Specifying and Verifying Hardware-based Security Enforcement Mechanisms

Thomas LETAN

Agence Nationale de la Sécurité des Systèmes d'Information
CentraleSupélec, Inria Rennes, IRISA

Director Ludovic MÉ
Advisors Pierre CHIFFLIER
Guillaume HIET

October 25, 2018

Agenda

Context & Contributions

Specifying and Verifying HSE Mechanisms

Modular Verification of Complex Systems

Conclusion & Perspectives

Agenda

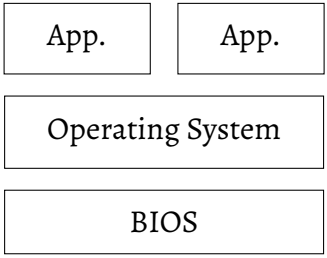
Context & Contributions

Specifying and Verifying HSE Mechanisms

Modular Verification of Complex Systems

Conclusion & Perspectives

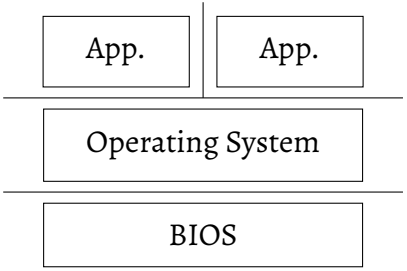
Context: Security of Low Level Software Components



Software components of

- Various origin
- Various quality
- Various level of trust

Context: Security of Low Level Software Components



Software components of

- Various origin
- Various quality
- Various level of trust

Each layer has to protect itself against upper (untrusted) layers

Hardware-based Security Enforcement Mechanisms

Trusted Software Components

configure

Hardware Components

to constrain

Untrusted Software Components

wrt.

Security Policies

Hardware-based Security Enforcement Mechanisms

Trusted Software Components

configure

Hardware Components

to constrain

Untrusted Software Components

wrt.

Security Policies

Operating System

sets up

Page Tables and Rings

to constrain

End-user Applications

to remain

Inside a Sandbox

The BIOS

The BIOS is provided by the hardware manufacturer

Boot sequence Initialize the hardware platform

Runtime Keep the platform in a working state

From a security perspective

- Shall continue to operate even in presence of a compromised software stack
- Most privileged software components of the stack

BIOS is the Root of Trust

BIOS HSE Mechanism

SMM	The most privileged x86 operating mode
SMRAM	Dedicated memory region of the DRAM protected by the Memory Controller
SMI	Hardware, non-maskable interrupt

BIOS HSE Mechanism

SMM	The most privileged x86 operating mode
SMRAM	Dedicated memory region of the DRAM protected by the Memory Controller
SMI	Hardware, non-maskable interrupt

$$\text{SMRAM} \Rightarrow \text{Integrity} \wedge \text{Confidentiality}$$

$$\text{SMI} \Rightarrow \text{Availability}$$

Selection of SMM Vulnerabilities

Prior to 2008	Incorrect configuration of `SMRAMC`
2009	SMRAM Cache Poisoning Attack
2014	Sinkhole
2015	Speed Racer, `SENTER` Sandman
Until today	Incorrect configuration of `BIOS_CNTL`

Selection of SMM Vulnerabilities

2009	SMRAM Cache Poisoning Attack
2014	Sinkhole
2015	Speed Racer, SENTER Sandman

Compositional Attacks

- Only legitimate (wrt. specifications) use of hardware features
- Flaws in hardware specifications

Selection of SMM Vulnerabilities

Prior to 2008 Incorrect configuration of `SMRAMC`

Until today Incorrect configuration of `BIOS_CNTL`

Misconfiguration Vulnerabilities

- Hardware features are available
- But are not correctly used by manufacturers

Challenges of HSE Mechanisms

A HSE mechanism implementation is correct if

- Hardware components expose sound APIs
- Trusted software components correctly use them

In between several verification domains

- Hardware verification
- Machine code verification
- System software verification

State of the Art: Hardware Verification

Verifying intrinsic properties of hardware components, *e.g.* the processor

x86 Execution Engine: Kaivola et al. 2009
SGX: Leslie-Hurd, Caspi, and Fernandez 2015

ARM Formal Specification: Reid 2016

XOM Tamper-resistant hardware: Lie et al. 2003

- The rest of the architecture is less regarded
- Hardware/software co-designs are less regarded

State of the Art: Machine Code Verification

Providing an abstract machine to reason about machine code execution

x86 Morrisett et al. 2012
Goel et al. 2014

- Abstract most of the concrete machine
- Reason only about one software component

State of the Art: System Software Verification

Verifying the correctness of a system software component

Hypervisor	VirtCert: Barthe et al. 2014
Kernel	seL4:Klein et al. 2009
	CertiKOS: Gu et al. 2016
	Pip Jomaa et al. 2016

- Model only the relevant hardware features
- Tied to one particular software implementation

Contributions

A theory of HSE mechanisms to formally specify and verify them

- *SpecCert: Specifying and Verify Hardware-based Security Enforcement Mechanisms*, Formal Methods 2016, Letan, Chifflier, et al. 2016
- Proof of correctness of the HSE mechanism implemented by x86 BIOSes at runtime in Coq

A Compositional Verification Framework

- *Modular Verification of Program with Effects and Effect Handlers in Coq*, Formal Methods 2018, Letan, Régis-Gianas, et al. 2018
- FreeSpec, a general-purpose framework for the Coq theorem prover

