Decomposing end-to-end refinement proofs across multiple semantics within separation logics

Hai Dang, David Swasey, Gregory Malecha, Gordon Stewart, Abhishek Anand, and others

Iris Workshop
May 24, 2023
BedRock virtualization stack

- Guest OS
- Guest OS
- virtual hardware (CPU, memory, devices, …)
- virtual switch + virtual network cards
- virtual multiplexer for a serial host device
- other controllers of host devices

Legend
- Physical hardware
- BedRock component
- Open-source component
- Untrusted component

- Basic mechanisms for virtualization, scheduling, and separation/management of physical resources
- Higher-level interfaces to manage and share resources (with policies conforming to NOVA’s isolation rules)
To prove: end-to-end refinement

The Bare-metal Property: “any guest behavior possible on the BedRock stack is also possible on the bare-metal”

Physically separate machines with network communications + …

end-to-end: compiled binary code of impl ⊆ the formal spec
Challenge: heterogeneous semantics

Multiple language abstractions
machine <-> ASM <-> C++

Master Controller (Root Task)

NOVA Microkernel

impl

reconfigurable multiple components
each with its own semantics
Decomposing refinement in separation logic

- **Horizontally** across multiple component semantics (in both *impl* and *spec*)
  - linking multiple (modularly developed) separation logics into one

- **Vertically** across multiple language abstractions (compiler correctness + language interoperability)
  - building language abstraction layers with resources

- **All in separation logic?**
  - PRO: avoid intermediate operational semantics, more expressive with resources
  - CONS: complex logic (later, fancy update modality, etc.)
A path towards incremental end-to-end refinement in separation logics

- We develop general frameworks to decompose multiple semantics in both the implementation and the specification.

- We develop a demo setup of (simplified x86) CPUs + memory + I/O devices
  - decompose a refinement proof using the frameworks
  - construct abstractions from machine logics to an ASM logic.

- We outline a path towards incremental end-to-end refinement within separation logic, with sketches on the abstractions from ASM to C++.
VMM

\( \text{init} \implies \) 
\( \text{VMM\_init} \)
\( \text{vSwitch\_init} \)
\( \text{UMX\_init} \)
\( \text{Device\_init} \)

CPP logic

\( \text{impl\_resources} \implies \) 
\( \text{WP\_cpp} \)

\( \text{nov\_safe\_ec} \)
\( \text{nov\_safe\_dev} \)

Userland

Kernel

\( \text{virtual\_tpts\_to} \implies \) 
\( \text{WP\_cpp} \)

\( \text{CPU\_fewer\_regs} \)
\( \text{ASM\_codes} \)
\( \text{CPU\_interrupt} \)

Base logsics

\( \text{refine\_inv(R)} \)

\( \text{impl\_resources} \implies \) 
\( \text{WP\_io} \)

\( \text{big\_impl} \) 
\( \text{big\_spec} \)

\( S_{\text{init}} \)

\( \text{ghost update decomposition} \)
\( \text{C++ app verifications} \)

\( \text{C++ and ASM abstractions} \)

\( \text{Iris} \)

\( \text{adequacy} \)

\( \text{NOVA logic} \)

\( \text{NOVA binary interface} \)

\( \text{Verification of NOVA C++ code + inline ASM} \)

\( \text{Verification of NOVA ASM boot code (setting up page tables ...)} \)

\( \text{C++ abstractions inside NOVA} \)

\( \text{ASM abstractions} \)

\( \text{NOVA} \)

\( \text{input/output decomposition} \)

\( \text{input/output based refinement} \)

\( \text{Iris} \)

\( \text{adequacy} \)

\( \text{Copyright 2023. All Rights Reserved.} \)
big_spec decomposition

program verifications

languages abstractions

big_impl decomposition
An open-world setup for multiple semantics:
Hardware components (in both impl and spec) are modeled as processes communicating through request/response (output/input) events

- Events allows for flexible and dynamic communications among components
  - A CPU read or write is a request event that can be responded by a device other than the RAM memory

- All visible events are from physical I/O devices
CPU || Memory || PL011 (I/O Device)

