Formal Development of the Pip Protokernel

Narjes Jomaa, David Nowak, Paolo Torrini

Joint work with the Pip team

ENTROPY 2018

January 25, 2018

This work is partially supported by the European Celtic-Plus Project ODSI C2014/2-12.
The Pip protokernel: a brief system overview (David Nowak)

Pip design principles and security properties (Narjes Jomaa)

From the executable specification to C (Paolo Torrini)
The Pip protokernel: a brief system overview (David Nowak)

Pip design principles and security properties (Narjes Jomaa)

From the executable specification to C (Paolo Torrini)
This research is part of the European project ODSI.
- Led by Orange
- 1 academic partner: The university of Lille
- 8 industrial partners from France, Romania, and Spain
- In Lille: 3 PhD students and 1 postdoctoral researcher.
- The Pip protokernel is one of the foundations of this project.
- Security protocols are designed on top of Pip.
- Case studies by industrial partners: IoT, M2M, SCADA
- Common Criteria certification
Memory isolation between applications

Why? For safety and security

How? By software (OS kernel), and hardware (MMU, CPU kernel mode)

Correct? Ensured by a formal proof in Coq

Feasible? Yes, by reducing the trusted computing base to its bare bone

reducing the TCB ⇒ increasing feasibility of a formal proof & reducing the attack surface

simplifying the specification language ⇒ increasing feasibility of verified translation to C
From monolithic kernel to the Pip protokernel

Applications

File System  Device Drivers

IPC  Scheduling

Multiplexing

Virtual Memory  Control Switching

Monolithic Kernel
From monolithic kernel to the Pip protokernel

Applications

File System    Device Drivers

IPC    Scheduling

Multiplexing

Virtual Memory    Control Switching

Microkernel
From monolithic kernel to the Pip protokernel

- Applications
- File System
- Device Drivers
- IPC
- Scheduling
- Multiplexing
- Virtual Memory
- Control Switching

Exokernel / Hypervisor
From monolithic kernel to the Pip protokernel

Applications

File System    Device Drivers

IPC    Scheduling

Multiplexing

Virtual Memory    Control Switching

The Pip protokernel
Partition tree

Pip organizes the memory into hierarchical partitions.

Example

```
    p1.1  p1.2  p1.3  p2.1  p2.2
    /    /    /    /    /
   Linux FreeRTOS

user space  multiplexer

kernel space  Pip
```
Partition tree: the point of view of Pip

The contents of each partition is not relevant for Pip.

- **Horizontal isolation**
  Partitions in different subtrees are isolated from each other, e.g. $P_{1.1}$ cannot access memory of $P_{1.2}$ or $P_2$.

- **Vertical sharing**
  A partition has access to the memory of its descendants.

- **Kernel isolation**
  Pip is isolated from all partitions.

```
   P1.1   P1.2   P1.3   P2.1   P2.2
      |       |       |
  __________|_______|_______|
         |       |       |
     P1     |       |       |
         |       |       |
  __________|_______|_______|
         |       |       |
     P2     |       |       |
         |       |       |
  __________|_______|_______|
         |       |       |
 user space |       |       |
         |       |       |
  __________|_______|_______|
         |       |       |
 kernel space |       |       |
         |       |       |
  __________|_______|_______|
         |       |       |
       P_root |       |       |
         |       |       |
  __________|_______|_______|
         |       |       |
       Pip |       |       |
```
Partition tree: dealing with interrupts

- **Software interrupts**
  - Pip deals with software interrupts to itself, e.g. FreeRTOS asks Pip to create a new partition.
  - Pip forwards other software interrupts to the caller’s parent, e.g. $p_{1.2}$ make a system call to Linux.

- Pip forwards **hardware interrupts** to the root partition, e.g. a network packet has arrived.
Pip system calls

10 elementary system calls

▶ Memory management
  - createPartition: creates a child partition
  - removePartition: deletes a child partition
  - addVaddr: lends a memory page to a child
  - removeVaddr: removes a memory page from a child
  - pageCount: the number of needed configuration pages
  - prepare: gives needed configuration pages
  - collect: takes back unused configuration pages
  - mappedInChild: returns the child using a given page

▶ control switching
  - dispatch: notifies a partition about an interrupt
  - resume: restores the context of a partition
Software layers

- Hardware
- Pip Hardware Abstraction Layer (HAL)
- Pip service layer
- A sub-partition
- Another sub-partition
- Root partition
- A sub-sub-partition
- Another sub-sub-partition

User mode
Kernel mode

Language:
- Gallina (the language of the Coq proof assistant)
- C and assembly language
- Any language
Applications

- The HAL of Pip has been ported to:
  - QEMU (x86)
  - x86
  - The Galileo board (Intel Pentium-compliant embedded board)

- Kernels ported on Pip
  - FreeRTOS: Tasks can be isolated in sibling partitions.
  - Linux 4.10.4: More involved because Linux configures MMU.