Agenda

Context & Contributions

Specifying and Verifying HSE Mechanisms

Modular Verification of Complex Systems

Conclusion & Perspectives

Goal: Formal Specifications

Formal specification of HSE mechanisms to address

1. Compositional attacks
2. Hardware misconfiguration

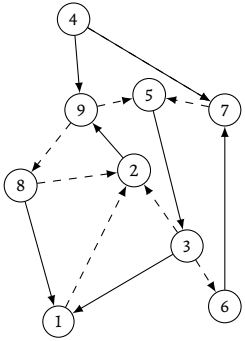
Software Developers Perspective

- Unambiguous list of requirements
- Focus on security

Hardware Designers Perspective

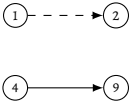
- Foundation of a verification process

Prerequisite: Hardware Model (1/2)



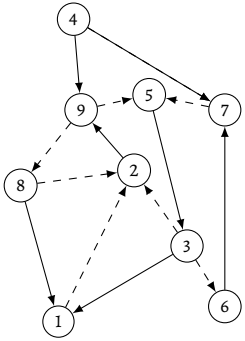
Hardware model as a Labeled Transition System

- Set of states
  Registers values, RAM content
- Set of labeled transitions
  - Software transitions
    ISA semantics
  - Hardware transitions
    Hardware interrupts



$$\Sigma \triangleq \langle H, L_H, L_S, T \rangle$$

Prerequisite: Hardware Model (2/2)




```
mov (%ecx), %eax
```

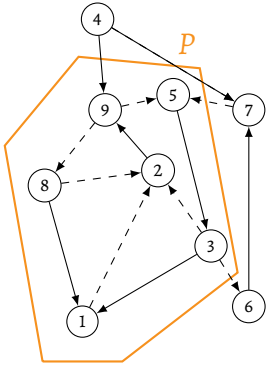
Scenario 1

1. Read the content of %ecx
2. Read the DRAM
3. Update the value of %eax

Scenario 2

1. Read the content of %ecx
2. Raise a Page Fault

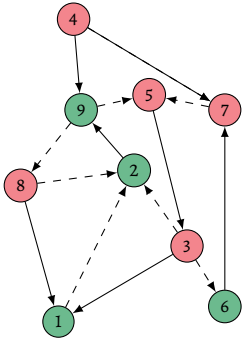
Step 1. Security Policy



Goal: Enforcing a security policy

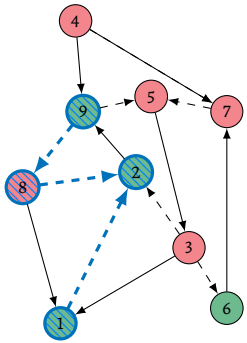
- Safety and liveness properties (Schneider 2000)
- $P \subseteq \mathcal{R}(\Sigma)$, where $\mathcal{R}(\Sigma)$ the set of traces of Σ

Step 2. HSE Mechanism



- S the set of software components
  $S = \{\texttt{bios}, \texttt{os}, \texttt{app}_1, \texttt{app}_2\}$
- $T \subseteq S$ the subset of trusted software components **which implements the HSE mechanism**
  $T = \{\texttt{bios}\}$ $\quad$ $U = S \backslash T$
- $context : H \rightarrow S$ to determine which software component in executed in a given state
  $context(s) = \texttt{bios}$ *iff the core is in SMM*

Step 2. HSE Mechanism



List of requirements

- over states: safe hardware configurations

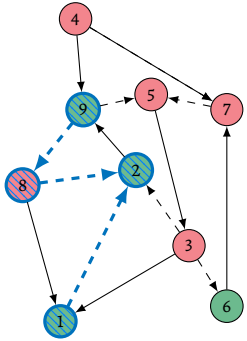
  The SMRAM has to be locked

- over software transitions: safe software executions

  Do not execute code outside of the SMRAM

$$\Delta \triangleq \langle S, T, context, state\_req, trans\_req \rangle$$

HSE Laws



Attacker Model

We do not make hypotheses about the behavior of untrusted softwre components

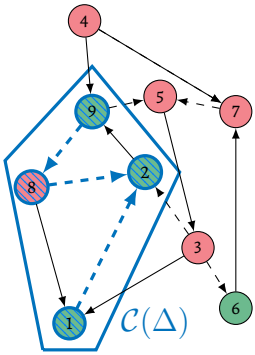
$$\forall (h, l) \in H \times L_S,$$
$$context(h) \notin T \Rightarrow trans\_req(h, l)$$

Requirements consistency

Requirements over transitions preserves requirements over states

$$\forall (h, l, h') \in \mathcal{T}(\Sigma),$$
$$state\_req(h)$$
$$\wedge\ (l \in L_S \Rightarrow trans\_req(h, l))$$
$$\Rightarrow state\_req(h')$$

Compliant Traces



A trace $\rho \in \mathcal{R}(\Sigma)$ complies with Δ when

- Its initial state satisfies the requirements over states

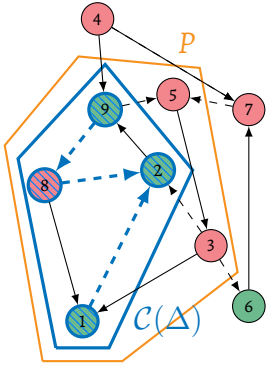
$$state\_req(init(\rho))$$

- Trusted software components satisfy the requirements over software transitions

$$\forall (h, l, h') \in trans(\rho),$$
$$l \in L_S \Rightarrow trans\_req(h, l)$$

