mathcal{K}$.
- **Vulnerabilities in SW** (ROM) could endanger secrecy.
- **Custom HW** must be taken into account.
- Security depends on **HW features** such as interrupt masking.

Growing Interest in HW/SW Co-designs

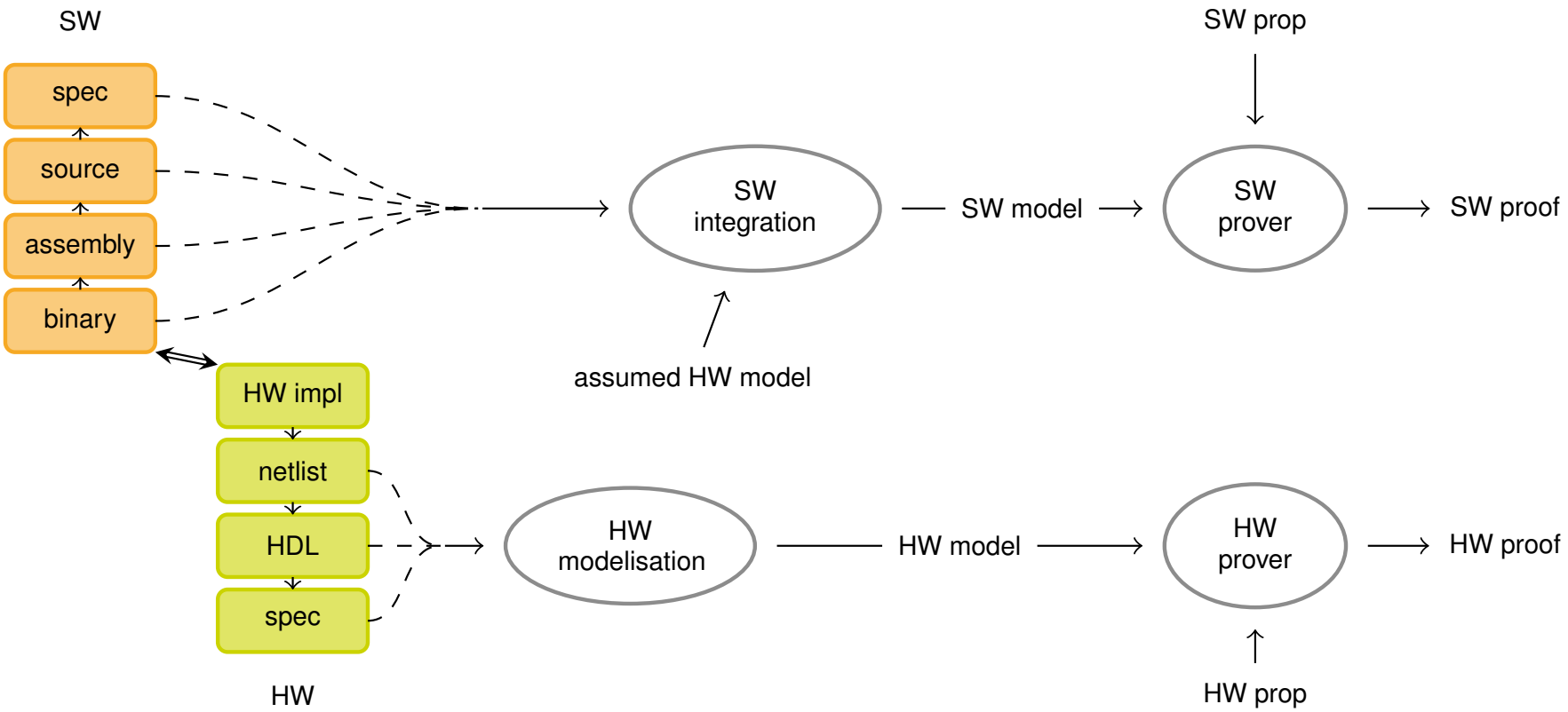
HW modification is costly **but**:

- **Mass production** makes HW customization affordable.
- **Some HW modifications** are cheaper than others.
- In some cases, **strong security** guarantees can't be achieved in pure SW.