- Porting a kernel to Pip essentially consists of:
  - removing privileged instructions and operations, and
  - replacing them with system calls to Pip (paravirtualization).

- Drhystone benchmark: low overhead of 2.6% in terms of CPU cycles
Formal verification

- Formal verification of an executable specification of Pip
  Addressed by Narjes Jomaa in the next part of this presentation

- Verified translation of the executable specification into C
  Addressed by Paolo Torrini in the final part of this presentation
The Pip protokernel: a brief system overview (David Nowak)

Pip design principles and security properties (Narjes Jomaa)

From the executable specification to C (Paolo Torrini)
The configuration of a partition

- Partition descriptor ($PD$)
- MMU tables
- Shadow 1 ($SH1$) and Shadow 2 ($SH2$)
- Linked list ($L$)
MMU briefly

Data structure of partitions

- **MMU structure**: Define assigned pages and access control
- **Mirror the MMU structure**
  - **Shadow 1**: Find out which pages are assigned to children and which pages are used as a partition descriptor identifier (*security*)
  - **Shadow 2**: Ease getting back the ownership of assigned pages (*efficiency*)
- **List (L)**: Ease getting back the ownership of pages lent to the kernel (*efficiency*)
Pip design principles

- Hardware state: the part that is relevant to model the partition tree
  - the partition that is currently active
  - the physical memory where Pip stores its own data
- Exclude the use of all objects that would require a GC: lists, trees → Encoding these structure in physical memory using the HAL
Security properties
The horizontal isolation property

**Definition** \( HI \ s \colon \text{Prop} \colon= \)

\[
\forall \text{parent} \ \text{child1} \ \text{child2} \colon \text{page},
\]

\[
\text{parent} \in (\text{partitionTree} \ s) \rightarrow
\]

\[
\text{child1} \in (\text{children} \ \text{parent} \ s) \rightarrow
\]

\[
\text{child2} \in (\text{children} \ \text{parent} \ s) \rightarrow
\]

\[
\text{child1} \neq \text{child2} \rightarrow
\]

\[
(\text{allocatedPages} \ \text{child1} \ s) \cap (\text{allocatedPages} \ \text{child2} \ s) = \emptyset.
\]

- Sibling partitions cannot access each others memory.
Hierarchical TCB (vertical sharing)

**Definition** \( \text{VS} \ s : \ Prop := \)

\[ \forall \text{parent} \ \text{child} : \text{page}, \]

\[ \text{parent} \in (\text{partitionTree} \ s) \rightarrow \]

\[ \text{child} \in (\text{children} \ \text{parent} \ s) \rightarrow \]

\[ (\text{allocatedPages} \ \text{child} \ s) \subseteq (\text{assignedPages} \ \text{parent} \ s). \]

- All the pages allocated for a partition are included in the pages assigned to its ancestors
The kernel isolation property

Definition $\text{KI}_s : \text{Prop} :=$

$\forall$ partition1 partition2 : page,

$\text{partition1} \in (\text{partitionTree } s) \rightarrow$

$\text{partition2} \in (\text{partitionTree } s) \rightarrow$

$(\text{ownedPages partition1 } s) \cap (\text{kernelPages partition2 } s) = \emptyset$.

- No partition can access the pages owned by the kernel.
Information flow property

- As a corollary to VS and HI: Non-influence property for isolated partition was proved
- Abstract information flow model
- Assumption about hardware side effects
Verification approach
Verification approach

Hoare logic on top of the LLI (Low Level Interface) monad

{{Precondition}} Program {{Postcondition}}

- Program: a monadic function (of type LLI A)
- Precondition: a unary predicate on the starting state
- Postcondition: binary predicate on the returned value and on the ending state

\[
\text{Definition hoareTriple}\{A : \text{Type}\}
\]

\[
(P : \text{state} \to \text{Prop}) \ (m : \text{LLI A})
\]

\[
(Q : A \to \text{state} \to \text{Prop}) : \text{Prop} :=
\]

\[
\forall s, \ P s \to \text{match m s with}
\]

\[
| \ \text{val } (a, s') \Rightarrow Q a s' \\
| \ \text{undef } _._ \Rightarrow \text{False}
\]

end.