$\mathcal{C}(\Delta)$ is the set of compliant traces

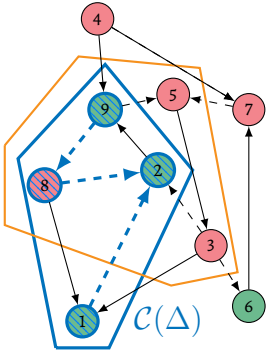
Step 3. Correctness



Implementing Δ is a sufficient condition in order to enforce P

$$\forall \rho \in \mathcal{C}(\Delta), \rho \in P$$

Step 3. Correctness



Implementing Δ is a sufficient condition in order to enforce P

$$\forall \rho \in \mathcal{C}(\Delta), \rho \in P$$

Typically, compositional attacks will prevent to conclude the correctness proof.

SpecCert

Scope

MINX86 A minimal x86 model

$\Delta_{\texttt{bios}}$ A formalization of the HSE implemented by the BIOS

Results

- $\Delta_{\texttt{bios}}$ satisfies the attacker model (law 1)
- $\Delta_{\texttt{bios}}$ has consistent requirements (law 2)
- $\Delta_{\texttt{bios}}$ is correct wrt. the code injection policy

BIOS HSE Mechanism Overview

Requirements over states

- PC contains a SMRAM address if in SMM
- SMBASE contains a SMRAM address
- SMRAM contains code owned by the BIOS
- SMRAMC register has been locked
- SMRR registers are correctly configured
- Cached SMRAM is owned by the BIOS

Requirements over transitions

When CPU is in SMM (BIOS)

- Do not modify the SMRR
- Do not jump outside of the SMRAM

Code Injection Policy

When a CPU in SMM fetches an instruction, this instruction is owned by the BIOS

SpecCert Implementation

- 2 000 lines of definitions, 2 500 lines of proofs
- 150 lemmas and theorems
- Available as a free software (CeCILL-B)

`https://github.com/lthms/SpecCert`

Lessons Learned

Our theory allows for

- Reasoning about hardware/software co-designs
- Without modeling software components

A hardware model remains mandatory

- Practicable
- Reusable

Agenda

Context & Contributions

Specifying and Verifying HSE Mechanisms

Modular Verification of Complex Systems

Conclusion & Perspectives

Goal: Compositional Reasoning

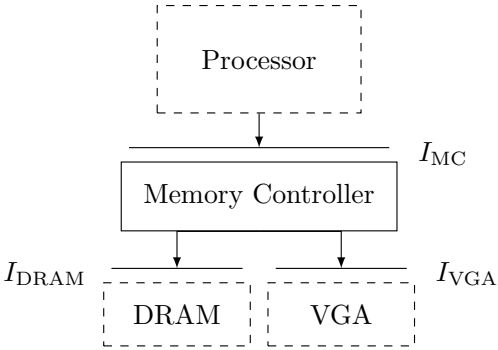
Ease the reasoning about component-based systems

- Assume-guarantee (Pnueli 1985)

Our contributions is twofold

- Modular verification approach for component-based systems
  - In isolation *and* in composition
- An implementation of this approach in Coq
  - Algebraic effects and effect handlers (Bauer and Pretnar 2015)

Modeling Interactions with Interfaces



- Set of operations expected to produce a result
- Each component
  - Exposes one interface
  - Uses many interfaces

Abstract Specifications

Given an Interface, we define two classes of requirements

Precondition Which operations can be used at a given time?

Postcondition What guarantee to expect from their result?

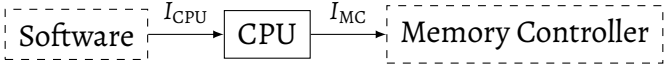
$$\text{Precondition} \Rightarrow \text{Postcondition}$$

Illustration of the Proposed Formalism

CPU

Illustration of the Proposed Formalism

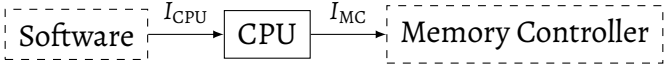
The CPU executes a **Software Component** thanks to a **Memory Controller**.



$$\frac{}{I_{\text{CPU}} \xrightarrow{CPU} I_{\text{MC}}}$$

Illustration of the Proposed Formalism

We want to prove the CPU complies with a couple of pre and postconditions $(\mathbb{P}, \mathbb{Q})$.

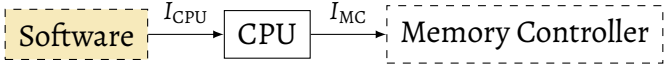


$$\frac{I_{\text{CPU}} \xrightarrow{CPU} I_{\text{MC}}}{\mathbb{P}}$$

$$\mathbb{Q}$$

Illustration of the Proposed Formalism

We assume the Software Component will enforce the precondition $\mathbb{P}$.

