CertiKOS: From Hacker-Resistant OS to Certified Heterogeneous Systems

Zhong Shao
Yale University
December 8, 2021

Acknowledgement: Ronghui Gu, Newman Wu, Hao Chen, Jeoung Kim, Jeremie Keenig, Vilhelm Sjoberg, Mengli Liu, Man-Ki Yoon, Jung-Eun Kim, Lionel Rieg, Quentin Carbonneaux, Unsung Lee, Ji Yong Shin, David Costanzo, Tahina Ramananandro, Hans Van Zelst, Shu-Chun Weng, Zhencao Zhang, Liang Gu, Jan Hoffmann, and Joshua Lockerman. This research is supported in part by DARPA CRASH and HACMS programs and NSF SaTC and Expeditions in Computing programs.

The Future of CS: Heterogeneous Systems

- Hardware Components
  - Multicore CPU, MMU/OOMMU, Cache, FPGA, GPU, TPU, ASIC, Domain-Specific HW, …
  - Cyber-physical systems, smart cars, robotics, IoTs, smart cities, blockchains, clouds, …

- Human and Social Components
  - Passengers, pedestrians, insurance companies, lawyers, policy-makers, …
  - Banks, consumers, currencies, smart contracts, crypto coins, DAO, federal reserve, …

- OS Components
  - Processes, schedulers, containers, device drivers, virtual machines, file systems, sockets, databases, logs, atomic objects, transactions, network stacks, security protocols, …

- SW & PL Components
  - Data structures; objects; transactions; modules; threads; interrupt handlers; exception; communication channels; concurrent & distributed objects; containers; micro-services; …
  - Language specs, compilers, interpreters, JIT, binary translator; parsers, serializer / pretty printer, …

- Challenge: how to establish strong trust & accountability properties?
  - Safety, security (isolation), resource efficiency, availability, accountability, extensibility, …

PL Research: The Charm

- Uncover the essence (i.e., semantic abstraction) of various (SW & HW) systems, or systems of systems

- Use these new theories to bridge different areas of specialty (e.g., via new lang., compilers, tools) and then build better & more secure/reliable systems

Rich Logical Foundations:
- category theory
- universal algebra & co-algebra
- inductive vs. coinductive types
- classical vs. intuitionistic vs. linear logics
- monad vs. comonad
- equality (HTT) vs simulation
- Curry-Howard correspondence
- lambda & process calculi
- denotational vs operational semantics

New PLs, DSLs, and Compilers:
- static vs dynamic typing vs. “no typing”
- explicit resource manager vs. GC
- effects vs. encapsulation
- OO vs. function programming
- New multiparadigm languages: Java Scala, Swift, Rust, Go, C++, C#
- Concurrent vs. parallel vs distributed programming
- proof assistant languages
- information flow control & security
- certified software & compilers

PL Meets OS: An Ideal Marriage Yet to Happen?

- PL is to uncover the laws of abstraction in the cyber world
- PL is to use abstraction to reduce complexities
- PL depends on the underlying OS for sys lib.
- Many PL issues are easily resolved in OS

- OS is to build layers of abstraction (i.e., VMs) for the cyber world
- OS is full of complexities
- OS is to manage, multiplex, and virtualize resources
- OS really needs PL help to provide safety and security guarantee
The CertiKOS / DeepSpec Project

**Killer-app:** high-assurance “heterogeneous” systems of systems!

**Conjecture:** today’s PLs fail because they ignored OS, and today’s OSes fail because they get little help from PLs

**New Insights:**
- deepspec & certified abstraction layers;
- a unifying framework for composing heterogeneous components (via game semantics + linear logic connectives)

**Opportunities:**
- New certified system software stacks (CertiKOS ++)  
- New certifying programming languages (DeepSEA vs. C & Asm)  
- New certified programming tools  
- New certified modeling & arch. description lang. (DeepSEA)  
- We verify all interesting properties (correctness, safety, security, availability, …)

CertiKOS Problem Definition

- **What is a certified OS?**
  - an OS binary implements its specification?
  - what should its specification be like?