States that if the precondition holds then

- the postcondition holds; and
- there is no undefined behavior
The need of consistency properties

- We cannot prove the following invariant
  \[
  \{\{\text{HI & VS & KI}\}\} \text{ API\_service } \{\{\text{HI & VS & KI}\}\}
  \]

- Properties about the Pip’s data structure are missing
  - The precondition should be strengthened with consistency properties
  - The consistency properties must also be preserved
  \[
  \{\{\text{HI & VS & KI & C}\}\} \text{ API\_service } \{\{\text{HI & VS & KI & C}\}\}
  \]

- \(\text{consistency} \approx \text{well-formedness of Pip’s data structures}\)
Example: createPartition invariant

\{\{\text{HI} \&\ \text{VS} \&\ \text{KI} \&\ \text{C}\}\} \ \text{createPartition} \ v1 \ v2 \ v3 \ v4 \ v5 \ \{\{\text{HI} \&\ \text{VS} \&\ \text{KI} \&\ \text{C}\}\}
Proceed forward using transitivity (1/2)

```
{{HI & VS & KI & C}}

perform currentPart := getCurPartition in
perform ptv1FromPD := getTableAddr currentPart v1 nbL in
...
if negb accessv1 then ret false else
  writeAccessible ptv1FromPD idxv1 false ;;
...

{{HI & VS & KI & C}}
```
Proceed forward using transitivity (2/2)

First sub-goal:

\{\{HI & VS & KI & C}\}\}

getCurPartition

\{\{HI & VS & KI & C & P currentPart \}\}  

Second sub-goal:

\{\{HI & VS & KI & C & P currentPart\}\}

perform ptv1FromPD := getTableAddr currentPart v1 nbL in
...
if negb accessv1 then ret false else
writeAccessible ptv1FromPD idxv1 false ;;
...
\{\{HI & VS & KI & C\}\}
Verification overview

<table>
<thead>
<tr>
<th>Invariants</th>
<th>line of proof</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>createPartition (300 loc)</code></td>
<td>≈ 60000</td>
</tr>
<tr>
<td><code>createPartition + addVaddr (50 loc)</code></td>
<td>≈ 78000</td>
</tr>
<tr>
<td><code>createPartition + addVaddr + mappedInChild(20 loc)</code></td>
<td>≈ 78250</td>
</tr>
</tbody>
</table>

**Table:** Overview of the proof
The Pip protokernel: a brief system overview (David Nowak)

Pip design principles and security properties (Narjes Jomaa)

From the executable specification to C (Paolo Torrini)
Translating to C

Coq executable model and extracted OCaml code:

- needs big runtime environment
- not efficient enough

We need a translation to low level languages:

- HAL: manual implementation in assembly and C
- Service Layer: C code automatically generated from Gallina
- currently compiled by GCC
Translating to C

Coq executable model and extracted OCaml code:
  ▶ needs big runtime environment
  ▶ not efficient enough

We need a translation to low level languages:
  ▶ HAL: manual implementation in assembly and C
  ▶ Service Layer: C code automatically generated from Gallina
  ▶ currently compiled by GCC

However: we want a verified translation to CompCert C
  ▶ certified compilation
  ▶ tail-recursive optimisation
Pip monadic code (MC)

- Low-level HAL primitives
- Higher-level monadic code (MC)

Fixpoint initVTable timeout shadow1 idx :=
  match timeout with
  | 0 ⇒ ret tt
  | S timeout1 ⇒
    perform max := getMaxIndex in
    perform res := Index.ltb idx max in
    if (res) then
      perform daddr := getDefaultVAddr in
      writeVirEntry shadow1 idx daddr ;;
      perform nidx := Index.succ idx in
      initVTable timeout1 shadow1 nidx
    else ...
  end.
Translation to C

We use a Haskell-implemented translator (*digger*) to translate from the Gallina AST of MC to C.
MC is a shallow embedding, i.e. a semantic representation of a language in Coq, based on a set of Gallina definitions.

\[
\text{Definition } \text{ret} : A \rightarrow \text{LLI } A := \text{fun } a \ s \Rightarrow \text{val } (a, s).
\]

\[
\text{Definition } \text{bind} : \text{LLI } A \rightarrow (A \rightarrow \text{LLI } B) \rightarrow \text{LLI } B := \\
\quad \text{fun } m \ f \ s \Rightarrow \text{match } m \ s \text{ with} \\
\quad \mid \text{val } (a, s') \Rightarrow f a s' \\
\quad \mid \text{undef } a s' \Rightarrow \text{undef } a s' \text{ end.}
\]

\[
\text{perform } x := m \text{ in } e \text{ for } \text{bind } m \ (\text{fun } x \Rightarrow e) \\
\quad m ;; e \text{ for } \text{bind } m \ (\text{fun } _ \Rightarrow e)
\]