It's because HW modification is costly that **formally verifying** it is essential.

Verifying both Hardware and Software

Different models and methods



Different methods of verification.

- SW: symbolic execution, taint propagation, model checking, . . .
- HW: model checking, equivalence checking, . . .

Verifying both Hardware and Software
But close interactions

However, HW and SW may have **close interactions**:

- SW and HW parts involved in a protocol;
- HW impacts the way SW is executed.

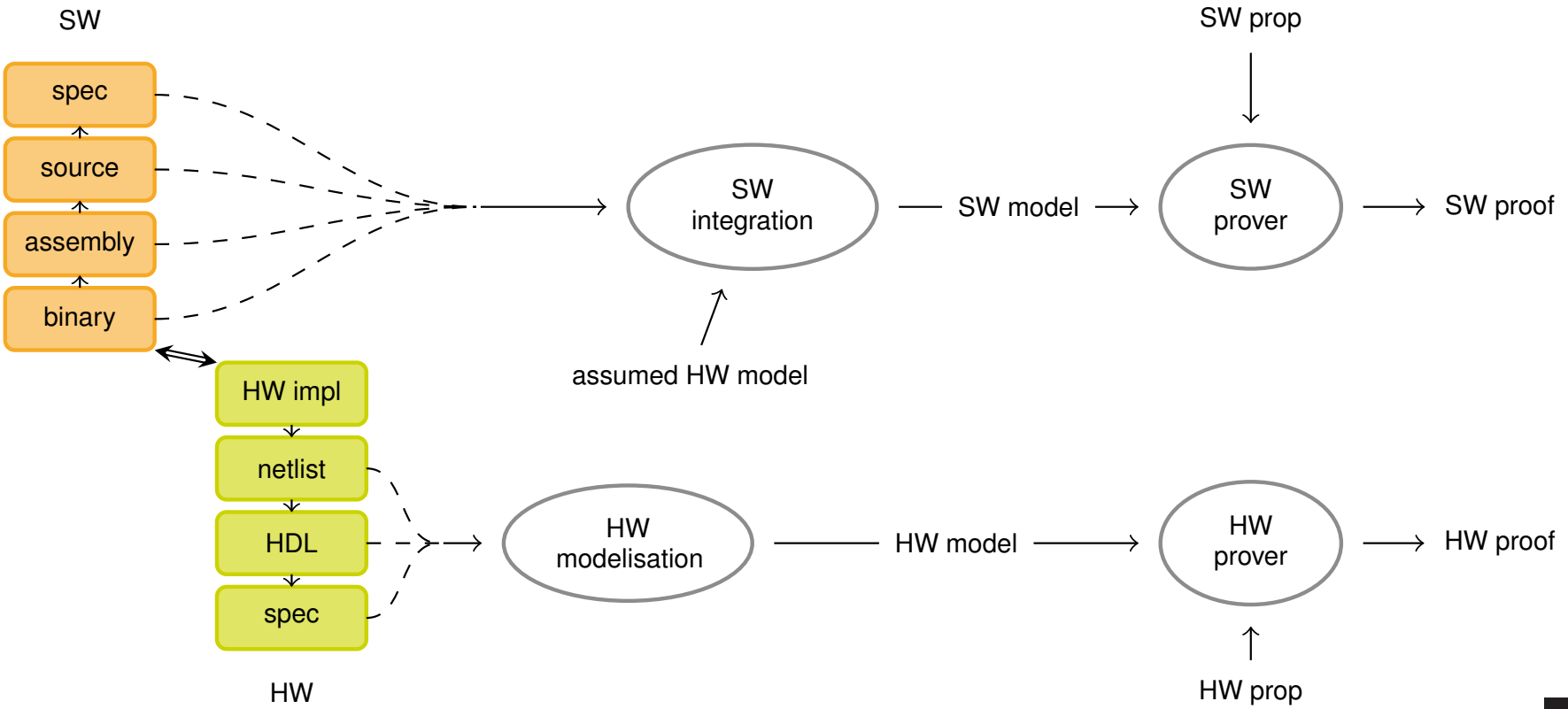
This is particularly true for **security designs**.

