Deep Specifications and Certified Abstraction Layers

Ronghui Gu  Jérémie Koenig  Tahina Ramananandro  Zhong Shao
Newman Wu  Shu-Chun Weng  Haozhong Zhang\textsuperscript{1}  Yu Guo\textsuperscript{1}

Yale University  \textsuperscript{1}University of Science and Technology of China

January 17, 2015

http://flint.cs.yale.edu
Motivation

How to build reliable & secure system software stacks?
Motivation

Android architecture & system stack

Motivation

Visible software components of the Linux desktop stack

From http://en.wikipedia.org/wiki/Linux
Motivation

Software stack for HPC clusters

From http://www.hpcwire.com/2014/02/24/comprehensive-flexible-software-stack-hpc-clusters/
Motivation

Cisco’s FAN (Field-Area-Network) protocol layering

From https://solutionpartner.cisco.com/web/cegd/overview
Motivation

Apollo Mobile Communication Stack
http://www.layer2connections.com/apollo_clients.html

Web Application Development Stack
From http://www.brightware.co.uk/Technology.aspx
Motivation (cont’d)

• Common themes: all system stacks are built based on abstraction, modularity, and layering

• Abstraction layers are ubiquitous!

Such use of abstraction, modularity, and layering is “the key factor that drove the computer industry toward today’s explosive levels of innovation and growth because complex products can be built from smaller subsystems that can be designed independently yet function together as a whole.”

Do We Understand Abstraction?

**In the PL community:**

(abstract in the small)

- Mostly formal but tailored within a single programming language (ADT, objects, existential types)
- Specification only describes type or simple pre- & post condition
- Hide concrete data representation (we get the nice `repr. independence` property)
- Well-formed `typing` or `Hoare-style judgment` between the impl. & the spec.

**In the System world:**

(abstract in the large)

- Mostly informal & language-neutral (APIs, sys call libraries)

---

**Something magical going on ...**

What is it? between the impl. & the spec.
Problems

• What is an abstraction layer?
• How to formally specify an abstraction layer?
• How to program, verify, and compile each layer?
• How to compose abstraction layers?
• How to apply certified abstraction layers to build reliable and secure system software?
Our Contributions

- We introduce **deep specification** and present a language-based formalization of **certified abstraction layer**

- We developed new languages & tools in Coq
  - A formal layer calculus for composing certified layers
  - ClightX for writing certified layers in a C-like language
  - LAsm for writing certified layers in assembly
  - CompCertX that compiles ClightX layers into LAsm layers

- We built multiple **certified OS kernels** in Coq
  - mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest
What is an Abstraction Layer?

- overlay $L_2$
  - abs-state
  - memory
  - primitives

- underlay $L_1$
  - abs-state
  - memory
  - primitives

C or Asm module implementation

$M$

$R$

$abs$

$mem$

$+$

$-$
**Example: Page Tables**

**concrete C types**

```c
struct PMap {
    char * page_dir[1024];
    uint page_table[1024][1024];
};
```

**abstract Coq spec**

```coq
Inductive PTPerm : Type :=
    | PTP
    | PTU
    | PTK.

Inductive PTEInfo :=
    | PTEValid (v : Z) (p : PTPerm)
    | PTEUnPresent.

Definition PMap := ZMap.t PTEInfo.
```
Example: Page Tables

abstract state

\[ \text{PMap} := \text{ZMap}.t \ PTE\text{Info} \]
\[ (\ast \ vaddr \rightarrow (paddr, \ perm) \ast) \]

Invariants: kernel page table is a direct map; user parts are isolated

abstract primitives (Coq functions)

Function page_table_init = …
Function page_table_insert = …
Function page_table_rmv = …
Function page_table_read = …

memory

char * page_dir[1024];
uint page_table[1024][1024];

C functions

int page_table_init() { … }
int page_table_insert { … }
int page_table_rmv() { … }
int page_table_read() { … }
Formalizing Abstraction Layers

What is a **certified** abstraction layer \((L_1, M, L_2)\)?

- **overlay interface**
- **spec \(L_2\)** with abstract state \(abs\)
- **simulation (implements) relation** \(R(abs, mem)\)
- **module \(M\)** with concrete state: \(mem\)
- **calling abstract primitives in \(L_1\)**
- **underlay interface**
- **spec \(L_1\)**

Recorded as the **well-formed layer** judgment

\[ L_1 \vdash_R M : L_2 \]
The Simulation Relation

$\mathcal{L}_1 \vdash_R \mathcal{M} : \mathcal{L}_2$ \quad $\Rightarrow$ \quad $\mathcal{L}_2 \leq_R \llbracket \mathcal{M} \rrbracket \mathcal{L}_1$

**Forward Simulation:**

- Whenever $L_2(f)$ takes $\text{abs}_1$ to $\text{abs}_2$ in one step, and $R(\text{abs}_1, \text{mem}_1)$ holds,
- then there exists $\text{mem}_2$ such that $\llbracket \mathcal{M} \rrbracket (L_1)(f)$ takes $\text{mem}_1$ to $\text{mem}_2$ in zero or more steps, and $R(\text{abs}_2, \text{mem}_2)$ also holds.