Value types: \textit{bool} and subtypes of \textit{nat}
Example: a function defined in Coq, using the monadic code:

```coq
Definition getFstShadow (partition : page) : LLI page :=
  perform idx := getSh1idx in
  perform idxSucc := Index.succ idx in
  readPhysical partition idxSucc.
```

and its generated translation to C:

```c
uintptr_t getFstShadow (const uintptr_t partition) {
  const uint32_t idx = getSh1idx ();
  const uint32_t idxSucc = succ (idx);
  return readPhysical (partition, idxSucc);
}
```
Problem: generating verified code

General solution: define a semantic translation from weak to strong (w.r.t. types), and reverse it

However: we do not want to define a semantics of C in Coq, we want to use an existing one which also provides compilation – CompCert C.
Verified translation: our approach

1. we build a Coq representation of MC as a deep embedding (DEC) and specify formally its semantics
   – operationally, implementing an SOS interpreter
   – denotationally, as interpretation of DEC into Gallina

2. use the denotational semantics to verify the translation of Pip into DEC

3. use the operational semantics to verify the translation to CompCert C
Translation through DEC

DEC is defined in terms of abstract datatypes: possible to manipulate it as an object in Coq – e.g. to define a formal translation from it

```
MC
- ;; -
perform := _ in _
if _ then _ else _
(MC. F _ )
(HAL. F _ )
```

```
DEC
BindN _ _
BindS _ _ _
IfThenElse _ _ _
Apply _ _
Modify _ _ _
```

Haskell tool

Coq function

CompCert C
For the two semantics to agree:

for $P$ a DEC program, $\text{DEC2MC4val} (\text{SOS\_Int } P) = \text{DEC2MC } P$

$\text{Pip} = \text{DEC2MC} (\text{Haskell\_MC2DEC } \text{Pip})$
From DEC to C

Semantic soundness: need for a proof that behaviour is preserved.

Essentially – like adding a compilation step.
DEC expressions

\[
\text{Inductive } \text{Exp} : \text{Type} := \\
\quad \mid \text{Val} \ (v: \text{Value}) \mid \text{Var} \ (x: \text{Id}) \\
\quad \mid \text{BindN} \ (e_1: \text{Exp}) \ (e_2: \text{Exp}) \\
\quad \mid \text{BindS} \ (x: \text{Id}) \ (t: \text{option VTyp}) \ (e_1: \text{Exp}) \ (e_2: \text{Exp}) \\
\quad \mid \text{IfThenElse} \ (e_1: \text{Exp}) \ (e_2: \text{Exp}) \ (e_3: \text{Exp}) \\
\quad \mid \text{Apply} \ (f: \text{Id}) \ (\text{prms: Prms}) \ (\text{fuel: Exp}) \\
\quad \mid \text{Modify} \ (t_1 \ t_2: \text{VTyp}) \ (xf: \text{XFun t1 t2}) \ (\text{prm: Exp}) \\
\quad \mid \text{BindMS} \ (\text{env: valEnv}) \ (e: \text{Exp}) \\
\quad \mid \text{Call} \ (f: \text{Id}) \ (\text{prms: Prms}) \\
\text{with Prms : Type := PS (es: list Exp).}
\]

Recursive functions terminate (as in MC)
Parameter \textbf{Id}: Type.
Parameter \textbf{State}: Type.

\textbf{Inductive Fun} : Type :=
\textbf{FC} \ (\textit{formal} _{prms}: \textit{list} \ (\textit{Id} \ * \ \textit{VTyp}) \ (\textit{ret} _{type}: \ \textit{VTyp})
\ (\textit{default}: \ \textit{Value}) \ (\textit{body}: \ \textit{Exp}) .

\textbf{Record XFun} \ (\textit{dt1} \ \textit{dt2}: \ \textit{VTyp}) : \ Type :=
\{ \textit{x}_\text{modify} : \ \textit{State} \rightarrow (\textit{mcTyp} \ \textit{dt1}) \rightarrow \ \textit{State} \ * \ (\textit{mcTyp} \ \textit{dt2}) \}. 
Operational semantics (small-step)

\[ \phi \text{ function environment} \quad \delta \text{ datavalue environment} \]

\textbf{Static:}

\begin{align*}
\vdash \phi &:: \Phi \\
\Phi; \Delta &\vdash \text{exp} :: \text{vtyp} \\
\vdash \text{well\_typed} \phi &
\end{align*}

\begin{align*}
\vdash \delta &:: \Delta \\
\Phi; \Delta &\vdash \text{prms} :: \text{ptyp}
\end{align*}

