Hyperkernel: Push-Button Verification of an OS Kernel

The OS Kernel is a critical component

• Essential for application correctness and security

• Kernel bugs can compromise the entire system
Your PC ran into a problem and needs to restart. We’re just collecting some error info, and then we’ll restart for you. (0% complete)

If you’d like to know more, you can search online later for this error: HAL_INITIALIZATION_FAILED
Your PPR collection complies.

If you'd like to contact us, just e-mail us. (0%)

![Image of a public recycling bin](Image-62x-156-to-1022x629)
Formal verification: high correctness assurance

- Write a spec of expected behavior
- Prove that implementation matches the spec

- Goal: How much can we minimize the proof burden
Formal verification: high correctness assurance

- Write a spec of expected behavior
- Prove that implementation matches the spec

Proof effort: 11 person years

- Goal: How much can we minimize the proof burden
Our result: Hyperkernel

• **Unix-like OS kernel:** based on xv6

• **Fully automated verification using the Z3 solver**
  • Functional correctness of system calls
  • Crosscutting properties (e.g., process isolation)

• **Limitations:**
  • Uniprocessor
  • Initialization & glue code unverified
Designing Hyperkernel for proof automation

_Xv6_  Hyperkernel

- Syscall semantics are loop-y and require writing loop invariants

- Kernel pointers difficult to reason about

- C is difficult to model
Designing Hyperkernel for proof automation

**Xv6**
- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

**Hyperkernel**
- Finite interface
Designing Hyperkernel for proof automation

Xv6

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

Hyperkernel

- Finite interface
- Separate user/kernel spaces and use identity mapping for kernel
Designing Hyperkernel for proof automation

**Xv6**

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

**Hyperkernel**

- Finite interface
- Separate user/kernel spaces and use identity mapping for kernel
- Verify LLVM intermediate representation (IR)
Designing Hyperkernel for proof automation

**Xv6**

- Syscall semantics are loop-y and require writing loop invariants
- Kernel pointers difficult to reason about
- C is difficult to model

**Hyperkernel**

- Finite interface
- Separate user/kernel spaces and use identity mapping for kernel
- Verify LLVM intermediate representation (IR)
Outline

• Verification workflow

• Finite interface design

• Demo

• Evaluation & lessons learned
Outline

• **Verification workflow**

• Finite interface design

• Demo

• Evaluation & lessons learned
Overview of verification workflow

Syscall Implementation

```c
int r;
struct proc *proc;
void *parent_hvm, *child_hvm;
r = alloc_proc(pid, pml4, stack, hvm);
if (r)
    return r;
proc = get_proc(current);
/* copy the kernel stack (saved registers) */
memcpy(get_page(stack), get_page(proc->stack), PAGE_SIZE);
parent_hvm = get_page(proc->hvm);
child_hvm = get_page(hvm);
/* copy stack */
```
Overview of verification workflow

State Machine Specification

Syscall Implementation

```c
int is called by sys_clone in many.
Upon entry, current’s hvm is already flushed.
Upon exit, ran_current() is called to return to user.

int clone_proc(pid_t pid, nl_pid_t pml, nl_t stack, nl_t new
int r);
struct proc *proc;
void *parent_hvm, *child_hvm;

r = alloc_proc(pid, pml, stack, hvm);
if (r)
    return r;
proc = get_proc(current);
/* copy the kernel stack (saved registers) */
memcpy(get_page(stack), get_page(proc->stack), PAGE
parent_hvm = get_page(proc->hvm);
child_hvm = get_page(hvm);
```
Overview of verification workflow

State Machine Specification

- Old
- Pre
- New

Syscall Implementation

```python
def sys_setRunnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```
Overview of verification workflow

State Machine Specification

```
def sys_setRunnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```
Overview of verification workflow

State Machine Specification

```
def sys_setRunnable(old, pid):
    pre = old.procs[pid].state == PROC_EMBRYO
    new = old.copy()
    new.procs[pid].state = PROC_RUNNABLE
    return pre, new
```
Overview of verification workflow

State Machine Specification

pre

old

new

Syscall Implementation

```python
def sys_setRunnable(old, pid):
    pre = old.procs[pid].state == proc_EMBRYO
    new = old.copy()
    new.procs[pid].state = proc_RUNNABLE
    return pre, new
```
Overview of verification workflow

State Machine Specification

Old \rightarrow pre \rightarrow New

Syscall Implementation

LLVM

Verifier
Overview of verification workflow

State Machine Specification

old → pre → new

Syscall Implementation

LLVM

Verifier

Bug

Counterexample

old
Syscall Implementation

Declarative Specification

State Machine Specification

Verifier

Counterexample

Bug
Cross-cutting properties:
- Correctness of reference counters
- Scheduler safety property
- **Process Isolation**
Cross-cutting properties:
- Correctness of reference counters
- Scheduler safety property
- **Process Isolation**

For any virtual address in a process $p$, if the virtual address maps to a page, the page must be exclusively owned by $p$. 
Cross-cutting properties:
- Correctness of reference counters
- Scheduler safety property
- **Process Isolation**

For any virtual address in a process \( p \), if the virtual address maps to a page, the page must be exclusively owned by \( p \).

```plaintext
page, success = page_walk(state, pid, va)

isolation = \textbf{Implies}(success,

state.pages[page].owner == pid)

Show: \textbf{ForAll}([pid, va], isolation)
```
Declarative Specification

