Hyperkernel: Push-Button Verification of an OS Kernel

### 2020310795 康鸿博

## What are the problems mentioned by the paper? (intro)

OS kernel is one of the most critical components of a computer system, and bugs in the OS kernel can be highly destructive, while also hard to find. Previous works have tried to build verified OS kernel, but their verifications introduce non-trivial cost.

This paper builds an OS kernel with a push-button verification.

## Summary of major innovations (intro)

To build a verified OS kernel, several key challenges must be solved. The main innovation of this paper is to give ideas on how to deal with them.

1. Hyperkernel carefully design the kernel interface that can both support complicated semantics and support scalable verification.
2. Hyperkernel use a separate kernel space, and use identity mapping for it. It’s done by X86 virtualization support provided by Intel VT-x and AMD-V. Virtual-to-Physical memory mapping brings complexities to verifications, because they are set by system calls on the one hand, and they change the behaviors of system calls on the other hand.
3. Hyperkernel is written in C, and verified in LLVM IR. Using LLVM IR avoids ambiguity in C language, while also avoiding machine specific instructions.
4. Interrupt is disabled during the execution of trap handlers to avoid nested trapping.

## How about the important related works/papers? (related work)

1. Verified OS kernels: seL4 is a famous verified OS kernel. Hyperkernel is closer to Unix-like kernel, and supports an automated verification.
2. Concurrent OS kernel: CertiKOS verifies concurrent OS kernel using Coq interactive theorem prover.
3. Co-designing system with proof automation: Yggdrasil does this to a file system, where it models crashes.
4. OS design: This paper borrow idea from Dune, where each process is virtualized and has control over hardware, and Hyperkernel lies in the Hypervisor mode with a different address mapping.
5. LLVM verification: used by several projects like Vellvm, Alive, etc.. Hyperkernel chooses to have finite loops/recursions and unrolls them to transform them to SMT for verification.

## What are some intriguing aspects of the paper? (design & implementation)

###### Proof Structure:

Hyperkernel use two types of semantic descriptions to verify its implementation: state-machine specifications for trap handlers, and declarative specifications that should be satisfied after any trap handler. Both are written in Python.

To use them to verify, the implementation of Hyperkernel is first compiled into LLVM IR, then transformed into Z3 SMT. Z3 solver will try to give counterexamples where the SMT generated by Python specifications and LLVM IR produce different results. If it succeeds, it gives an example to a kernel bug, else if proves the correctness of kernel code. Declarative specifications are assumed correct before each trap handler, and are checked after each.

###### Finite Interfaces (for Efficient Verification):

It’s non-trivial to design an interface that support efficient verification. To do this, Hyperkernel designed trap handlers that have only finite loops and recursion.

One example in the paper is the “dup(oldID)” system call, which creates a copy of a file descriptor with the smallest possible id. An infinite loop is used here to search for the smallest id. Hyperkernel changes it to “dup(oldID, newID)” to only check if “newID” is occupied to avoid this unbounded loop.

###### Avoiding Redundant Check:

An interesting advantage of verified kernel is to avoid redundant check. Kernel may use redundant check to make itself more robust towards unknown bugs. However, if the kernel is verified beforehand, these checks can be removed to improve performance.

###### Dealing with DMA and User Space Inputs:

Although Hyperkernel assumes a single core machine, DMA can still run concurrently with CPU. Hyperkernel deals with this by confining DMA updates into a DMA region, and treating this region as a “volatile” region with the possibility to hold any key at any time. User space inputs (like parameters of system calls) as checked in similar way.

###### Reasoning LLVM IR:

To verify the LLVM IR, besides showing that it produces the same result as the Python specification, Hyperkernel checks for all undefined behaviors. It’s done by adding side checks to avoid undefined behavior, assuming any value for undefined value, and avoiding poison values.

###### Efficient coding of declarative specifications:

To support efficient reasoning, Hyperkernel only use first-order logic. There are two important patterns whose specification isn’t naturally first-order logic: object ownership and reference counter. The paper show ways to convert them into first-order logic: build the “reverse map” for the first-order logic, and use a permutation to move references to the first “r” positions for reference counter with value “r”.

## How to test/compare/analyze the results? (experiment)

As for bug prevention, both types of specification (state-machine specification and declarative specification) help prevent bugs in the kernel implementation. Several interesting bugs are detected by the declarative verification on reference counting, showing the importance of declarative specifications.

Verification performance are fast and stable using different versions of Z3 solver. One pass takes 10-25 minutes on an eight-core Intel CPU. The main hinderance of run-time performance is the cost of “hypercall” since Hyperkernel runs in the “virtualized mode”. However, this also brings advantages: system calls can be directly caught by user programs. Also, “hypercall” latency drops significantly in recent years.

## How can the research be improved? (the bad side, future work, your idea)

I think what this paper does is that it uses Z3 to prove that the behaviors of “LLVM IR” are the same as “Python specification”. It’s true that this help disentangle complexities in the C language, but it’s still similar to “write two copies of the same program with two languages, and make sure they’re the same”. I don’t think this shows the correctness of the OS kernel. They’re more similar to a “unit test” without “value instantiation” in my perspective.

## If you write this paper, then how would you do? (your idea)

I don’t like the “theory” part of this paper. They’re redundant to me, and I think what it does is simply using mathematical symbols to annotate the descriptions without new information (e.g. deductions).

The experiment part is too short. Hope if space taken by the “theory” part can be used to include more test results and observations.

## What’s your test Results about the paper? (your action)

Nope.

## Give the survey paper list in the same research area (your survey)

[1] [Helgi Sigurbjarnarson](https://dblp.org/pid/150/6007.html), [James Bornholt](https://dblp.org/pid/142/3208.html), [Emina Torlak](https://dblp.org/pid/55/1457.html), [Xi Wang](https://dblp.org/pid/08/5760-5.html):  
**Push-Button Verification of File Systems via Crash Refinement.** [OSDI 2016](https://dblp.org/db/conf/osdi/osdi2016.html#Sigurbjarnarson16): 1-16

[2] [Venugopalan Ramasubramanian](https://dblp.org/pid/85/4250.html), [Emin Gün Sirer](https://dblp.org/pid/s/EminGunSirer.html):  
**Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays.**[NSDI 2004](https://dblp.org/db/conf/nsdi/nsdi2004.html#RamasubramanianS04): 99-112

[3] [Antony I. T. Rowstron](https://dblp.org/pid/r/AITRowstron.html), [Peter Druschel](https://dblp.org/pid/d/PDruschel.html):  
**Storage Management and Caching in PAST, A Large-scale, Persistent Peer-to-peer Storage Utility.**[SOSP 2001](https://dblp.org/db/conf/sosp/sosp2001.html#RowstronD01): 188-201

[4] Lu, Y., Shu, J., Chen, Y., & Li, T. (2017). Octopus: an rdma-enabled distributed persistent memory file system. In 2017*USENIX Annual Technical Conference*(USENIX ATC 17) (pp. 773-785).