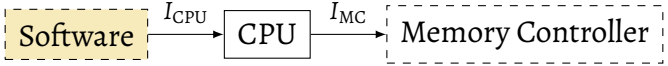


$$\frac{I_{\text{CPU}} \xrightarrow{CPU} I_{\text{MC}}}{\mathbb{P}}$$

$$\mathbb{Q}$$

Illustration of the Proposed Formalism

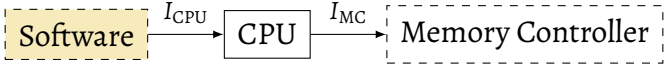
We want to verify that, according to the CPU model, the postcondition $\mathbb{Q}$ holds.



$$\frac{I_{\text{CPU}} \xrightarrow{CPU} I_{\text{MC}}}{\begin{array}{l}\mathbb{P} \\ \Downarrow \\ \mathbb{Q}\end{array}}$$

Illustration of the Proposed Formalism

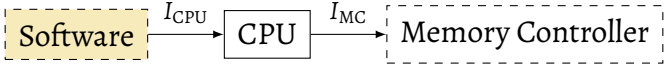
Rather than relying on the Memory Controller model, we'd rather identify a sufficient couple $(\mathbb{P}', \mathbb{Q}')$, and abstract away the MCH.



$$\begin{array}{ccc} I_{\text{CPU}} & \xrightarrow{CPU} & I_{\text{MC}} \\ \hline \mathbb{P} & & \mathbb{P}' \\ \\ \mathbb{Q} & & \mathbb{Q}' \end{array}$$

Illustration of the Proposed Formalism

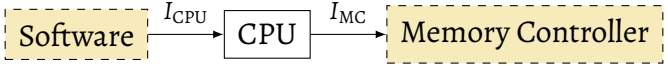
We prove that, because $\mathbb{P}$ is assumed and thanks to the CPU model, then the assumptions $\mathbb{P}'$ are met.



$$\begin{array}{ccc} I_{\text{CPU}} & \xrightarrow{CPU} & I_{\text{MC}} \\ \hline \mathbb{P} & \Longrightarrow & \mathbb{P}' \\ \\ \mathbb{Q} & & \mathbb{Q}' \end{array}$$

Illustration of the Proposed Formalism

We assume the Memory Controller enforces the post-condition $\mathbb{Q}'$.



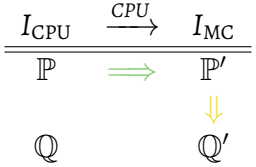
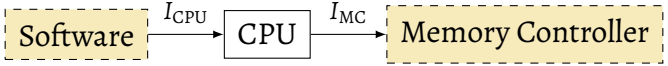


Illustration of the Proposed Formalism

We prove that, because $\mathbb{Q}'$ and thanks to CPU model, then $\mathbb{Q}'$ is verified.



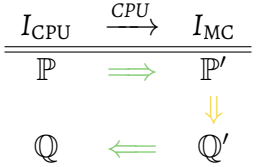
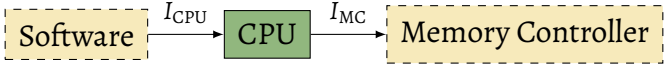


Illustration of the Proposed Formalism

In other words, our CPU model enforces $\mathbb{Q}$ as long as the Software component complies to $\mathbb{P}$.



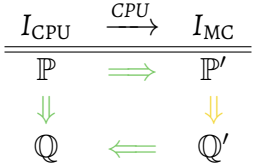


Illustration of the Proposed Formalism

We can then have a similar work with the Memory Controller model.