- **What properties do we want to prove?**
  - safety & partial correctness properties  
  - total functional correctness  
  - security properties (isolation, confidentiality, integrity, availability)  
  - resource usage properties (stack overflow, real time properties)  
  - race-freedom, atomicity, and linearizability  
  - liveness properties (deadlock-freedom, starvation freedom)

- **How to cut down the cost of verification?**

Motivation

Formal Verification

- mathematically prove  
- program meets specification  
- under all inputs  
- under all execution  
- rule out entire classes of attacks

Formal verification is the only way to give users confidence in the system.

— NSF SFM Report [2016]

Challenges: huge proof efforts

- seL4 [SOSP’09]
  - Proof 11 py
  - C 7.5k LOC
  - Asm 500 LOC unverified
  - C 1.3k LOC unverified
A Complex System

Challenges: Compositionality

Complete Verification

Challenges: Concurrency

I/O concurrency

multi-thread

multiprocessor

Challenges: Concurrency

CPU i

Complete Verification

CPU j
Contribution

Certified Abstraction Layers

CertiKOS

aim to solve all these challenges

Certified Abstraction Layers

verify existing systems

build the next generation heterogeneous systems designed to be reliable and secure

Certified Abstraction Layers

untangle

fine-grained lock

CPU i

CPU j

Certified Abstraction Layers

verify existing systems

build certified heterogeneous systems
Contribution

Certified Abstraction Layers

mCertiKOS [POPL'15]
certified sequential OS kernels
3k C&Asm, 1 py

Interrupt [PLDI'16a] 0.5 py

Security [PLDI'16b] 0.5 py

mC2 [OSDI'16] [PLDI'18]
the first formally certified concurrent
OS kernel with fine-grained locks
6.5k C&Asm, 2 py
Contribution

functional correctness
liveness
no stack/integer/buffer overflow
no race condition

Certified System Software

mC2

mC2

Coq

6.1k LOC
C layers
CompCertX
400 LOC
Asm layers
machine-checkable proof

Build a Certified System

User Application
Inter-Process Communication
Scheduling Module
Thread Queue Module
Spin-lock Module
Keyboard Driver

Compiler

Send

CPU 0

Keyboard

CPU 1
Certified Sequential Layer [POPL'15]

- **certified objects**

- specification of modules to trust

Certified Sequential Layer [POPL'15]

- **abs-state**

- primitives

Certified Sequential Layer [POPL'15]

- **certified objects**

- specification of modules to trust

Certified Sequential Layer

- module $M$

- memory $L_1$
Example: Thread Queue

typedef struct tcb {
    state s;
    tcb *prev, *next;
} tcb;

tcb tcbp[1024];

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

tdq* td_queue;

Example: Thread Queue

typedef struct tcb {
    state s;
    tcb *prev, *next;
} tcb;
tcb tcbp[1024];

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

Example: Thread Queue

tcb* dequeue(tdq* q) {
tcb *head, *next;
tcb *i = null;
if (!q) return i;
head = q -> head;
if (!head) return i;
i = head;
next = i -> next;
if (!next) {
    q -> head = null;
    q -> tail = null;
} else {
    next -> prev = null;
    q -> head = next;
}
return i;
}
Example: Thread Queue

```c
void dequeue(tdq* q) {
    tcb *head, *next;
    tcb *i = null;
    if (!q) return i;
    head = q -> head;
    if (!head) return i;
    i = head;
    next = i -> next;
    if (!next) {
        q -> head = null;
        q -> tail = null;
    } else {
        next -> prev = null;
        q -> head = next;
    }
    return i;
}
```

specification

```
Definition tcbp := ZMap.t state.
Definition td_queue := List Z.
```
**Example: Thread Queue**

### Specification

- Definition `tcbp := ZMap.t state.
- Definition `td_queue := List Z.

### Implementation

- Function `dequeue (q) := match q with
  - head :: q' => (q', Some head)
  - nil => (nil, None)
end.

