A certified page allocator in uCore

The first automatedly verified page allocator in the world

Xingyu Xie

Tsinghua University

June 15, 2020

seL4: the beginning of the story

KIT kernel: the pioneer

- published in 1989
- about 300 lines of machine code
- verified by Boyer-Moore theorem prover

The evolution of microkernels:

- Mach (1st): hot in 1980s, designed for computers in networks
- L4 (2nd): late 1990s, designed for high performance
- seL (3rd): commenced in 2006 by NICTA, the first-ever general-purpose operating-system kernel

The methodologies of microkernels

The design paradigm of microkernels:

A concept is inside the microkernel only if it can't be outside.

The mechanisms provided (L4):

- address space (page table and memory protection)
- threads and scheduling
- inter-process communication

Three methodologies of system verification

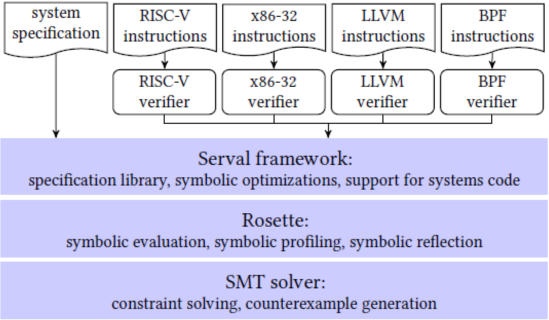
Methodology	Prover	Verified software
Interactive	Coq, Isabelle/HOL	seL4, CertiKOS, ...
Auto-active	Dafny	Verve, Komodo, ...
Push-button	Serval	Yggdrasil, Hyperkernel, ...

表 1: Three Methodologies in software verification

1. Background: verified operating systems
2. Serval: an automated verification toolchain
3. Page Allocator: our experience of verification
4. Summary

Overview of Serval

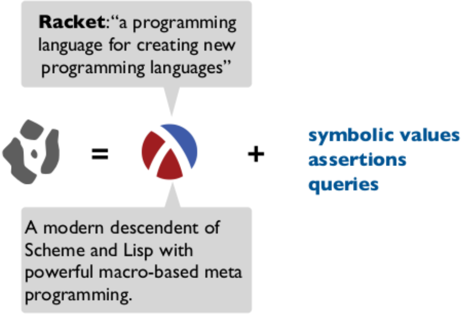
The most lightweight operating system verification framework nowadays[1].



[1]SOSP'19 Luke Nelson et al. Scaling symbolic evaluation for automated verification of systems code with Serval.

Rosette: Introduction

Rosette[2] is a solver-aided programming language in Racket family.



Basic principles: symbolic execution and bounded model checking

[2]PLDI'14 Emina Torlak and Rastislav Bodik. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages.

Rosette: Features

- Solvable type: booleans, integers, reals, bitvector, uninterpreted functions...
- Symbolic-aided queries: symbolic constant, assertion, verification, angelic execution
- Advanced queries: debug, synthesis
- Debugger and profiler[3]

[3] OOPSLA'18 James Bornholt and Emina Torlak. Finding Code That Explodes Under Symbolic Evaluation.

Verification of operating systems by Serval

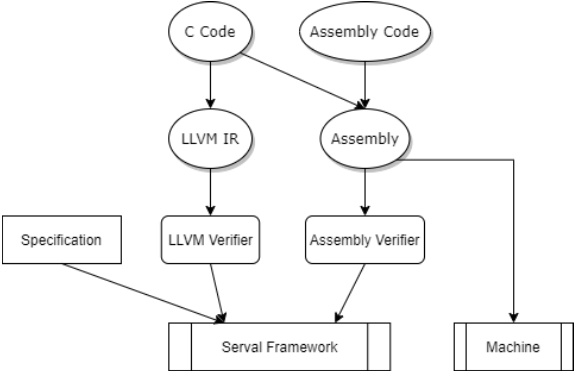


图 1: The framework from the view of operating system

State-machine refinement

$$RI(c) \Rightarrow RI(f_{impl}(c))$$
$$(RI(c) \land AF(c) = s) \Rightarrow AF(f_{impl}(c)) = f_{spec}(s)$$

where RI is representation invariant, AF is abstraction function, f is a function (or to say, a state transition), c is a concrete state, s is a specifcation state.

After refinement, we could prove safety properties further.

Lightweigt

"Four person-weeks to verify an operating system."