- CPU
  - !read, !write
  - ?read, ?write

- Memory
  - ?read, ?write

- PL011
  - !send
  - ?receive

- Internal communication

- Visible events (external comm)
LTS Composition ($Cs = \parallel_{\text{LTS}} Cs[i]$)

- **Tau step**
  \[
  Cs[i] \sim{\tau}\rightarrow c'
  \]
  \[
  Cs \sim{\tau}\rightarrow Cs[i := c']
  \]

- **External communication step**
  \[
  Cs[i] \sim{e}\rightarrow c'
  \]
  \[
  e \text{ request/response} \leftrightarrow (\text{Fext } e) \text{ request/response}
  \]
  \[
  Cs \sim{\text{Fext } e}\rightarrow Cs[i := c']
  \]

- **Internal comm step**
  \[
  Cs[i1] \sim{e1}\rightarrow c1'
  \]
  \[
  Cs[i2] \sim{e2}\rightarrow c2'
  \]
  \[
  (e2,e1) \text{ is (request,response)}
  \]
  \[
  Cs \sim{\tau}\rightarrow Cs[i1 := c1'][i2 := c2']
  \]

- Like a threadpool but each thread has its own semantics
- $\tau$ are internal steps
- External communications produce externally visible events
- Internal communications are matching request/response pairs (*atomic* step)
Formally defining the notion of \( \subseteq \) as **trace refinement**

\[
\text{big_impl} \; \text{trace}_R \text{_refines} \; \text{big_spec} := \\
\forall \sigma', TR, \sigma_{\text{init}} \sim \{TR\} \sim^* \sigma' \Rightarrow \\
\exists s', tr, s_{\text{init}} \sim \{tr\} \sim^* s' \land \text{forall}2 \; R \; (\no_{\tau} \; TR) \; (\no_{\tau} \; tr)
\]
Warm up 2: a lightweight refinement setup in Iris

refine_inv(R, TR) :=
∃ s tr, ●s * s_{init} ~{tr}~> s *
Forall2 R (no_τ TR) (no_τ tr)

- instantiate Iris with the big_impl LTS
- encode big_spec as ghost state
- define a refinement inv that relates traces
- prove WP σ while maintaining the refinement inv

\[ \bigcirc_{\text{impl\_resources}} \vdash WP s_{\text{init}} \{ \lambda, \text{False} \} \]

Enters separation logic

Iris adequacy

\[ \text{big_impl \ trace\_R\_refines \ big_spec} \]

∀ σ' TR, σ_{init} ~{TR}~>* σ' ⇒ ∃ s' tr, s_{init} ~{tr}~> s' ∧ Forall2 R (no_τ TR) (no_τ tr)
Decomposing multiple semantics of `big_impl`

**big_impl**

**decomposition**

**Base logics**

<table>
<thead>
<tr>
<th>CPU.regs c1 rs1</th>
<th>CPU.regs c2 rs2</th>
<th>DRAM.bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIO c1</td>
<td>WPIO c2</td>
<td>WPIO DRAM</td>
</tr>
</tbody>
</table>

**Refine inv(R)**

<table>
<thead>
<tr>
<th>CPU.interrupt</th>
<th>PL011.reg</th>
</tr>
</thead>
<tbody>
<tr>
<td>WPIO p1</td>
<td>WPIO ...</td>
</tr>
</tbody>
</table>

**Entering separation logic**

<table>
<thead>
<tr>
<th>Impl_resources</th>
<th>WPIO s_init</th>
</tr>
</thead>
<tbody>
<tr>
<td>Impl_resources</td>
<td>WPIO s_init</td>
</tr>
</tbody>
</table>

**big_impl** `trace_R_refines` **big_spec**

\[
\forall \sigma' \, \text{TR}, \sigma_{\text{init}} \sim \{\text{TR}\} \Rightarrow \exists \ s' \, \text{tr}, \ s_{\text{init}} \sim \{\text{tr}\} \Rightarrow s' \wedge \text{Forall2 R} \ (\text{no}_\tau \text{ TR}) \ (\text{no}_\tau \text{ tr})
\]
Building a logic with multiple semantics