###Executable

- Function `dequeue (q) := match q with
  - head :: q' => (q', Some head)
  - nil => (nil, None)
end.
Deep spec $L_2$ captures all we need to know about $M$ over $L_1$. Any property about $M$ can be proved using $L_2$ alone. No need to look at $M$ again.
mCertiKOS

memory management

seq machine

mCertiKOS

certified sequential kernel

seq machine

mCertiKOS

kernel

Trap
PM
TM
MM

mCertiKOS

kernel

Trap
PM
TM
MM

mCertiKOS

kernel

Trap
PM
TM
MM

mCertiKOS

kernel

Trap
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM

mCertiKOS

kernel

Trap
VM
PM
TM
MM
mCertiKOS 3k LOC [POPL’15] 1 person year
Can boot Linux as a guest

Concurrent Framework [OSDI’16, PLDI’18]
certified sequential kernel

multicore machine CPU0 CPU1 CPU2 CPU3
Contribution

In this paper, we present CCAL—a fully compositional semantic model for lock-based concurrent systems. It must also support vertical composition [2] of these concurrent layers. A general "parallel layer composition rule" can handle concurrent objects. Over Gu et al. [16], this paper presents a set of building blocks for specifying and verifying concurrent abstraction layers. As shown in Fig.1, CCAL consists of a novel compositional semantic model [30] (for releasing locks). This allows us to give specifications for lock primitives and support vertical composition libraries such as queuing locks, condition variables (CV), message-passing primitives [2].

Case Study

struct ticket_lock {
  volatile uint n, t;
};
//Methods provided by L₀
extern void acq();
extern void rel();
extern cpu_id();

//Methods provided by L₁
extern void acq();
extern void rel();
extern cpu_id();

//Methods provided by L₂
extern void acq();
extern void update_x();
//Client program P
//Thread running on CPU 1
void T1 () { update_x(); }
//Thread running on CPU 2
void T2 () { update_x(); }
strategies and game semantics

strategy \( \psi_p[i] \)

How will the program \( p \) generate events on behalf of CPU \( i \) at each step regarding the given logical log \( \mathcal{L} \)?
Strategies and Game Semantics

$$\psi_{\text{FAI}_t}[1]$$

?l, !1.\text{FAI}_t, $t$

logical

log $l$

2.\text{FAI}_t 1.\text{FAI}_t 2.\text{FAI}_t 1.\text{FAI}_t$

$3$

void acq () {
uint my_t = \text{FAI}_t();
while(get_n() != my_t) {
hold();
}
}

Strategies and Game Semantics

extern uint \text{FAI}_t();
extern uint get_n();
extern void inc_n();
extern void hold();
Given the current $\log \ l$, how the module $\text{Macq}$ running over $\text{Lo[l]}$ will generate events on behalf of CPU $i$ at each step.
//Methods provided by L0
extern uint get_n();
extern void inc_n();
extern uint FAI_t();
extern void hold();

//M1 module
void acq () {
    uint my_t = FAI_t();
    while(get_n() != my_t){};
    hold();
}
void rel () { inc_n() ; }

//M2 module
int x = 0; //shared variable x
void update_x() {
    acq(); x += cpu_id(); rel();
}

//Methods provided by L2
void T1 () { update_x(); }
void T2 () { update_x(); }

//Client program P
//Thread running on CPU 1
void T1 () { update_x(); }
//Thread running on CPU 2
void T2 () { update_x(); }

//Methods provided by L0
extern uint get_n();
extern void inc_n();
extern uint FAI_t();
extern void hold();

//M1 module
void acq () {
    uint my_t = FAI_t();
    while(get_n() != my_t){};
    hold();
}
void rel () { inc_n() ; }

//M2 module
int x = 0; //shared variable x
void update_x() {
    acq(); x += cpu_id(); rel();
}

//Methods provided by L2
void T1 () { update_x(); }
void T2 () { update_x(); }

//Client program P
//Thread running on CPU 1
void T1 () { update_x(); }
//Thread running on CPU 2
void T2 () { update_x(); }

logical log l
2.FAI_t 2.FAI_t
Strategies and Game Semantics

\[ (P \oplus M_1 \oplus M_2) L_0 \]

?l, !1.FAI_t, $t

logical

2.FAI_t 2.FAI_t 1.FAI_t

$2

//Methods provided by L_0
extern uint get_n();
extern void inc_n();
extern uint FAI_t();
extern void hold();

//M_1 module
void acq () {
  uint my_t = FAI_t();
  while(get_n()!=my_t){}
  hold();
}

void rel () { inc_n() ; }

//Methods provided by L_2
//Client program P
//Thread running on CPU 1
void T1 () { update_x(); }
//Thread running on CPU 2
void T2 () { update_x(); }

C

//Methods provided by L_0
extern uint get_n();
extern void inc_n();
extern uint FAI_t();
extern void hold();