State Machine Specification

Syscall Implementation

Verifier

Counterexample
Outline

• Verification workflow

• **Finite interface design**

• Demo

• Evaluation & lessons learned
Verification through symbolic execution

• **Goal**: Minimize proof burden
  - No manual proofs or code annotations

• **Symbolic execution**
  - Fully automated technique, used in bug-finding
  - Full functional verification if program is free of loops and state is finite
  - Feasible when units of work sufficiently small for solving

• **Hyperkernel approach**: Finite interface design
Overview of techniques

• Safely push loops into user space
• Explicit resource management
• Decompose complex syscalls
• Validate linked data structures
• Smart SMT encodings
Overview of techniques

• Safely push loops into user space
• Explicit resource management
• Decompose complex syscalls
• Validate linked data structures
• Smart SMT encodings
The `sbrk()` system call

User space
virtual address space

```
void *sbrk(intptr_t increment)
```
The `sbrk()` system call

```c
void *sbrk(intptr_t increment)
```

increments the program's data space by `increment` bytes.
The `sbrk()` system call

```c
void *sbrk(intptr_t increment)
```

increments the program's data space by `increment` bytes.
The sbrk() system call

**Goal:** Redesign sbrk(); ensuring process isolation.

```
void *sbrk(intptr_t increment)
```

increments the programs data space by increment bytes
The sbrk() system call: Dealing with loops

```c
void *sbrk(intptr_t increment)
```
The sbrk() system call: Dealing with loops

```c
void *sbrk(intptr_t increment)
```
The sbrk() system call: Dealing with loops

```c
void *sbrk(intptr_t increment)
```

Diagram:
- Page table root
- Entry
- 4K page
The sbrk() system call: Dealing with loops

```c
void *sbrk(intptr_t increment)
```

```c
void *sbrk_one_page()
```
The sbrk() system call: Decomposition

```c
void *sbrk_one_page()
```

![Diagram](image)
The sbrk() system call: Decomposition

```c
void *sbrk_one_page()
```
The `sbrk()` system call: Decomposition

void *sbrk_one_page()

alloc_pdpt(...)  alloc_pd(...)  alloc_pt(...)  alloc_frame(...)

PML4 table  page directory  page table  page directory  page table  4K page
The sbrk() system call: Decomposition

```
void *sbrk_one_page()
```

- `alloc_pdpt(...)`
- `alloc_pd(...)`
- `alloc_pt(...)`
- `alloc_frame(...)`

[Diagram showing the decomposition of sbrk() system call involving allocation functions and page table structures.]
The sbrk() system call: Decomposition

```c
int alloc_pdpt(int pml4, size_t index)

int alloc_pd(int pdpt, size_t index)

int alloc_pt(int pd, size_t index)

int alloc_frame(int pt, size_t index)
```
The sbrk() system call: Explicit allocation

1. App allocates memory.
2. Kernel searches for a free page.
3. App receives the page number.
The `sbrk()` system call: Explicit allocation

- Kernel keeps track of per-page metadata: owner/type
- User space searches for free page; kernel validates

![Diagram showing interaction between App and Kernel]
The sbrk() system call: Finite Interface

- `int alloc_pdpt(int pml4, size_t index, int free_pn)`
- `int alloc_pd(int pdpt, size_t index, int free_pn)`
- `int alloc_pt(int pd, size_t index, int free_pn)`
- `int alloc_frame(int pt, size_t index, int free_pn)`

Any composition of these system calls maintains isolation

For any virtual address in a process p, if the virtual address maps to a page, the page must be exclusively owned by p.
# Implementation

<table>
<thead>
<tr>
<th>Component</th>
<th>Lines</th>
<th>Languages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kernel implementation</td>
<td>7,616</td>
<td>C, assembly</td>
</tr>
<tr>
<td>State-machine specification</td>
<td>804</td>
<td>Python</td>
</tr>
<tr>
<td>Declarative specification</td>
<td>263</td>
<td>Python</td>
</tr>
<tr>
<td>Verifier</td>
<td>2,878</td>
<td>C++, Python</td>
</tr>
<tr>
<td>User-space implementation</td>
<td>10,025</td>
<td>C, assembly</td>
</tr>
</tbody>
</table>
Outline

• Verification workflow

• Finite interface design

• Demo

• Evaluation & lessons learned
Demo

• Hyperkernel in action

• Catching a low-level bug producing a stack trace

• Catching a process isolation bug producing a visualized counterexample
Outline

• Verification workflow

• Finite interface design

• Demo

• Evaluation & lessons learned
What was the development effort?

• Write a state machine specification

• Relate LLVM data structures to abstract specification state

• Write checks for the representation invariants if needed.
What was the development effort?

• Write a state machine specification

• Relate LLVM data structures to abstract specification state

• Write checks for the representation invariants if needed.

• Adding and verifying a system call usually takes < 1 hour
Is the design effective for scalable verification?

- 45 minutes on a single core machine
- 15 minutes on an 8-core Intel i7
- Not sensitive to system parameters (e.g., number of pages)

- Design is effective for scalable verification
Conclusion

• Feasible to verify simple Unix-like OS kernel

• Automatic verification through symbolic execution
  • Make interface finite
  • Decompose complex system calls to scale verification

• Verifiability as a first-class system design concern

• http://locore.cs.washington.edu/hyperkernel