Contents

Successive verification of HW & SW

The idea

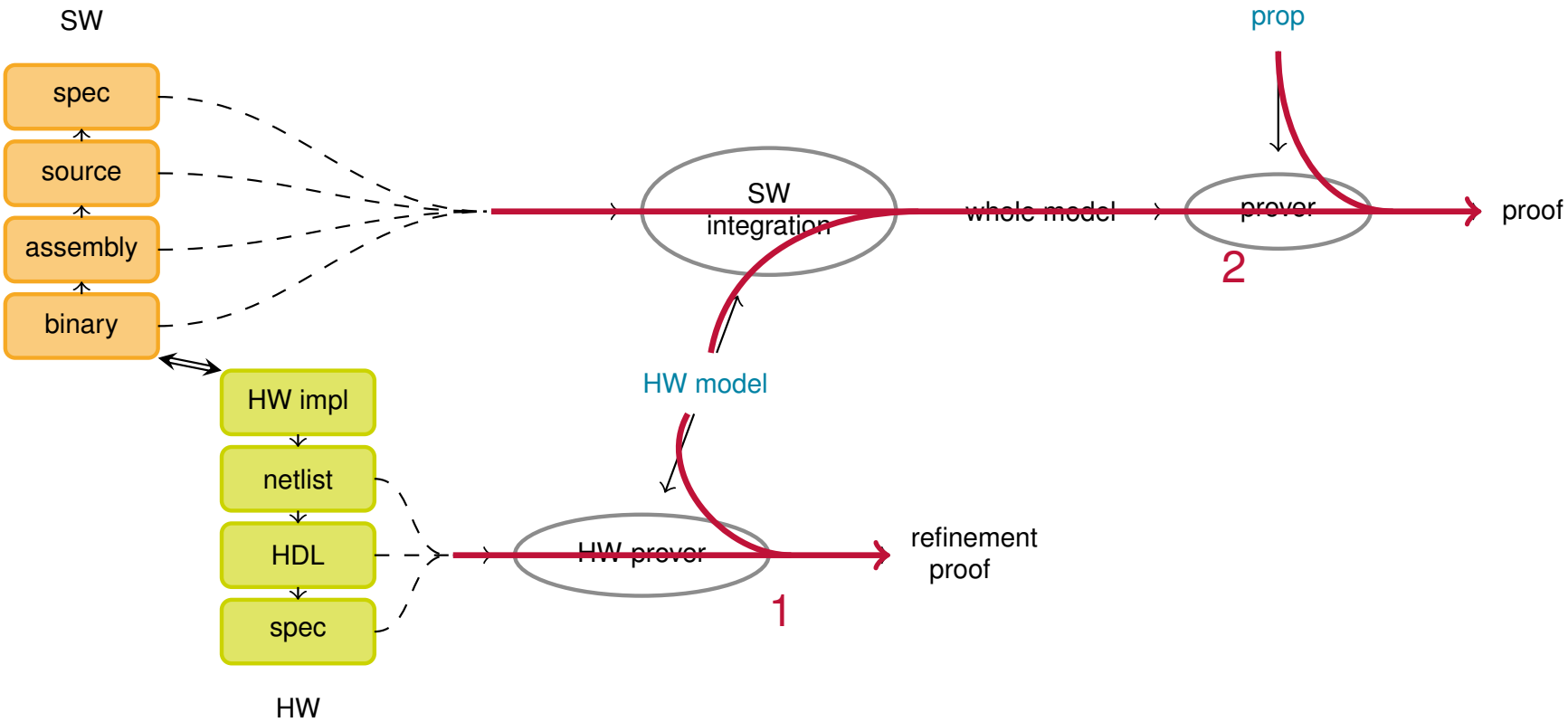
Independant Verification



Successive verification of HW & SW

The idea

Successive Verification



Successive verification of HW & SW
The idea

Manual expression of a **formal model** that:

- enables **HW** to be proved correct against this model,
- enables the verifier to express properties in this **formal environment**,
- and formalizes the effects of **SW instructions** on the model.