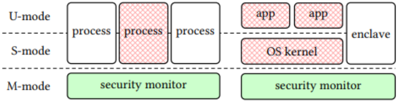
- LoC of Serval framework: 1,244
- LoC of RISC-V verifier: 1,036
- LoC of LLVM verifier: 789
- LoC of CertiKOS: 1,988 + 859
- LoC of Komodo: 2,310 + 1,462

Limitations

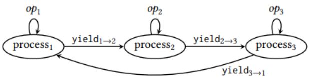
The limitations of this toolchain:

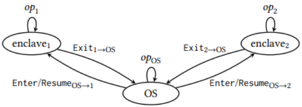
- all loops and recursions must be statically bounded
- the pointer, which is common in operating systems, cannot be supported at all
- the time cost of symbolic execution explodes exponentially sometimes
- bugs in the toolchain
- others that you'll meet only when verifying

Retrofitting



(a) CertiKOS[s]   (b) Komodo[s]





CertiKOS[4]: spawn of process and quota, scheduled in a list.

Komodo[5]: SGX-like enclave mechanism, complete page table

[4] OSDI'16 Ronghui Gu et al. CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels.

[5] SOSP'17 Andrew Ferraiuolo et al. Komodo: Using verification to disentangle secure-enclave hardware from software.

uCore: an education-purposed operating systems

- uCore: for course *Operating Systems*
- ISA: x86 32, RISC-V 32, RISC-V 64
- LoC: about 13k (RISC-V 64 step-by-step version)

Page Allocator

Page Allocator: to allocate the physical pages to threads.

```
// APIs of our page allocator
void init() { ... }
void init_memmap(size_t base, size_t num) { ... }
size_t alloc_pages(size_t num) { ... }
void free_pages(size_t base, size_t num) { ... }
void nr_free_pages() { ... }
```

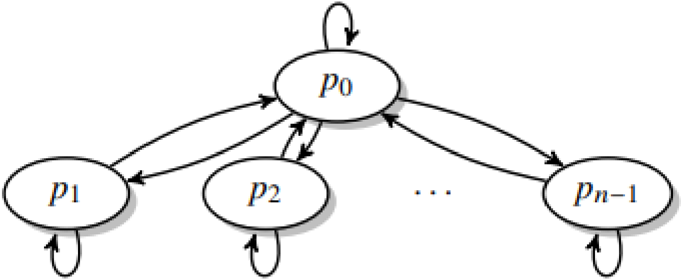
Page allocator is not accepted by most microkernels as it's complicated, the verification of page allocator is really challengable.

We've verified two safety properties of page allocator:

- `nr_free` = [the number of available pages]
- noninterference

Noninterference: Intuitional Concept

Noninterference is a strict multilevel safety property, first described by Goguen and Meseguer in 1982.



Noninterference: Formal Definition

We use a formal definition simplified from Nickle[6].

$$\forall tr' \in \mathrm{purge}(tr, \mathrm{dom}(a)).$$
$$\mathrm{output}(\mathrm{run}(s, tr), a) = \mathrm{output}(\mathrm{run}(s, tr'), a)$$

Explanations

- a: action (init, alloc, free, ...)
- tr: trace (sequence of actions)
- $\mathrm{dom}(a)$: the pages that a is supposed to influence
- $\mathrm{purge}(tr, P)$: all possible traces that is produced by deleting actions a, where $\mathrm{dom}(a) \cap P = \emptyset$, from tr.

Flaw: too general and expressive to prove automatedly.

[6] OSDI'18 Helgi Sigurbjarnarson et al. Nickel: A Framework for Design and Verification of Information Flow Control Systems

Unwinding Conditions

unwinding: the states of p in s and t are the same, notated as $s \overset{p}{\approx} t$, where s and t are two specification states and p is a page.

- unwinding is an equivalence relation
  - reflexivity: $s \overset{p}{\approx} s$
  - symmetry: $s \overset{p}{\approx} t \Rightarrow t \overset{p}{\approx} s$
  - transitivity: $r \overset{p}{\approx} s \wedge s \overset{p}{\approx} t \Rightarrow r \overset{p}{\approx} t$
- local respect: $p \notin \mathrm{dom}(a) \Rightarrow s \overset{p}{\approx} \mathrm{step}(s, a)$
- weak step consistency:
  $s \overset{p}{\approx} t \wedge (\forall q \in \mathrm{dom}(a). s \overset{q}{\approx} t) \Rightarrow \mathrm{step}(s, a) \overset{p}{\approx} \mathrm{step}(t, a)$

Summary

Finally, we firstly automatedly verify a page allocator (but still not complete yet).

- What to verify? MicroKernel (page allocator)
- How to verify? Serval / Rosette / Z3 (symbolic execution)
- What property to verify? Noninterference

Future Plans

Broaden the ability of this toolchain: symbolic execution and bounded model checking still *not* enough at all for many realistic scenarios.

- loop
- pointer

Find more to verify by this lightweight toolchain

- implementation: network? contract? some other model?
- safety property: need to learn the knowledge of some specific domain, e.g. crash safe of file system

Self-reflection: how to debug an symbolic executor?

Thanks!