\textbf{Dynamic:}

\begin{align*}
\phi; \delta &\vdash (\text{state}, \text{fuel}, \text{exp}) \rightarrow (\text{state}', \text{fuel}', \text{exp}') \\
\phi; \delta &\vdash (\text{state}, \text{fuel}, \text{prms}) \rightarrow (\text{state}', \text{fuel}', \text{prms}')
\end{align*}
Type soundness (SOS interpreter)

Type soundness for expressions (similarly for parameters):

\[ \forall \Phi \Delta \text{ exp vtyp}, \quad \Phi; \Delta \vdash \text{exp :: vtyp} \rightarrow \]
\[ \forall \phi \delta \text{ state fuel}, \quad \vdash \text{well_typed } \phi \rightarrow \]
\[ \vdash \phi :: \Phi \rightarrow \]
\[ \vdash \delta :: \Delta \rightarrow \]

\[ \Sigma! \text{ state’ fuel’ v}, \]
\[ \phi; \delta \vdash (\text{state, fuel, exp}) \rightarrow (\text{state’, fuel’, Val v}) \]

Proved in Coq, by double induction on fuel and the mutually defined typing relations.
Operational semantics (Coq code)

Inductive ExpTyping :
  list (Id*FTyp) → list (Id*Value) → Exp → VTyp → Type
with PrmsTyping :
  list (Id*FTyp) → list (Id*Value) → Prms → PTyp → Type

Inductive FEnv_WT (fenv: list (Id*Fun)) : Type

Inductive AConfig (T: Type) : Type :=
  Conf (state: W) (fuel: nat) (qq: T)

Inductive EStep (fenv: list (Id*Fun)) :
  list (Id*FCall) → list (Id*Value) →
  AConfig Exp → AConfig Exp → Type
with PrmsStep (fenv: list (Id*Fun)) :
  list (Id*FCall) → list (Id*Value) →
  AConfig Prms → AConfig Prms → Type
Denotational semantics

\[ \Theta_e : \Theta_t \text{funEnv} \rightarrow \Theta_t \text{valEnv} \rightarrow \forall e : \text{Exp}, \ \text{ILL State} (\Theta_t (\tau e)) \]

\[
\begin{align*}
\Theta_e \cdot_\cdot (\text{Val } v) &= \text{ret } (\text{ext } v) \\
\Theta_e \cdot \text{VS } (\text{Var } x) &= \text{ret } (\text{find } x \ \text{VS}) \\
\ldots
\end{align*}
\]

\[
\begin{align*}
\Theta_e \text{FS VS } (\text{BindS } x \cdot e_1 \cdot e_2) &= \text{let } t = \Theta_t (\tau e_1) \ \text{in} \\
&\quad \text{bind } (\Theta_e \text{FS VS } e_1) (\Theta_e \text{FS } ((x, t) :: \text{VS}) e_2) \\
\ldots
\end{align*}
\]

\[
\begin{align*}
\Theta_e \text{FS VS } (\text{Call } f \ \text{prms}) &= \\
&\quad \text{bind } (\Theta_{es} \text{FS VS } prms) (\text{find } f \ \text{FS}) \\
\Theta_e \text{FS VS } (\text{Modify } xf \ \text{prm}) &= \\
&\quad \text{bind } (\Theta_e \text{FS VS } prm) (x_{\text{modify }} xf)
\end{align*}
\]

Provable in Coq: the two semantics (operational and denotational) agree
Summarising

META-LANGUAGE (Coq–Gallina)

Types
Check

MC
programs
(Pip model)

Compute
Values

DEC2MC

SOS REPRESENTATION (DEC)

Types
Check

programs
Compute
Values

CompCert C

Types
Check

programs
Compute
Values

MC2DEC
(in Haskell)

X
System:
Q. Bergougnoix, N. Jomaa, M. Yaker, J. Cartigny, G. Grimaud, S. Hym, D. Nowak,
Proved Memory Isolation in Real-Time Embedded Systems through Virtualization,
submitted

Formal modelling and verification of security properties:
N. Jomaa, P. Torrini, D. Nowak, G. Grimaud,
Proof-oriented Design of a Separation Kernel with Minimal TCB,
submitted

Translation:
S. Hym, V. Oudjail, Digger: Haskell repository, https://github.com/2xs/digger
To find out more

http://pip.univ-lille1.fr

The Pip Development Team thanks you for your attention