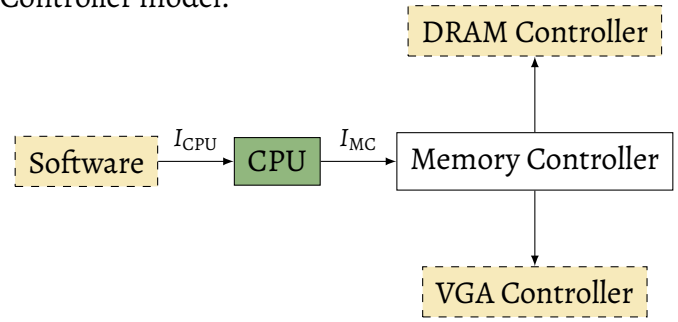


Illustration of the Proposed Formalism

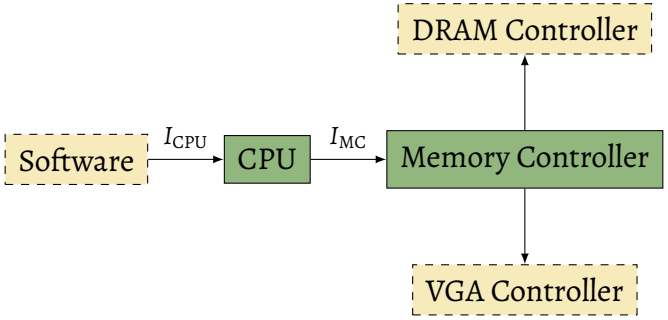


Illustration of the Proposed Formalism

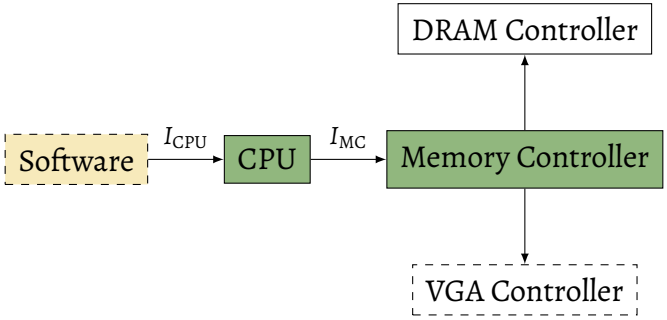


Illustration of the Proposed Formalism

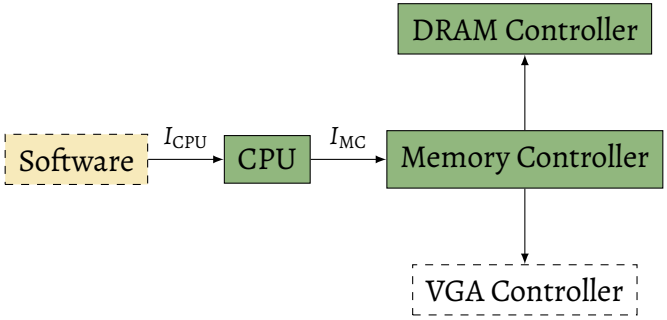
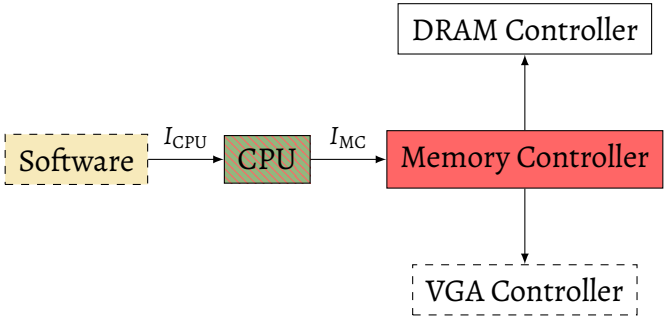


Illustration of the Proposed Formalism



Modeling Interfaces in Coq

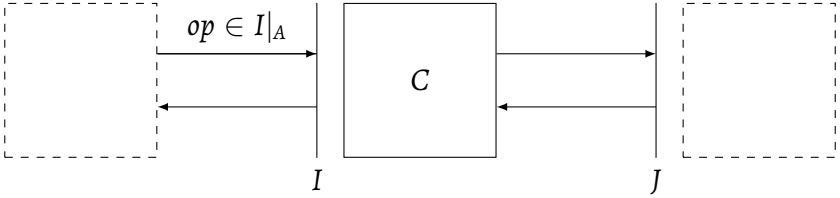
- I is a set of computational requests to be sent to a component
- $I|_A \subseteq I$ is the subset of requests whose results belongs to A

$$
\begin{array}{rcl}
I_{\mathtt{MC}} & \triangleq & \mathit{Read} : \{\mathtt{in\_smm}, \mathtt{unprivileged}\} \times \mathtt{PhysAddr} \to I_{\mathtt{MC}}|_{\mathtt{Word}} \\
& | & \mathit{Write} : \{\mathtt{in\_smm}, \mathtt{unprivileged}\} \times \mathtt{PhysAddr} \times \mathtt{Word} \to I_{\mathtt{MC}}|_{\{\emptyset\}}
\end{array}
$$

$$\mathit{Read}(\mathtt{in\_smm}, \mathtt{0x151}) \in I_{\mathtt{MC}}|_{\mathtt{Word}}$$

$$\mathit{Write}(\mathtt{in\_smm}, \mathtt{0x1ed}, \mathtt{0xcafe}) \in I_{\mathtt{MC}}|_{\{\emptyset\}}$$

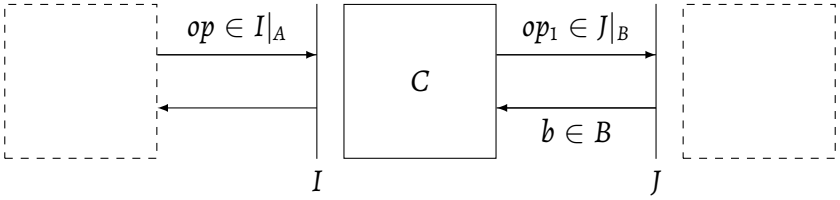
Use Case: Presentation



The component C

1. Receives a computational request op through I

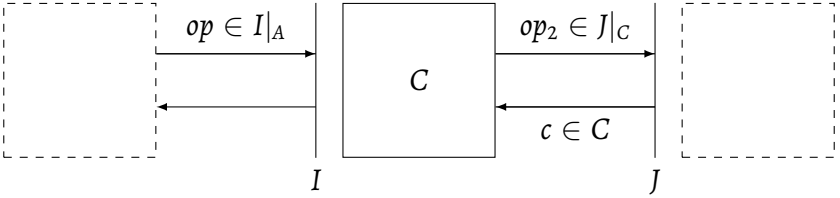
Use Case: Presentation



The component C

1. Receives a computational request op through I
2. Sends a computational request op_1 to J
3. Waits for a result b

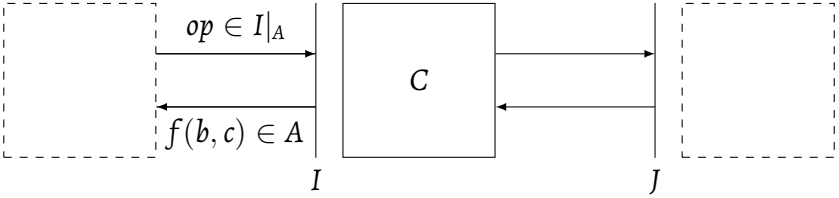
Use Case: Presentation



The component C

1. Receives a computational request op through I
2. Sends a computational request op_1 to J
3. Waits for a result b
4. Sends a computational request op_2 to J
5. Waits for a result c