//M_1 module
void acq () {
  uint my_t = FAI_t();
  while(get_n()!=my_t){}
  hold();
}

void rel () { inc_n() ; }

//Methods provided by L_2
//Client program P
//Thread running on CPU 1
void T1 () { update_x(); }
//Thread running on CPU 2
void T2 () { update_x(); }

C
Strategies and Game Semantics

$E_{hs} - 1 2 2 1 1 1 2 1 2 1 1 2 1 2 2 2 2 2$

logical log \( l \)

Strategies and Game Semantics

$E_{hs} - 1 2 2 1 1 2 1 2 1 2 1 1 2 1 2 2 2 2 2$

logical log \( l \)
Strategies and Game Semantics

\[ \mathcal{E}_{hs} \]

\[ \begin{array}{cccccccc}
\mathcal{E}_{hs} & 1 & 2 & 2 & 1 & 1 & 2 & 1 & 2 \\
?l, !1.get_n, \$n=t & ?l, !1.get_n, \$m=t & \mathcal{L}_0[1] & \mathcal{L}_0[2] & \mathcal{L}_0[1] & \mathcal{L}_0[2]
\end{array} \]

Set of logical logs

\[ \begin{array}{cccccccc}
\end{array} \]
Strategies and Game Semantics

\[ \mathcal{E}_{hs} = \begin{cases} 1 & \vdash 2 & 1 & \vdash 1 & \vdash 2 & 1 & \vdash 2 & 1 & \vdash 2 & 1 & \vdash 2 & 1 & \vdash 2 & 2 & 2 & \end{cases} \]

\[ \vdash \begin{cases} 1.x+1 & \vdash 2.x+2 \end{cases} \]

\[ x = 3 \]

Environment Context \( \mathcal{E} \in \mathcal{R} \)
$\psi(E, l_0)$

$\psi(E, l_0) \leq_R \psi'$

$E \in \mathcal{R}$

$\psi'(E', l'_0)$

Add fuel (f) to prove liveness
\[ \leq_R \]
\[ \psi_{acq}[i] \]
\[ \mathcal{R}_{j \neq i} : \text{will release lock within } k \text{ steps} \]

\[ \mathcal{R}_m : \text{(fairness) each CPU will be rescheduled within } m \text{ steps} \]

\[ \mathcal{R}_{cpu} : \#CPU = c \text{ is bounded} \]

mutual exclusion?
Strategy Refinement

\[ \leq_R \]

\[ \psi_{acq[i]} \]

mutual exclusion?

\[ \psi_{rel[i]} \]

Certified Concurrent Abstraction Layer

\[ \leq_R \]

\[ \psi_{acq[i]} \]

Certified Concurrent Abstraction Layer

\[ \leq_R \]

\[ \psi_{rel[i]} \]

Horizontal Composition
void update_x () {
    acq();
    x += cpu_id();
    rel();
}

\begin{align*}
\text{L}_1[i] & \leq R' \quad \text{L}_2[i]
\end{align*}
**Vertical Composition**

\[ L_2[i] \]
\[ R' \]
\[ M_2 \]
\[ L_1[i] \]  
\[ \oplus \]
\[ R \]
\[ M_{\text{acq}} \oplus M_{\text{rel}} \]
\[ L_0[i] \]

**Parallel Composition**

\[ R_{j \neq i} : \text{will release lock within } k \text{ steps} \]

\[ R_{hs} : \text{(fairness) each CPU will be rescheduled within } m \text{ steps} \]

\[ R_{cpu} : \#CPU = c < 2^{32} \]

\[ L_2[i] \]
\[ R \circ R' \]
\[ M_2 \]
\[ M_{\text{acq}} \oplus M_{\text{rel}} \]
\[ L_0[i] \]

**Vertical Composition**

\[ L_2[i] \]
\[ R \circ R' \]
\[ M_2 \]
\[ L_1[i] \]
\[ \oplus \]
\[ R \]
\[ M_{\text{acq}} \oplus M_{\text{rel}} \]
\[ L_0[i] \]  

