Using Build-Integrated Static Checking to Preserve Correctness Invariants

Hao Chen
University of California, Davis

Jonathan Shapiro
Johns Hopkins University

Motivation

- A key problem in creating secure systems:
  - Demonstrate the correspondence between the design and implementation, and
  - Preserve the correspondence, and
  - Achieve these goals cost effectively
- Success stories
  - Static analysis tools have found many bugs

Open questions

- Are these tools cost-effective for preventing bugs?
  - How easy to write specifications by non-tool developers?
  - How easy to integrate the tools into the build process?
  - How much overhead does the checking add to the build process?

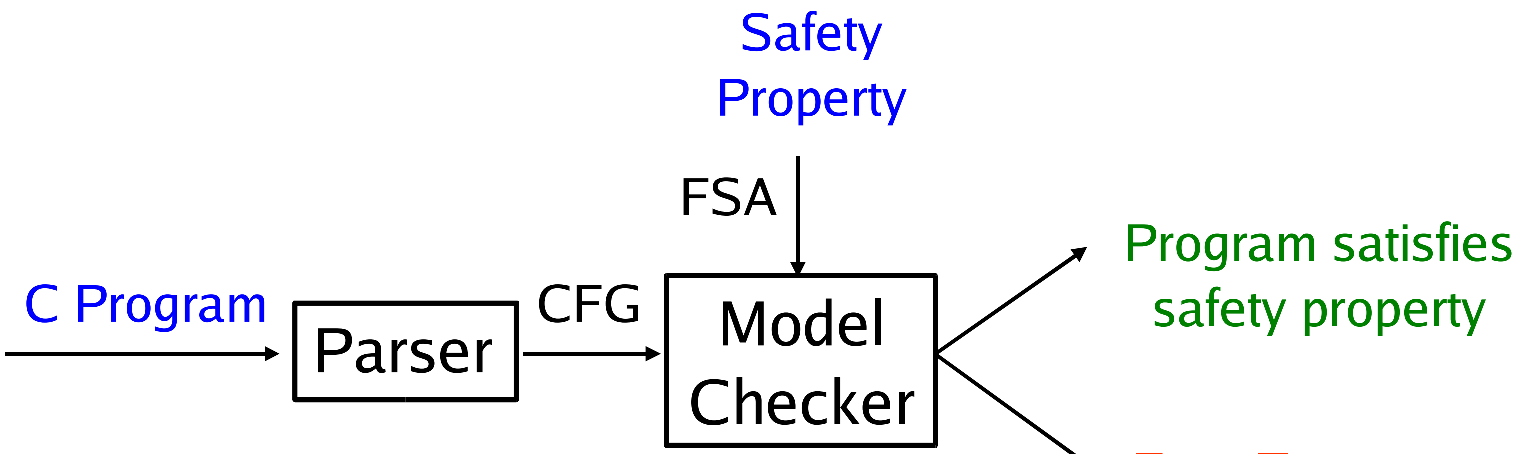
Our work

- Empirically examine the cost and gain of embedding a static analysis tool into the development cycle of software
- Conduct a case study
  - Analysis tool: MOPS
  - Software checked: EROS

MOPS

- A static analysis tool that checks source programs for temporal safety properties
  - *e.g.: a setuid-root program must drop privilege before making risky system calls*.
- Sound analysis under certain assumptions
  - Memory safety
  - No non-local jumps
  - No pointer aliasing
  - ...

The MOPS process



FSA: finite state automaton
CFG: control flow graph

Treat the model checker as a black box for this talk

EROS

- A capability-based OS running on commodity hardware
- We focus on the EROS microkernel
  - Interrupt-style kernel
  - Single-level storage
  - Caching design

Properties checked

- Transactional requirement in system calls
- Sleeping and yielding
- Interrupt enables and disables
- Caching requirement
- Consistency in the memory subsystem