For each function $f$ in $\text{Dom}(L_2)$, the simulation relation is compositional per-module semantics $\llbracket \cdot \rrbracket$.
Reversing the Simulation Relation

$L_1 \vdash_R M : L_2 \quad \Rightarrow \quad L_2 \leq_R \sem{M} L_1$

If $\sem{M}(L_1)$ is deterministic relative to external events (\textit{a la} CompCert)

$\sem{M} L_1 \leq_R L_2$

$\sem{M} L_1 \sim_R L_2$

$\sem{M}(L_1)$ and $L_2$ simulates each other!

$L_2$ captures everything about running $M$ over $L_1$
Deep Specification

$L_2$ is a deep specification of $M$ over $L_1$ if under any valid program context $P$ of $L_2$, $\left[ P \oplus M \right](L_1)$ and $\left[ P \right](L_2)$ are observationally equivalent.

$L_2$ captures everything about running $M$ over $L_1$.

Making it “contextual” using the whole-program semantics $\left[ \bullet \right]$.
Why Deep Spec is Really Cool?

Deep spec $L$ captures all we need to know about a layer $M$.

- No need to ever look at $M$ again!
- Any property about $M$ can be proved using $L$ alone.

**Impl. Independence**: any two implementations of the same deep spec are *contextually equivalent*.
Is Deep Spec Too Tight?

• Not really! It still *abstracts* away:
  – the *efficient* concrete data repr & impl. algorithms & strategies

• It can still be *nondeterministic*:
  – External nondeterminism (e.g., I/O or scheduler events) modeled as a set of deterministic traces relative to external events (*a la* CompCert)
  – Internal nondeterminism (e.g., sqrt, rand, resource-limit) is also OK, but any two implementations must still be *observationally equivalent*

• It *adds* new logical info to make it *easier-to-reason-about*:
  – auxiliary abstract states to define the full functionality & invariants
  – accurate precondition under which each primitive is valid
Problem w. Shallow Specs

- C or Asm module
- shallow spec A
- shallow spec B

C & Asm Module Implementation

C & Asm Modules w. Shallow Spec A

Want to prove another spec B?

Need to revisit & reverify all the code!
Shallow vs. Deep Specifications

C or Asm module

C & Asm Module Implementation

C & Asm Modules w. Shallow Specs

C & Asm Modules w. Deep Specs

C or Asm module

shallow spec

deep spec
How to Make Deep Spec Work?

No languages/tools today support deep spec & certified layered programming

**Challenges:**

- Implementation done in C or assembly or …
- Specification done in richer logic (e.g., Coq)
- Need to mix both and also simulation proofs
- Need to compile C layers into assembly layers
- Need to compose different layers
Our Contributions

• We introduce **deep specification** and present a language-based formalization of **certified abstraction layer**

• We developed new languages & tools in Coq
  – A formal layer calculus for composing certified layers
  – ClightX for writing certified layers in a C-like language
  – LAsm for writing certified layers in assembly
  – CompCertX that compiles ClightX layers into LAsm layers

• We built multiple **certified OS kernels** in Coq
  – mCertiKOS-hyper consists of 37 layers, took less than one-person-year to develop, and can boot Linux as a guest
What We Have Done

Layer Spec \( L \)

Extended Asm Language \( \text{LAsm} \)

Clight

CompCert

Asm

Parametrize it w. abstract states & primitives in \( L \)

\( \text{LAsm}[L] \)

\( \text{CompCertX}[L] \) compositional compiler

\( \text{ClightX}[L] \)

LayerLib calculus

Layered prog. in ClightX

Layered prog. in \( \text{LAsm} \)

Link everything together

Coq
LayerLib: Vertical Composition

\[
\frac{L_1 \vdash_R M : L_2}{L_1 \vdash_{R \circ S} M \oplus N : L_3}
\]
Example: Thread Queues

High Abs-State: 1 :: 0 :: 2 :: nil

Low Abs-State:
- tcbp(0):
  - tcbp[0]
  - tcbp[1] head
- tcbp(1):
  - tcbp[1] head
  - tcbp[2] tail
- tcbp(2):
  - tcbp[2] tail