The **presence of the verifier is needed** to bridge the semantic gap between HW and SW

Successive verification of HW & SW

Feasibility and drawbacks

Is it feasible?

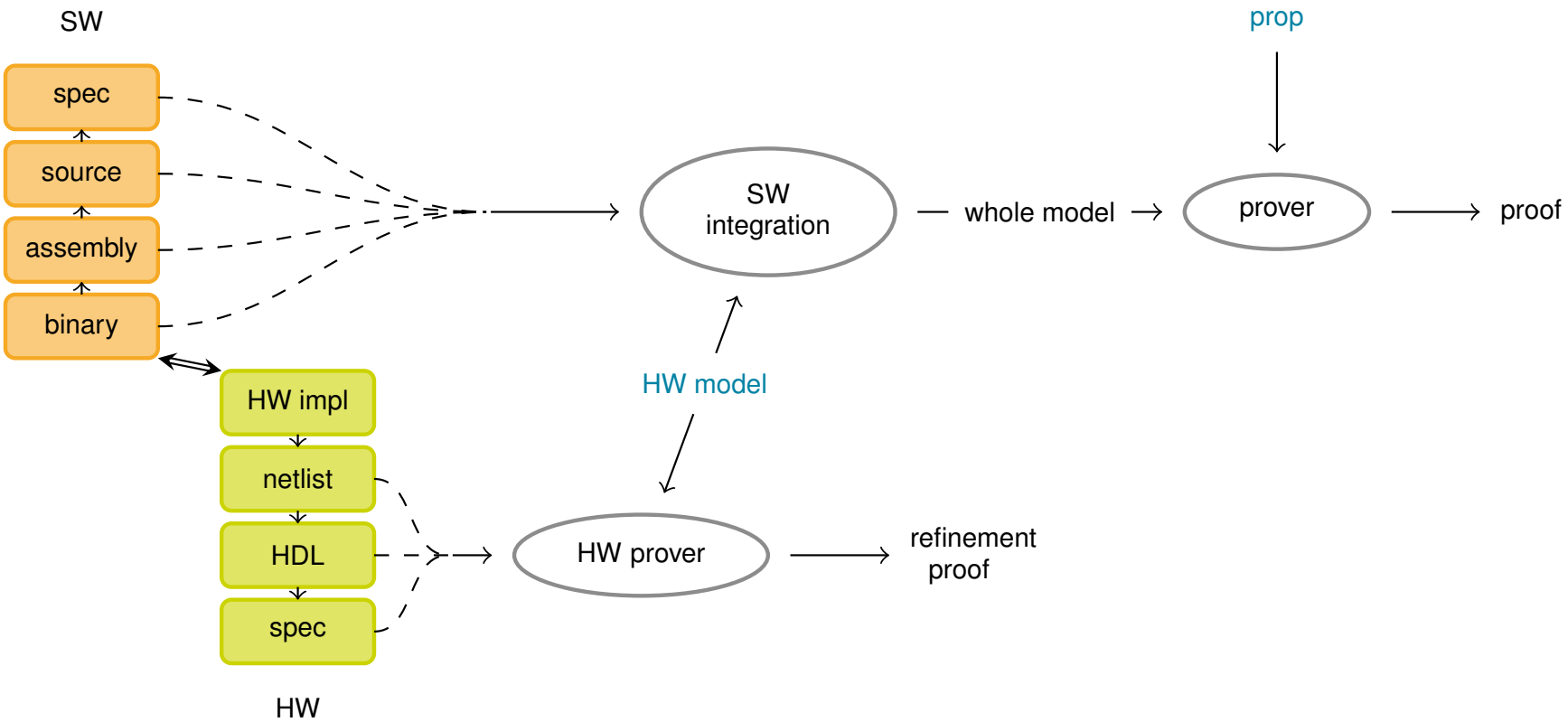
- Finding such model is tedious and involves a **lot of manual effort**.
- Feasible when **SW & HW are disjoint enough** to find a simple formal interface.