Consider a monolithic logic of a CPU + DRAM + a PL011 (I/O device)

```plaintext
{ cpu.reg r2 v' * cpu.reg r1 pl011.data_reg * pl011.reg data_reg [v] }

MOV r2 [r1]  read from PL011 ([r1]) to r2
{ cpu.reg r2 v * cpu.reg r1 pl011.data_reg * pl011.reg data_reg [ ] }

{ cpu.reg r2 v * cpu.reg r3 l * l ↦ v' }

MOV [r3] r2  write r2 to DRAM ([r3])
{ cpu.reg r2 v * cpu.reg r3 l * l ↦ v }
```
Decomposing a logic for multiple semantics

The monolithic logic considers steps of the whole monolithic machine

\[
\begin{align*}
\{ \text{Pre} \} \quad & \sigma_{\text{CPU+DRAM+PL011}} \xrightarrow{~~>} \sigma'_{\text{CPU+DRAM+PL011}} \\
& \{ \text{Post} \}
\end{align*}
\]

Instead, we want to consider each component’s steps modularly.

\[
\begin{align*}
\{ \cdots \} \quad & \sigma_{\text{CPU}} \xrightarrow{!\text{cpu.Read } l \; v} \sigma'_{\text{CPU}} \quad \{ \cdots \} \\
\{ \cdots \} \quad & \sigma_{\text{CPU}} \xrightarrow{!\text{cpu.Write } l \; v} \sigma'_{\text{CPU}} \quad \{ \cdots \}
\end{align*}
\]

\[
\begin{align*}
\{ \cdots \} \quad & \sigma_{\text{DRAM}} \xrightarrow{?\text{dram.Write } l \; v} \sigma'_{\text{DRAM}} \quad \{ \cdots \}
\end{align*}
\]

\[
\begin{align*}
\{ \cdots \} \quad & \sigma_{\text{PL011}} \xrightarrow{?\text{pl011.DataRead } l \; v} \sigma'_{\text{PL011}} \quad \{ \cdots \}
\end{align*}
\]
WPio captures extra *rely-guarantee* assumptions/obligations one may have when interacting with the environment.

- the requester not only shows that it can make its own step, but also *helps the responder making the matching step, and vice versa*
- request/response conditions capture the atomic spec of the responder, often as AUs

Decomposing multiple semantics with WPio
## Linking logics for multiple semantics with WPio

<table>
<thead>
<tr>
<th>Logics linked through internal comm (matching request/response) events</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>{ cpu.reg r1 l * cpu.reg r2 v' } !cpu.Read l v { cpu.reg r1 l * cpu.reg r2 v }</code></td>
</tr>
<tr>
<td><code>{ cpu.reg r1 l * cpu.reg r2 v } !cpu.Write l v { cpu.reg r1 l * cpu.reg r2 v }</code></td>
</tr>
<tr>
<td><code>{ pl011.reg data_reg [v] } ?pl011.DataRead l v' { v' = v * pl011.reg data_reg [] }</code></td>
</tr>
<tr>
<td><code>{ l ➞ v' } ?dram.Write l v { l ➞ v }</code></td>
</tr>
</tbody>
</table>

Logics more modularly developed

```plaintext
 cpu.reg r1 pl011.data_reg * cpu.reg r2 v' * pl011.reg data_reg [v] ⊢
 cpu.WPio (MOV r2 [r1], !cpu.Read pl011.data_reg v)
 { cpu.reg r1 pl011.data_reg * cpu.reg r2 v * pl011.reg data_reg [] }
```

```plaintext
 cpu.reg r1 l * cpu.reg r2 v * l ➞ v' ⊢
 cpu.WPio (MOV [r1] r2, !cpu.Write l v) { cpu.reg r1 l * cpu.reg r2 v * l ➞ v }
```

```plaintext
 emp ⊢ dram.WPio (?dram.Read l v) { emp }
```
Decomposing multiple semantics of `big_impl`

**Theorem [WPIO to WP]:** \( WPIO \ \sigma \vdash WP \ \sigma \)

**Linking Theorem [Decomposition of WPIO’s]:**
\[
[*set]c \in Cs, WPIO c \vdash WPIO \left(\parallel_{LTS} Cs\right)
\]

Base logics
- `WPio c1`
- `WPio c2`
- `WPio DRAM`
- `WPio p1`
- `WPio ...`

`impl_resources →* WPIO σinit`

`impl_resources →* WP σinit`

`refine_inv(R)`

Entering separation logic

**Linking Theorem:**
\[
\forall \sigma' TR, \sigma_{init} \sim\{TR\}\sim* \sigma' \Rightarrow \exists s' tr, s_{init} \sim\{tr\} \sim* s' \land \forallall R (\text{no}_\tau TR) (\text{no}_\tau tr)
\]

Copyright 2023. All Rights Reserved.
Refinement and WPio

Theorem [WPio to WP]:
WPio $\sigma \vdash WP \sigma$

WPio request/response conditions for external communication steps of physical I/O devices interact with refinement.