Concrete Memory:
- tcbp[0] head
- tcbp[1] head
- tcbp[2] tail

Ready States:
- tcbp(0)
- tcbp(1)
- tcbp(2)
Example: Thread Queues

**C Implementation**
```c
typedef enum {
    TD_READY, TD_RUN,
    TD_SLEEP, TD_DEAD
} td_state;

struct tcb {
    td_state tds;
    struct tcb *prev, *next;
};

struct tdq {
    struct tcb *head, *tail;
};

struct tcb tcbp[64];
struct tdq tdqp[64];

struct tcb * dequeue (struct tdq *q) {
    ...... }
```

**Low Layer Spec in Coq**
```coq
Inductive td_state :=
| TD_READY | TD_RUN |
| TD_SLEEP | TD_DEAD.

Inductive tcb :=
| TCBV (tds : td_state) |
    (prev next : Z)

Inductive tdq :=
| TDQV (head tail: Z)

Record abs ={
    tcbp : ZMap.t tcb;
    tdqp : ZMap.t tdq }

Function dequeue (d : abs) (i : Z) :=
```

**High Layer Spec in Coq**
```coq
Inductive td_state :=
| TD_READY | TD_RUN |
| TD_SLEEP | TD_DEAD.

Definition tcb := td_state.

Definition tdq := List Z.

Record abs' ={
    tcbp : ZMap.t tcb;
    tdqp : ZMap.t tdq }

Function dequeue (d : abs') (i : Z) :=
match (d.tdqp i) with
| h :: q' => Some(set_tdq d i q', h)
| nil => None
end
```
Example: Dequeue

High Abs-State

Concrete Memory

Low Abs-State

Ready 1 2 :: nil

Ready 1 2

Ready 0

Ready 0
Conflicting Abstract States?

$P$ is a client program that interacts with two modules, $M_1$ and $M_2$, through interfaces $L_1$ and $L_2$, respectively. $L_1$ has an abstract state denoted by $abs1$ and $L_2$ has an abstract state denoted by $abs2$. The question arises whether these abstract states are compatible.
LayerLib: Horizontal Composition

- $L_1$ and $L_2$ must have the same abstract state
- both layers must follow the same simulation relation $R$
Programming & Compiling Layers

\[ L \vdash_R M_c : L_1 \quad \Rightarrow \quad L_1 \leq_R \sem{M_c}_{\text{ClightX}}(L) \]

**CompCertX correctness theorem** (where \textit{minj} is a special kind of memory injection)

\[ \sem{M_c}_{\text{ClightX}}(L) \leq_{\text{minj}} \sem{\text{CompCertX}[L](M_c)}_{\text{LAsm}}(L) \]

\[ L_1 \leq_{R \cdot \text{minj}} \sem{\text{CompCertX}[L](M_c)}_{\text{LAsm}}(L) \]

\textit{R} must absorb such memory injection: \( R \cdot \text{minj} = R \) then we have:

\[ L_1 \leq_R \sem{\text{CompCertX}[L](M_c)}_{\text{LAsm}}(L) \]

Let \( M_a = \text{CompCertX}[L](M_c) \) then \( L \vdash_R M_a : L_1 \) (LAsm)
Our Contributions

• We introduce **deep specification** and present a language-based formalization of **certified abstraction layer**

• We developed new languages & tools in Coq
  – A **formal layer calculus** for composing certified layers
  – ClightX for writing certified layers in a C-like language
  – LAsm for writing certified layers in assembly
  – CompCertX that compiles ClightX layers into LAsm layers

• We built multiple **certified OS kernels** in Coq
  – mCertiKOS-hyper consists of **37 layers**, took less than **one-person-year** to develop, and can boot **Linux** as a guest
Case Study: mCertiKOS

Single-core version of CertiKOS (developed under DARPA CRASH & HACMS programs), 3 kloc, can boot Linux

Aggressive use of abstraction over deep specs (37 layers in ClightX & LAsm)
Decomposing mCertiKOS

Based on the abstract machine provided by boot loader

Physical Memory and Virtual Memory Management
(11 Layers)
Decomposing mCertIKOS (cont’d)

Thread and Process Management (14 Layers)
Decomposing mCertiKOS (cont’d)

Virtualization Support (9 Layers)
Decomposing mCertiKOS (cont’d)

Current Target: Single-Core CertiKOS Syscall and Trap Handlers (3 Layers)

Syscall and Trap Handlers (3 Layers)
Variants of mCertiKOS Kernels

(base)

- TRAP
- PROC
- THR
- VM
- MM

(hyp)