How could we **automate** this?

Unified verification

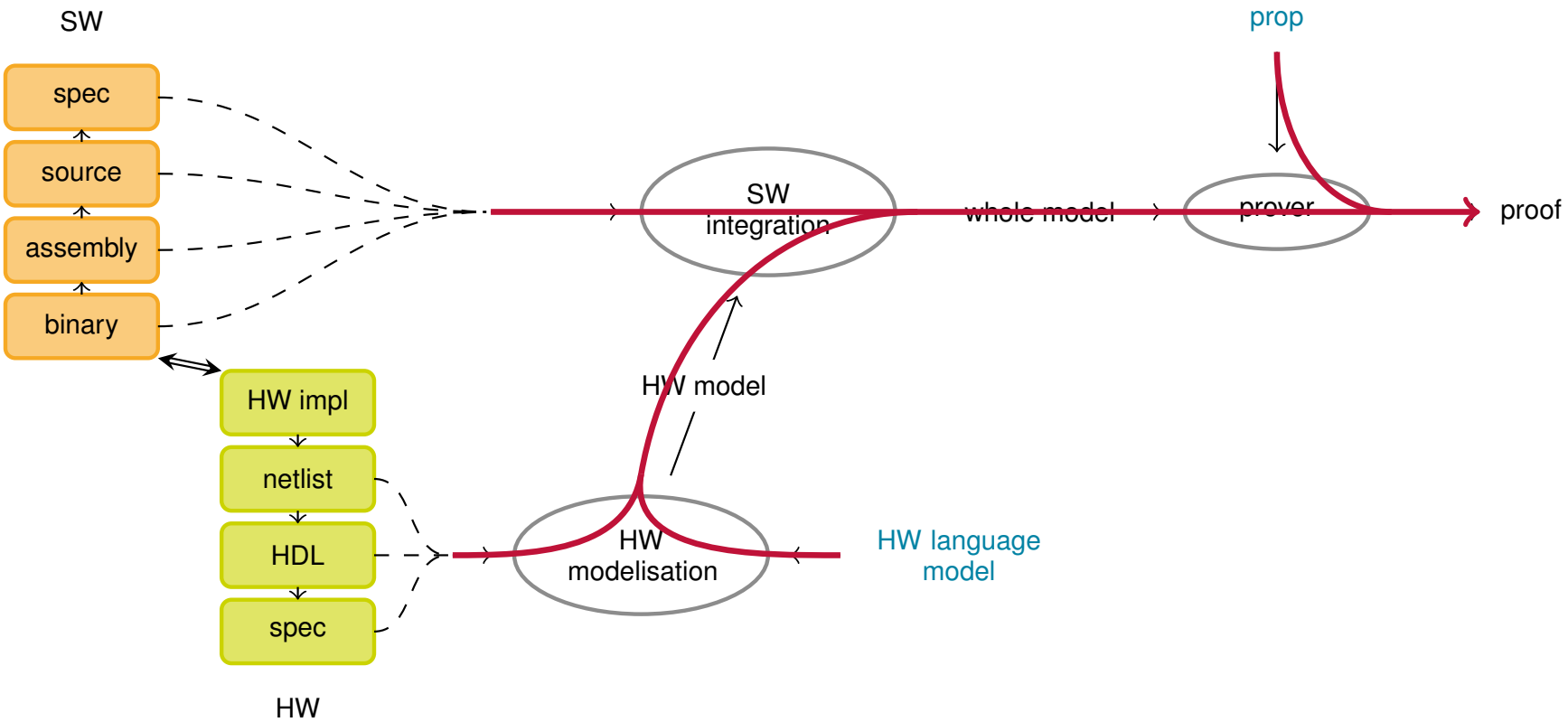
The idea

Successive Verification



Unified verification
The idea

Unified Verification



Unified verification
The idea

Use a **formal representation of the HDL**.

