Deep Specifications and Certified Abstraction Layers

Ronghui Gu  Jérémie Koenig  Tahina Ramananandro  Zhong Shao
Newman Wu  Shu-Chun Weng  Haozhong Zhang  Yu Guo

Yale University

May 4th, 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/

Essential Software and Management Tools Needed to Build a Powerful, Flexible, and Highly Available Supercomputer.

HPC Programming Tools

Middleware Applications and Management

Operating Systems

<table>
<thead>
<tr>
<th>Resource Management / Job Scheduling</th>
<th>SLURM</th>
<th>Grid Engine</th>
<th>MOAB</th>
<th>Altair PBS Pro</th>
<th>IBM Platform LSF</th>
<th>Torque/Maui</th>
</tr>
</thead>
<tbody>
<tr>
<td>File System</td>
<td>NFS</td>
<td>(ext3, ext4, XFS)</td>
<td>PanFS</td>
<td>Lustre</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Provisioning</td>
<td>Cray® Advanced Cluster Engine (ACE) management software</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cluster Monitoring</td>
<td>Cray ACE (ISCB and OpenIPMI)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Power Mgmt</td>
<td>Cray ACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Remote Console Mgmt</td>
<td>Cray ACE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| Operating System                    | Linux (Red Hat, CentOS, SUSE) |

Performance Monitoring | HPCC | Perfctr | IOR | PAPI/IPM | netperf |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Development Tools</td>
<td>Cray® Compiler Environment (CCE)</td>
<td>Intel® Cluster Studio</td>
<td>PGI (PGI CDK)</td>
<td>GNU</td>
<td></td>
</tr>
<tr>
<td>Application Libraries</td>
<td>Cray® LibSci, LibSci_ACC</td>
<td>MVAPICH2</td>
<td>OpenMPI</td>
<td>Intel® MPI (Cluster Studio)</td>
<td></td>
</tr>
</tbody>
</table>
Motivation

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

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

- 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:

Something magical going on … What is it?
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, take less than one-person-year to develop, and can boot Linux as a guest
What is an Abstraction Layer?

What does a layer \((L_1, M, L_2)\) do?

overlay \(L_2\)

primitives

abs-state

memory

C or Asm module implementation

underlay \(L_1\)

primitives

abs-state

memory
Example: Thread Queues

$L_3$

\[
1 :: 0 :: 2 :: \text{nil}
\]

$tcbp(0)$

\[
\begin{array}{c|c|c|c}
\text{Ready} & 1 & 2 & \text{Ready} \\
\end{array}
\]

$tcbp(1)$

\[
\begin{array}{c|c|c|c}
\text{Ready} & 0 & & \text{Ready} \\
\end{array}
\]

$tcbp(2)$

\[
\begin{array}{c|c|c|c}
\text{Ready} & 0 & & \text{Ready} \\
\end{array}
\]

$tcbp[0]$

$tcbp[1]$ head

$tcbp[2]$ tail

$L_1$

$tcbp[0]$ head

$tcbp[1]$ head

$tcbp[2]$ tail
Example: Dequeue

$L_3$  
1 :: 0 :: 2 :: nil

$tcbp(0)$

$tcbp(1)$

$tcbp(2)$

$L_2$

Ready 1 2

Ready 0

Ready 0

$tcbp[0]$  
$tcbp[1]$  
$tcbp[2]$
The Simulation Relation

well-formed judgment

$L_1 \vdash_R M : L_2$
Forward Simulation:

- Whenever $L_2(f)$ takes $\text{abs1}$ to $\text{abs2}$ in one step, and $R(\text{abs1}, \text{mem1})$ holds,
- then there exists $\text{mem2}$ such that $\llbracket M \rrbracket(L_1)(f)$ takes $\text{mem1}$ to $\text{mem2}$ in zero or more steps, and $R(\text{abs2}, \text{mem2})$ also holds.
Reversing the Simulation Relation

\[ L_1 \preceq_R M : L_2 \rightarrow L_2 \preceq_R \llbracket M \rrbracket L_1 \]

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

\[ \llbracket M \rrbracket L_1 \preceq_R L_2 \]

\[ \llbracket M \rrbracket L_1 \simeq_R L_2 \]

\( \llbracket M \rrbracket (L_1) \) and \( L_2 \) are bisimilar!

\( L_2 \) captures everything about running \( M \) over \( L_1 \)
Deep Specification

\[ \text{\texttt{\{ M \}} \ L_1 \sim_R L_2 } \]

\[ \text{\texttt{\{M\}}(L_1) \text{ and } L_2 \text{ are bisimilar!} } \]

\[ L_2 \text{ captures } \text{everything} \text{ about running } M \text{ over } L_1 \]

Making it “contextual” using the whole-program semantics \[\bullet\]

\[ \text{\texttt{\{ M \}} \ L_1 \sim_R L_2 } \]

\[ L_2 \text{ is a deep specification of } M \text{ over } L_1 \]

\[ \text{if under any valid program context } P \text{ of } L_2, \]

\[ \text{\texttt{\{ P } \oplus M \text{\}}(L_1) \text{ and } \texttt{\{ P \}}(L_2) \text{ are observationally equivalent} } \]
Why Deep Spec is Really Cool?

Deep spec $L$ captures all about a module $M$

- 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.

$L_2$ 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.
Problem w. Shallow Specs

C or Asm module

C & Asm Module Implementation

shallow spec A

C & Asm Modules w. Shallow Spec A

shallow spec B

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

shallow spec

C & Asm Modules w. Shallow Specs

deep spec

C & Asm Modules w. Deep Specs
How to Make Deep Spec Work?