![Diagram showing interaction between PL011, CPU, and DRAM]
big_spec decomposition

program verifications

languages abstractions

big_impl decomposition

Entering separation logic

Master Controller

NOVA binary interface

Verification of NOVA C++ code + inline ASM

Verification of NOVA ASM boot code (setting up page tables ...)

C++ abstractions inside NOVA

ASM abstractions

input/output based refinement

Userland Kernel

virtual_tpts_to ...

virtual_tpts_to ...

virtual_tpts_to ...

virtual_tpts_to ...

impl_resources -> WPio? \sigma_{init}

impl_resources -> WP? \sigma_{init}

big_impl trace_R_refines big_spec
ASM abstractions

- Going from machine logics to ASM logic ("compiler correctness" of assembler)
- Reusing existing logics looks promising [1,2]
- We develop a demo ASM logic without stating an operational semantics for ASM
  - After the WPio decomposition, abstract from WPio of CPU to WP asm
  - Intuition: to get to ASM, give up resources needed for the ASM abstractions
    - instruction fetch and decoding (code stored in memory)
    - jump labels

Building the ASM abstractions

Axiom abstract_asm :
\[ WP_{asm} c1 asmprog \vdash \forall pc q, cpu.pc c1 pc \rightarrow^* dram.bytes l q (assemble asmprog) \rightarrow^* WPio c1. \]

Solution :
Definition \[ WP_{asm} c1 \ prog := \]
\[ \forall pc q prog', cpu.pc c1 pc \rightarrow^* dram.bytes l q (assemble prog') \rightarrow^* \]
\[ \text{prog'} @ pc = \text{prog} \rightarrow^* \cdots \rightarrow^* WPio c1. \]

<table>
<thead>
<tr>
<th>Machine</th>
<th>cpu.pc c1 pc * dram.byte pc q (op_encode MOV [r1] r2) *</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>cpu.reg r1 l * cpu.reg r2 v * l \rightarrow v \vdash</td>
</tr>
<tr>
<td></td>
<td>cpu.WPio c1</td>
</tr>
<tr>
<td></td>
<td>{ cpu.pc c1 pc+1 * dram.byte pc q (op_encode MOV [r1] r2) * \cdots }</td>
</tr>
</tbody>
</table>

| ASM    | cpu.reg r1 l * cpu.reg r2 v * l \rightarrow v' \vdash |
|        | WP_{asm} c1 (MOV [r1] r2) \{ cpu.reg r1 l * cpu.reg r2 v * l \rightarrow v \} |
C++ abstractions!

- **Compiler correctness as separation logic specifications?**
- This would require building multiple layers of abstractions for C++ in the logic
- We need to allow breaking abstractions, consider inline ASM
- Goal: to get to C++, give up resources needed for the C++ abstraction

Axiom abstract_wp_cpp :

\[
\left(\left[\text{list}\right] \text{tpts_to} \ldots \ast WP_{\text{cpp}} (c1, \text{cpp_prog})\right) \vdash \\
\text{stack_frames} \ldots \ast \text{page_tables} \ldots \ast \text{bytes} \ldots \ast \ast WP_{\text{asm}} (c1, \text{asm_prog}).
\]
Sketching a C++ `tpts_to`

`tpts_to Tuchar p (Vint v)`

○ `{p := (Tuchar, Vint v)}`

**C++ Pointer**

- `(m : pointer_map)`

- `[*map] p ↦ (T, v), vbyte (ptr_addr p) T v`

**Virtual Memory**

- `page_tables ...`

- `[*map] ...,
  byte pa encode(T, v)`

- `...`