- Express the effect of **each HDL statement**,
- so that the composition of these is a **formal representation of the whole**.
- May **restrict the scope** of designs.
- Create an **interface** to integrate software.

Unified verification

ex: loosely coupled designs

E.g: HW and SW parts using a protocol to communicate [1]

- 2 agents communicate through a **clear interface**
- HW and SW describe the **behaviour of each agent**
- **doesn't really matter** whether it's HW or SW

Use a common language (as SystemC) and SW analysis tools

1. D. Kroening et al., *Formal Verification of SystemC by Automatic Hardware/Software Partitioning*

Unified verification
ex: tightly coupled designs

E.g.: Customizing core processor logic

- HW **customizes the way SW must be modelled**
- would require **low level representation of HW**
- **automated extraction of SW concepts** (program counter, stack frames, etc.) is nowaday mostly unfeasible
- SW representation that could be linked to a low level representation of HW: **binary format**

Find a compromise between **exhaustivity of HW description** and **scalability** of the proof?

Contents

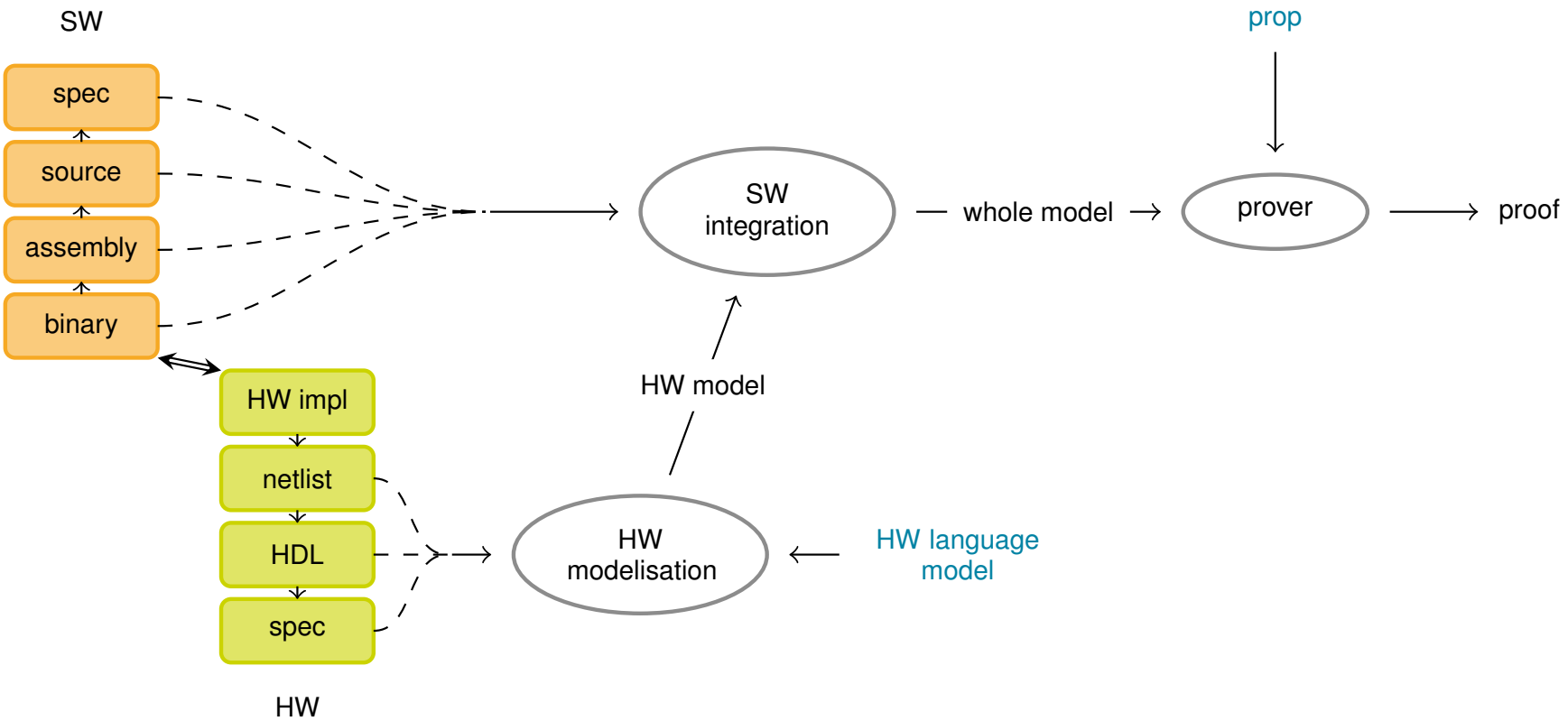
SMASHUP: What is it?