- TRAP
- VIRT
- PROC
- THR
- VM
- MM
- PThread
- PSched
- PCID
- PAbQueue
- PTDQInit
- PTDQIntro
- PTCBInit
- PTCBIntro
- PKCtxOp
- PKCtx

(rz)

- TRAP
- VIRT
- PROC
- THR
- VM
- MM
- VVM
- VSVM
- VVMCBOp
- VSVMIntro
- VVMCBInit
- VVMCBIntro
- VSVMSwitch
- VNPTInit
- VNPTIntro

(emb)

- PROC
- THR
- MM
- TSysCall
- TTrap
- TTrapArg
Example: Page Fault Handler

TSysCall

TTrap

TTrapArg

PProc

PUCtx

... 

PCID

... 

PMap

MPTOp

MPTIntro

MAT

MATOp

... 

PreInit

ikern_set 

setcr3 

pf_get
Conclusions

• Great success w. today’s system software … but why?

• We identify, sharpen, & formalize two possible ingredients
  – abstraction over deep specs
  – a compositional layered methodology

• We build new lang. & tools to make layered programming rigorous & certified --- this leads to huge benefits:
  – simplified design & spec; reduced proof effort; better extensibility

• They also help verification in the small
  – hiding implementation details as soon as possible

• Still need better PL and tool support  (Coq / ClightX / LAsm)
Thank You!

Interested in working on the CertiKOS project? we are hiring & recruiting at all levels:

- postdocs,
- research scientists,
- PhD students, and visitors
A Subtlety for LAsm

Some functions (e.g., kernel context switch) do not follow the C calling convention and must be programmed in LAsm\([L]\).

\[ L \vdash_{R} M_{a} : L_{2} \quad \Rightarrow \quad L_{2} \leq_{R} [M_{a}]_{\text{LAsm}}(L) \]

**Problem:** per-module semantics \([M_{a}]_{\text{LAsm}}(L)\) is NOT deterministic relative to external events.

\[ [M_{a}]_{\text{LAsm}}(L) \leq_{R} L_{2} \]

Fortunately, whole-machine semantics \([\bullet]_{\text{LAsm}}(L)\) is deterministic relative to external events, so it can still be reversed:

\[ \forall P. [P \oplus M_{a}]_{\text{LAsm}}(L) \sim_{R} [P]_{\text{LAsm}}(L_{2}) \]
Layer Pattern 1: Getter/Setter

Hide concrete memory; replace it with Abstract State
Only the getter and setter primitives can access memory
Layer Pattern 2: AbsFun

Memory does not change
New implementation code does not access memory directly!
## Development Cost

<table>
<thead>
<tr>
<th>Development Activity</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development of ClightX and CompCertX</td>
<td>10 pm</td>
</tr>
<tr>
<td>Development of VCGen for ClightX</td>
<td>1.5 pm</td>
</tr>
<tr>
<td><strong>Verification of mm</strong></td>
<td></td>
</tr>
<tr>
<td>Design: first 3 layers</td>
<td>0.5 pm</td>
</tr>
<tr>
<td>Design: the rest 8</td>
<td>0.5 pm</td>
</tr>
<tr>
<td>Refinement Proof: first 2</td>
<td>1.2 pm</td>
</tr>
<tr>
<td>Refinement Proof: the rest</td>
<td>1 pm</td>
</tr>
<tr>
<td>C verification</td>
<td>2.5 pm</td>
</tr>
<tr>
<td><strong>Verification of proc</strong></td>
<td></td>
</tr>
<tr>
<td>Design: 14 layers</td>
<td>1 pm</td>
</tr>
<tr>
<td>Refinement Proof</td>
<td>0.5 pm</td>
</tr>
<tr>
<td>C Verification</td>
<td>1 pm</td>
</tr>
<tr>
<td><strong>Verification of virt</strong></td>
<td></td>
</tr>
<tr>
<td>Design: 9 layers</td>
<td>0.6 pm</td>
</tr>
<tr>
<td>Refinement Proof</td>
<td>0.4 pm</td>
</tr>
<tr>
<td>C Verification</td>
<td>0.3 pm</td>
</tr>
<tr>
<td><strong>Verification of trap</strong></td>
<td></td>
</tr>
<tr>
<td>Design: 3 layers</td>
<td>0.2 pm</td>
</tr>
<tr>
<td>Refinement Proof</td>
<td>0.1 pm</td>
</tr>
<tr>
<td>C Verification</td>
<td>0.1 pm</td>
</tr>
</tbody>
</table>

**Total: 9.9 pm + VCG Dev: 1.5 pm**