**Parallel Composition**

\[ R_{j \neq i} : \text{will release lock within } k \text{ steps} \]

\[ R_{hs} : \text{(fairness) each CPU will be rescheduled within } m \text{ steps} \]

\[ R_{cpu} : \#CPU = c < 2^{32} \]

\[ L_2[i, j] \]
\[ R \circ R' \]
\[ M_2 \]
\[ M_{\text{acq}} \oplus M_{\text{rel}} \]
\[ L_0[i, j] \]
Parallel Composition

$$\mathcal{R}_{hs} : \text{(fairness)} \text{each CPU will be rescheduled within m steps}$$

$$\mathcal{R}_{cpu} : \# \text{CPU} = c < 2^{32}$$

Case Study

$$\mathcal{E}_{hs}$$

$$\mathcal{L}_0[i, j]$$

$$\mathcal{R} \circ \mathcal{R}$$

$$\mathcal{L}_2[i, j]$$

$$\mathcal{R} \circ \mathcal{R}$$

$$\mathcal{M}_2$$

$$\mathcal{R} \circ \mathcal{R}$$

$$\mathcal{M}_{acq} \oplus \mathcal{M}_{rel}$$

$$\mathcal{L}_0[i, j]$$

Case Study

$$\mathcal{E}_{hs} - \begin{array}{cccccccc}
1 & 2 & 2 & 1 & 1 & 2 & 1 & 2
\end{array}$$
Soundness

\[ R_{hs} : \text{(fairness) each CPU will be rescheduled within } m \text{ steps} \]

\[ R_{cpu} : \#CPU = c < 2^{32} \]

\[ \{ P \oplus M1 \oplus M2 \} L0[1,2] \quad \xrightarrow{R \circ R'} \quad \{ P \} L2[1,2] \]

\[ = \{ 2.x\leftarrow 2, 1.x\leftarrow 1 \}, \{ 1.x\leftarrow 1, 2.x\leftarrow 2 \} \}\]

QED

Assembly Layers

Horizontal Composition
Vertical Composition
Parallel Composition

Software Scheduler

```c
void yield () {
    uint t = tid();
    ...
    enq (t, rdq());
    uint s = deq (rdq());
    ...
    context_switch (t, s)
}
```

Shared Queue Lib

Spinlock
Software Scheduler

sleep  yield  wakeup

Shared Queue Lib

Spinlock

Thread1

f1

yield

Thread2

g1  g2  yield

T1's view

T1  T1  T1

T2's view

T2  T2  T2

How to compose?
### Contribution Summary

#### Certified Concurrent Abstraction Layers

- CertiKOS
- CompCertX
- Inter-Process Communication
- Scheduling Module
- Thread Queue Module
- Spin-lock Module

#### CertiKOS

- Layer
- Reification proof
- Code verification
- Source code
- Proof linking

<table>
<thead>
<tr>
<th>Layer</th>
<th>Reification proof</th>
<th>Code verification</th>
<th>Source code</th>
<th>Proof linking</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Strategy Refinement

\[ \psi \leq_R \psi' \]

#### Thread-safe CompCert

<table>
<thead>
<tr>
<th>Intermediate layer interface for multi-threaded linking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layer</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>RThread</td>
</tr>
</tbody>
</table>

#### Case Study

**Build a Certified System**

- **Compiler**
- **User Application**
- **Inter-Process Communication**
- **Scheduling Module**
- **Thread Queue Module**
- **Spin-lock Module**
- **CPU 0**
- **Keyboard Driver**
- **CPU 1**
- **Security**
Case Study

Build a Certified System

Device Driver [PLDI16'a]

Build a Certified System

Device Driver [PLDI16'a]
**End-to-End Security [PLDI16'b]**

- **Observation function $O$**
  - specify and prove general security policies with declassification
  - security-preservation simulation
  - non-interference
  - found security-bugs: spawn, palloc,…

**Summary: The CertiKOS / DeepSpec Project**

- **Killer-app**: high-assurance “heterogeneous” systems of systems!
- **Conjecture**: today’s PLs fail because they ignored OS, and today’s OSes fail because they get little help from PLs

- **New Insights**:
  - deepspec & certified abstraction layers;
  - a unifying framework for composing heterogeneous components (via game semantics + linear logic connectives)

- **Opportunities**:
  - New certified system software stacks (CertiKOS ++)
  - New certifying programming languages (DeepSEA vs. C & Asm)
  - New certified programming tools
  - New certified modeling & arch. description lang. (DeepSEA)
  - We verify all interesting properties (correctness, safety, security, availability, …)