Simple Modelling and Attestation of Software and Hardware Using Proverif.

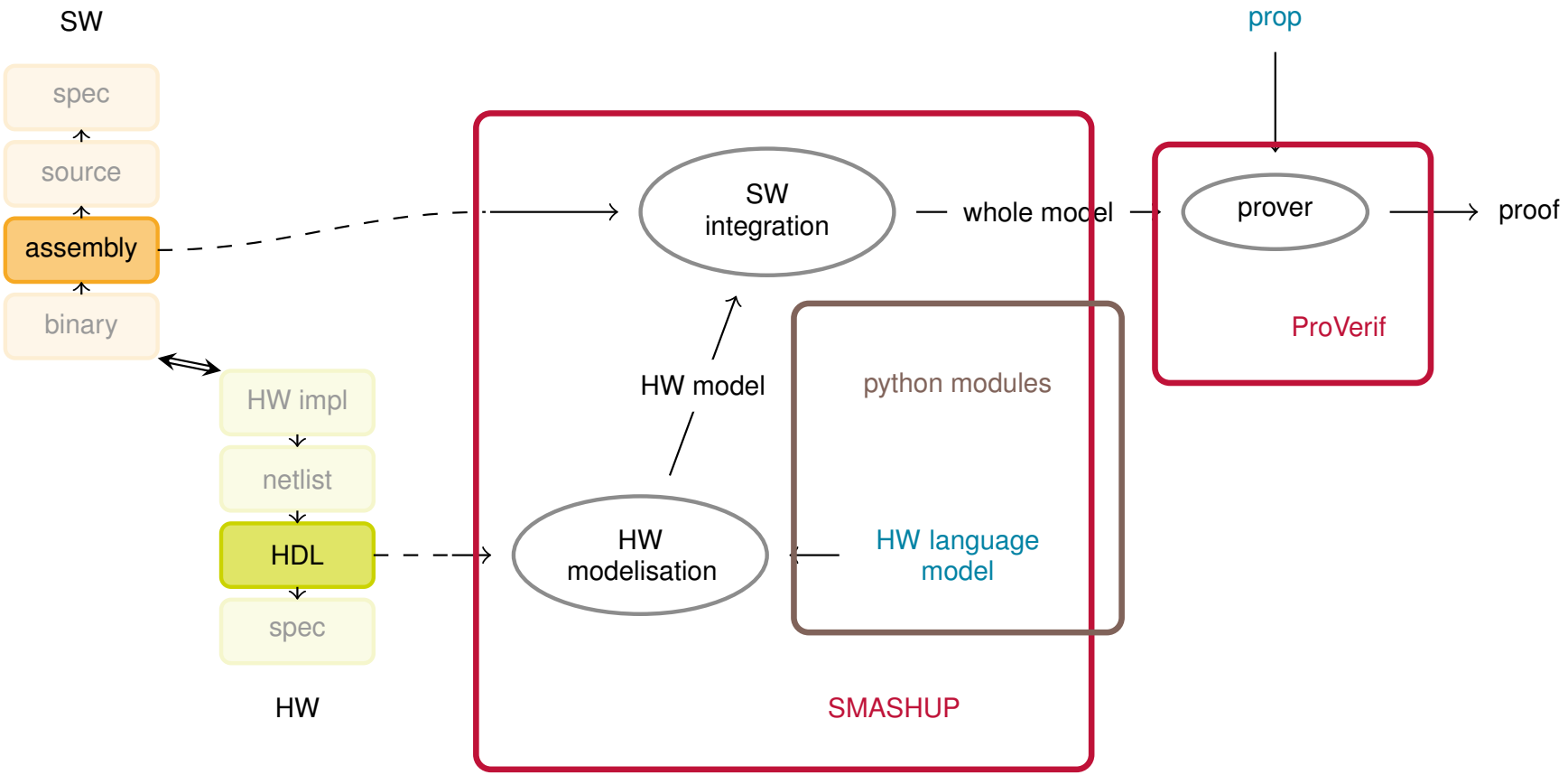
- A **python** compiler from HW + SW to ProVerif specification.
- SW is provided as **assembly language** (MSP430).
- HW is described as a **list of standard modules**.
- Properties are expressed as **secrecy** properties.