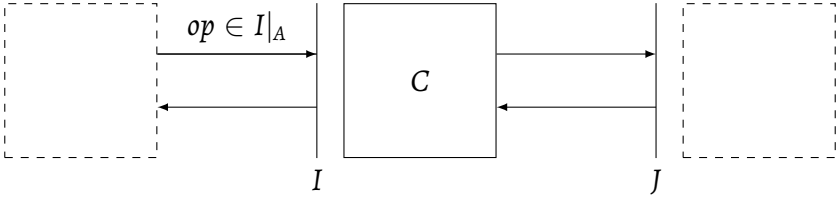
Use Case: Presentation



The component C

1. Receives a computational request op through I
2. Sends a computational request op_1 to J
3. Waits for a result b
4. Sends a computational request op_2 to J
5. Waits for a result c
6. Computes the result of op

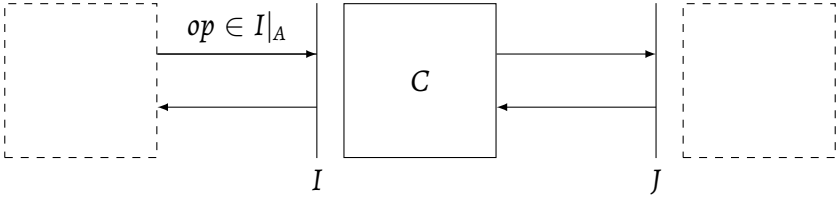
Use Case: Verification



Hypothesis: $\mathbb{P}_I(op)$
The component which uses I does it correctly

$$\frac{\mathbb{P}_I(op)}{\vdash}$$

Use Case: Verification

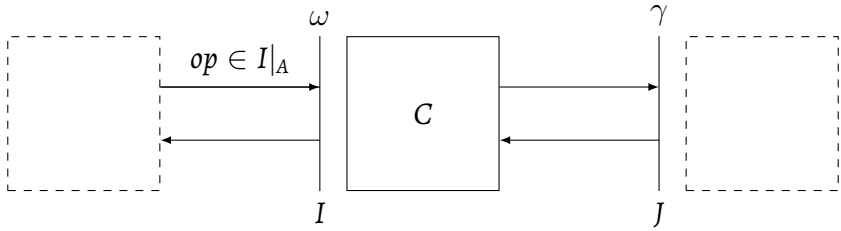


Hypothesis: $\mathbb{P}_I(op)$
The component which uses I does it correctly

Problem: Precondition (and postcondition) may vary in time

$$\frac{\mathbb{P}_I(op)}{\vdash}$$

Use Case: Verification

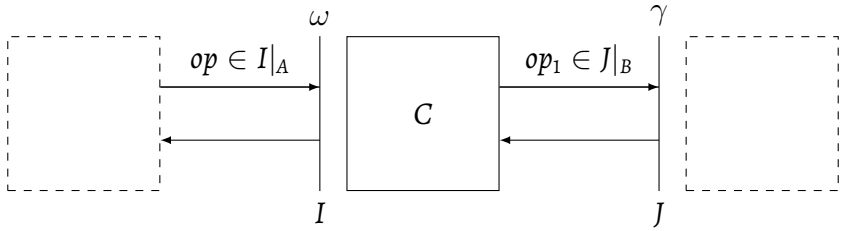


Solution: Parameterize precondition and postcondition by an abstract state

Hypothesis: $\mathbb{P}_I(op, \omega)$

$$\frac{\mathbb{P}_I(op, \omega)}{\vdash}$$

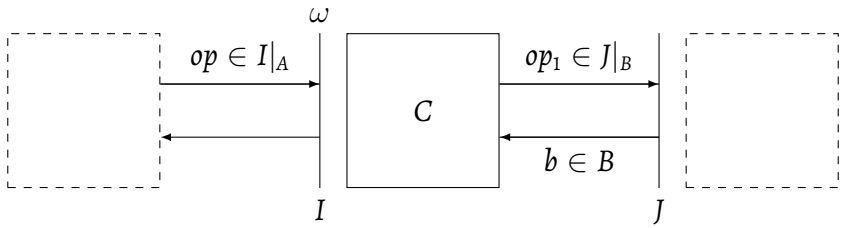
Use Case: Verification



Goal: $\mathbb{P}_J(op_1, \gamma)$
C shall correctly use J

$$\frac{\mathbb{P}_I(op, \omega)}{\vdash \mathbb{P}_J(op_1, \gamma)}$$

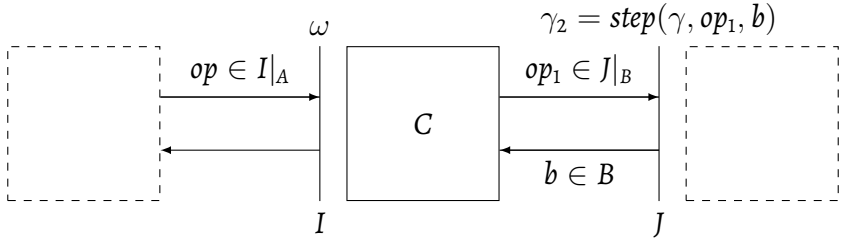
Use Case: Verification



Hypothesis: $\mathbb{P}_J(op_1, \gamma) \Rightarrow \mathbb{Q}_J(op_1, b, \gamma)$
The component which exposes J is correct

$$\frac{\mathbb{G}_J(op_1, b, \gamma) \quad \mathbb{P}_J(op_1, \gamma) \quad \mathbb{P}_I(op, \omega)}{\vdash}$$