No languages/tools today support deep spec & 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, take less than one-person-year to develop, and can boot Linux as a guest
What We Have Done

- **Coq**
  - Layer Spec $L$

- **Clight**
  - **CompCert**
  - **Asm**

- **Extended Asm Language**
  - **LAsm**

- **LayerLib calculus**
  - **Layered prog. in LAsm**

- **Layered prog. in ClightX**
  - Layered everything together
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, take less than one-person-year to develop, and can boot Linux as a guest
Variants of mCertiKOS Kernels

(base)
- TRAP
- PROC
- THR
- VM
- MM

(hyp)
- TRAP
- VIRT
- PROC
- THR
- VM
- MM

(rz)
- TRAP
- VIRT
- PROC
- THR
- VM
- MM

(emb)
- PROC
- THR
- MM

Variants of mCertiKOS Kernels

MAT
MATOp
MATIntro
PreInit

MPMap
MBit
MPTInit
MPTKern
MPTComm
MPTOp
MPTIntro

PTThread
PSched
PCID
PAbQueue
PTDQInit
PTDQIntro
PTCBInit
PTCBIntro
PKCtxOp
PKCtx

PProc
PUCtx
PIPC
PIPCIntro

VVM
VSVM
VVMCBOp
VSVMIntro
VVMCBInit
VVMCBIntro
VSVMSwitch
VNPTInit
VNPTIntro

TSysCall
TTrap
TTrapArg
Example: Page Fault Handler

TSysCall

TTrap

TTrapArg

PProc

PUCtx

... 

PCID

... 

PMap

MPTOp

MPTIntro

MAT

MATOp

...

PreInit

ikern_set

setcr3

pf_get
Performance

- Normalized macro benchmarks
- Linux on KVM and mCertiKOS, baseline is Linux on bare metal
Performance

- LMbench: Linux on mCertiKOS, Linux on KVM and Linux on bare metal
Performance

- IPC: seL4 on x86 and mCeriKOS
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!
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**, take less than **one-person-year** to develop, and can boot Linux as a guest
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
Programming & Compiling Layers

ClightX

\[ L \vdash_{R} M_{c} : L_{1} \]

\[ L_{1} \leq_{R} \llbracket M_{c} \rrbracket_{\text{ClighX}}(L) \]

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

\[ \llbracket M_{c} \rrbracket_{\text{ClighX}}(L) \leq_{\text{minj}} \llbracket \text{CompCertX}[L](M_{c}) \rrbracket_{\text{LAsm}}(L) \]

\[ L_{1} \leq_{R \circ \text{minj}} \llbracket \text{CompCertX}[L](M_{c}) \rrbracket_{\text{LAsm}}(L) \]

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

\[ L_{1} \leq_{R} \llbracket \text{CompCertX}[L](M_{c}) \rrbracket_{\text{LAsm}}(L) \]

Let \( M_{a} = \text{CompCertX}[L](M_{c}) \) then \( L \vdash_{R} M_{a} : L_{1} \)

LAsm
A Subtlety for LAsm

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

\[
L \vdash_R M_a : L_2
\]

\[
L_2 \leq_R [M_a]_{\text{LAsm}} (L)
\]

**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!
LayerLib: Vertical Composition

\[
\begin{align*}
L_1 \vdash_R M : L_2 \\
L_2 \vdash_S N : L_3 \\
L_1 \vdash_{R \circ S} M \oplus N : L_3
\end{align*}
\]

VCOMP
LayerLib: Horizontal Composition

• $L_1$ and $L_2$ must have the same abstract state
• both layers must follow the same simulation relation $R$
Case Study: mCertiKOS

Single-core version of CertiKOS (developed under DARPA CRASH & HACMS), 5 kloc, can boot Linux

Aggressive use of abstraction over deep specs (37 layers in ClightX and LAsm)

Based on the abstract machine provided by boot loader
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)
Development Cost

<table>
<thead>
<tr>
<th>Development of ClightX and CompCertX</th>
<th>10 pm</th>
</tr>
</thead>
<tbody>
<tr>
<td>Development of VCGen for ClightX</td>
<td>1.5 pm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification of mm</th>
<th>Design: first 3 layers</th>
<th>0.5 pm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Design: the rest 8</td>
<td>0.5 pm</td>
</tr>
<tr>
<td></td>
<td>Refinement Proof: first 2</td>
<td>1.2 pm</td>
</tr>
<tr>
<td></td>
<td>Refinement Proof: the rest</td>
<td>1 pm</td>
</tr>
<tr>
<td></td>
<td>C verification</td>
<td>2.5 pm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification of proc</th>
<th>Design: 14 layers</th>
<th>1 pm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Refinement Proof</td>
<td>0.5 pm</td>
</tr>
<tr>
<td></td>
<td>C Verification</td>
<td>1 pm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification of virt</th>
<th>Design: 9 layers</th>
<th>0.6 pm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Refinement Proof</td>
<td>0.4 pm</td>
</tr>
<tr>
<td></td>
<td>C Verification</td>
<td>0.3 pm</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Verification of trap</th>
<th>Design: 3 layers</th>
<th>0.2 pm</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Refinement Proof</td>
<td>0.1 pm</td>
</tr>
<tr>
<td></td>
<td>C Verification</td>
<td>0.1 pm</td>
</tr>
</tbody>
</table>

Total: 9.9 pm + VCG Dev: 1.5 pm