The specification produced can be checked with **ProVerif**.

SMASHUP: What is it?



SMASHUP: What is it?



Using ProVerif
Introduction

"ProVerif is a tool for automatically analyzing the security of cryptographic protocols."

- *automatically*: simple reasoning with Horn clauses
  - $\bigwedge_i p_i$ or $\bigwedge_i p_i \rightarrow q$
- *security*: naturally handles secrecy and authenticity properties
- *protocols*: multiple processes sending and receiving messages

Motivations: **simple logic** and **security orientation**

Using ProVerif

Reasoning with Horn clauses

Works on predicates. *E.g: attacker*(*var*) means the attacker knows value of var.

Horn clauses as logic bases. For instance:

$$\begin{array}{rrcl} & mess(ch, m) \wedge attacker(ch) & \rightarrow & attacker(m) \\ \text{and} & attacker(ch) \wedge attacker(m) & \rightarrow & mess(ch, m). \end{array}$$

Verification is based on unification of clauses:

$$\begin{array}{rrcl} & attacker(m) & \rightarrow & attacker(f(m)) \\ \text{and} & attacker(f(g(m))) & \rightarrow & attacker(m), \\ \text{results in} & attacker(g(m)) & \rightarrow & attacker(m). \end{array}$$

Using ProVerif

Application to verification of low-level SW

new predicate: *state*(pc, s) means *"a state where PC equals pc and system is in state s is reachable"*

effect of an instruction:

$$state(pc, s) \rightarrow state(pc', s')$$

Memory is modelled as an array of variables.

Example of HW modification (adding interrupts):

$$state(pc, s, 1) \rightarrow attacker(s)$$

$$\text{and} \qquad attacker(s') \wedge state(pc, s, 1) \rightarrow state(pc + 1, s', 1).$$

Limitations and discussion

Working with **concrete types**:

- **No representation** of numbers in ProVerif.
- **Simple arithmetic operations** increase complexity (ProVerif only allows constructors or reductions).
- *Idea*: interface ProVerif with **theory solvers** (bit vector, etc.).

Working at **binary level** (shellcodes, ROP, etc.).

Re-work the **HW Description Language** to enable finer-grained description of HW designs.

Contents

Conclusion

Summing it up:

- growing interest for **HW/SW Co-design**
- need for a method of **unified verification**
- a first step: **SMASHUP**

Thank you !

Questions?

Any Questions?