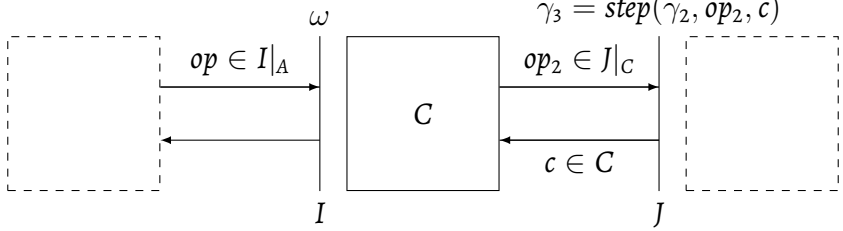
Use Case: Verification



Updating the abstract state attached to J updates the related pre and postcondition

$$\frac{\gamma_2 = step(\gamma, op_1, b) \quad \mathbb{G}_J(op_1, b, \gamma) \quad \mathbb{P}_J(op_1, \gamma) \quad \mathbb{P}_I(op, \omega)}{\vdash}$$

Use Case: Verification

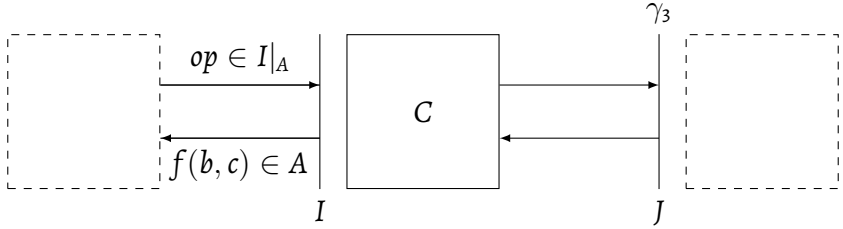


Goal: $\mathbb{P}_J(op_2, \gamma_2)$
C shall correctly use J

Hypothesis: $\mathbb{P}_J(op_2, \gamma_2) \Rightarrow \mathbb{Q}_J(op_2, c, \gamma_2)$
The component which exposes J is correct

$$\frac{\begin{array}{l}\gamma_3 = step(\gamma_2, op_2, c)\\ \mathbb{G}_J(op_2, c, \gamma_2)\\ \mathbb{P}_J(op_2, \gamma_2)\\ \gamma_2 = step(\gamma, op_1, b)\\ \mathbb{G}_J(op_1, b, \gamma)\\ \mathbb{P}_J(op_1, \gamma)\\ \mathbb{P}_I(op, \omega)\end{array}}{\vdash}$$

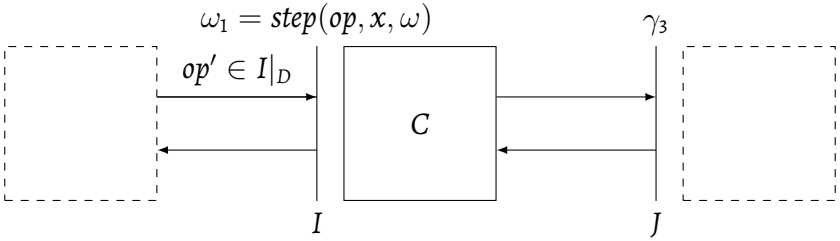
Use Case: Verification



Goal: $\mathbb{Q}_I(op, f(b,c), \omega)$
C shall correctly implement I

$$\frac{\begin{array}{l}\gamma_3 = step(\gamma_2, op_2, c)\\ \mathbb{G}_J(op_2, c, \gamma_2)\\ \mathbb{P}_J(op_2, \gamma_2)\\ \gamma_2 = step(\gamma, op_1, b)\\ \mathbb{G}_J(op_1, b, \gamma)\\ \mathbb{P}_J(op_1, \gamma)\\ \mathbb{P}_I(op, \omega)\end{array}}{\vdash \mathbb{Q}_J(op, f(b,c), \omega)}$$

Additional Challenges



- It is a co-inductive problem
  - Same problem with op', ω_1 and γ_2
  - Invariant-based reasoning
- C carries its own mutable state

Programs with Effects & Effect Handlers

$\rho : P_I(A)$ A program of effects of I whose result belongs to A

$\sigma : \Sigma_I$ A handler to compute results for effects of I

$$evalProgram_A : P_I(A) \times \Sigma_I \to A$$

Implementation

We rely on the `Program` monad (Apfelmus 2010)

- Easy instrumentation
- Introspection
- Extraction-compatible

Modeling

C is modeled

- In terms of programs of effects of J

$$C : I|_A \times S \to P_J(A \times S)$$

- As a handler for I

$$derive_C : \Sigma_J \times S \to \Sigma_I$$

```
match i with
| ReadMem a true (* ---- Privileged Read Access ---------- *)
  => read_dram a
(* ------------------------------------------------------- *)
| ReadMem a false (* --- Unprivileged Read Access---------- *)
  => s <- get_smram_lock                                    ;
     if andb (Smram_bool a) s
     then read_vga a
     else read_dram a
(* ------------------------------------------------------- *)
| WriteMem a v true (* - Privileged Write Access ---------- *)
  => write_dram a v
(* ------------------------------------------------------- *)
| WriteMem a v false (*  Unprivileged Write Access -------- *)
  => s <- get_smram_lock                                    ;
     if andb (Smram_bool a) s
     then write_vga a v
     else write_dram a v
(* ------------------------------------------------------- *)
end.
```

FreeSpec

A compositional framework for the Coq proof assistant

- Complete implementation of our formalism
  - Interfaces composition theorems
  - Invariant-based reasoning for component correctness
  - Coq tactics to automate the verification process
- Verification of a simplified Memory Controller
- Around 8 000 lines of code

`https://github.com/ANSSI-FR/coq-prelude`
`https://github.com/ANSSI-FR/FreeSpec`

Agenda

Context & Contributions

Specifying and Verifying HSE Mechanisms

Modular Verification of Complex Systems

Conclusion & Perspectives

Conclusion

- Hardware-based Security Enforcement mechanisms as a starting point
  - Foundation of computing platform security
  - Subject to compositional attacks
- A methodology to specify and verify HSE mechanisms
- A compositional reasoning framework to break down the hardware model

Perspectives

- Adapt our theory of HSE mechanisms to FreeSpec hardware models
- Leverage our theory of HSE mechanisms to verify system software components
- Introduce a model validation framework for FreeSpec

- SpecCert: Specifying and Verify Hardware-based Security Enforcement Mechanisms
  Thomas Letan, Pierre Chifflier, Guillaume Hiet, Pierre Néron, Benjamin Morin, Formal Methods 2016
- Modular Verification of Program with Effects and Effect Handlers in Coq
  Thomas Letan, Yann Régis-Gianas, Pierre Chifflier, Guillaume Hiet, Formal Methods 2018

`https://github.com/lthms/SpecCert`
`https://github.com/ANSSI-FR/FreeSpec`

Questions?

Bibliography I

- Apfelmus, Heinrich (2010). *The* `operational` *package*. `https://hackage.haskell.org/package/operational`.
- Barthe, Gilles et al. (2014). "System-level Non-Interference for Constant-Time Cryptography." In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*. ACM, pp. 1267–1279.
- Bauer, Andrej and Matija Pretnar (2015). "Programming with Algebraic Effects and Handlers." In: *Journal of Logical and Algebraic Methods in Programming* 84.1, pp. 108–123.
- Goel, Shilpi et al. (2014). "Simulation And Formal Verification Of x86 Machine-Code Programs That Make System Calls." In: *Formal Methods in Computer-Aided Design (FMCAD), 2014*. IEEE, pp. 91–98.
- Gu, Ronghui et al. (2016). "CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.." In: *OSDI*. Vol. 16, pp. 653–669.

Bibliography II

- Jomaa, Narjes et al. (2016). "Formal Proof of Dynamic Memory Isolation Based on MMU." In: *Theoretical Aspects of Software Engineering (TASE), 2016 10th International Symposium on*. IEEE, pp. 73–80.
- Kaivola, Roope et al. (2009). "Replacing Testing with Formal Verification in Intelˆ{\scriptsize\circledR} CoreTM i7 Processor Execution Engine Validation." In: *International Conference on Computer Aided Verification*. Springer, pp. 414–429.
- Klein, Gerwin et al. (2009). "seL4: Formal verification of an OS kernel." In: *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, pp. 207–220.
- Leslie-Hurd, Rebekah, Dror Caspi, and Matthew Fernandez (2015). "Verifying Linearizability of Intel® Software Guard Extensions." In: *International Conference on Computer Aided Verification*. Springer, pp. 144–160.

Bibliography III

- Letan, Thomas, Pierre Chifflier, et al. (2016). "SpecCert: Specifying and Verifying Hardware-based Security Enforcement." In: *21st International Symposium on Formal Methods (FM 2016)*. Springer.
- Letan, Thomas, Yann Régis-Gianas, et al. (2018). "Modular Verification of Programs with Effects and Effect Handlers." In: *28st International Symposium on Formal Methods (FM 2018)*. Springer.
- Lie, David et al. (2003). "Specifying and Verifying Hardware for Tamper-Resistant Software." In: *Security and Privacy, 2003. Proceedings. 2003 Symposium on*. IEEE, pp. 166–177.
- Morrisett, Greg et al. (2012). "RockSalt: Better, Faster, Stronger SFI for the x86." In: *ACM SIGPLAN Notices*. Vol. 47. 6. ACM, pp. 395–404.
- Pnueli, Amir (1985). "In Transition from Global to Modular Temporal Reasoning about Programs." In: *Logics and models of concurrent systems*. Springer, pp. 123–144.

Bibliography IV

- Reid, Alastair (2016). "Trustworthy specifications of ARM® v8-A and v8-M system level architecture." In: *Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design*. FMCAD Inc, pp. 161–168.
- Schneider, Fred B. (2000). "Enforceable Security Policies." In: *ACM Transactions on Information and System Security (TISSEC)* 3.1, pp. 30–50.