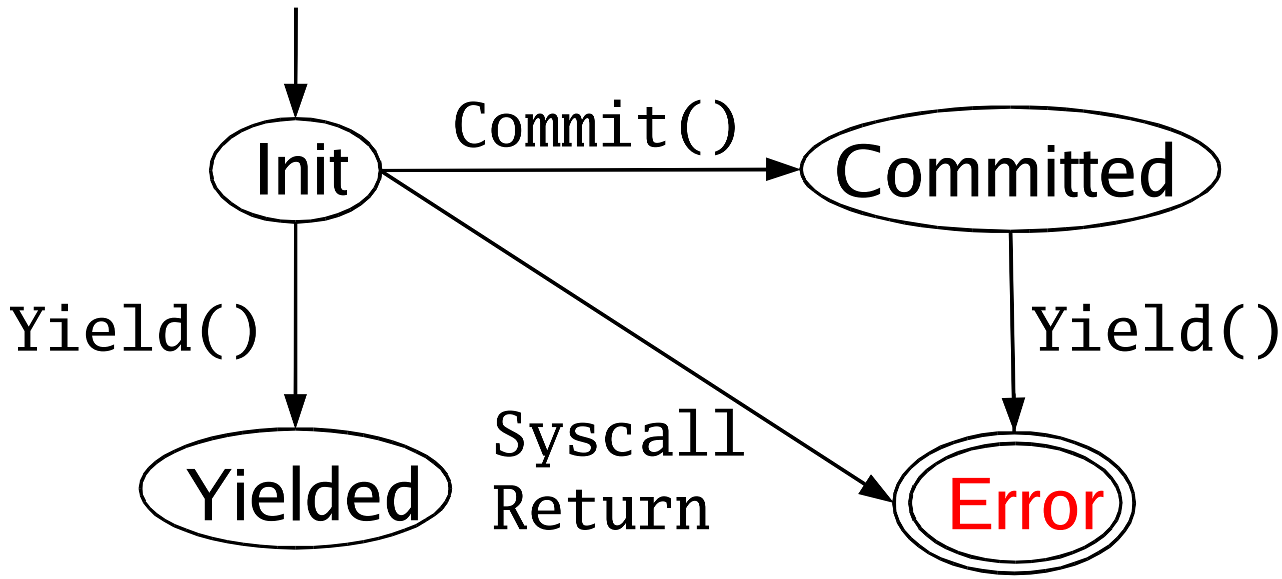
Property: transactional requirement in system calls

- EROS is an interrupt-style kernel
  - When blocked, a process does not retain a kernel stack
  - Upon wake up, the process restarts the system call

Commit point

- A commit point separates the two phases of a system call
  - Prepare phase: check preconditions.
    If preconditions unsatisfiable, `Yield()`
  - Action phase: must complete the operation
  - `Commit()` separates the two phases

Property: transactional requirement in system calls



- Every path should invoke exactly one of Yield() or Commit()
- After Commit(), should not invoke Yield()

Bug in system call transaction

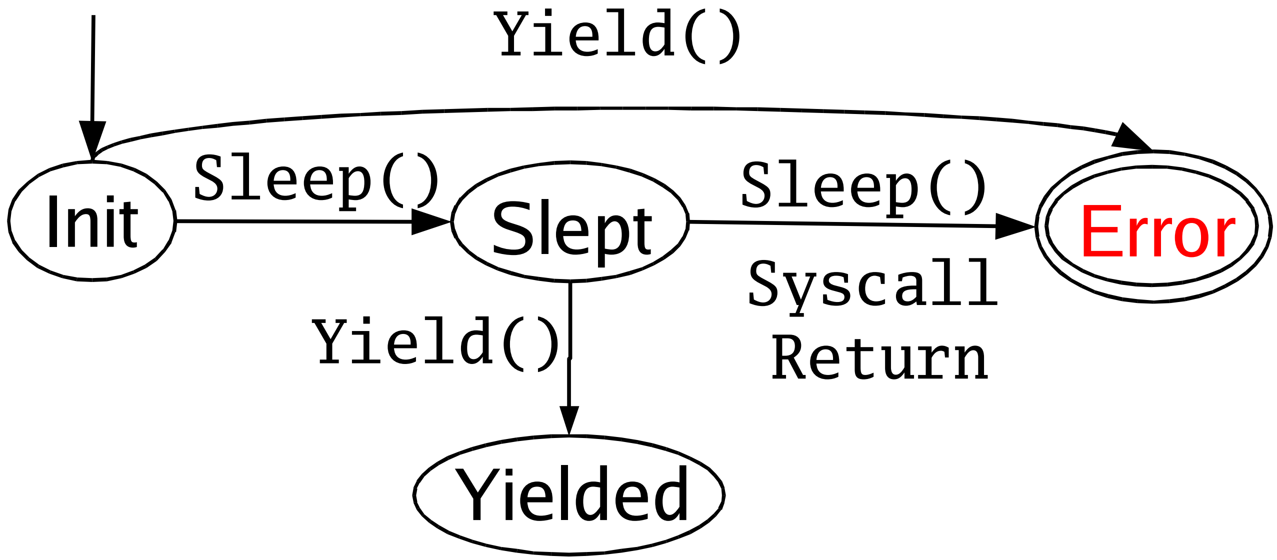
```
int syscall(...)
{
  // commit point
  Commit();
  ...
  p = malloc();
}

void *malloc(size_t len)
{
  if (memory unavaiable)
    Yield();
}
```

Property: Sleep() and Yield()

- EROS differs from typical kernels in that
  - A process can sleep on at most one queue at any time
  - Sleep() and Yield() are not atomically joined.
    - Sleep() places the process on a sleep queue
    - Yield() relinquishes the CPU

Sleep() and Yield(): first try



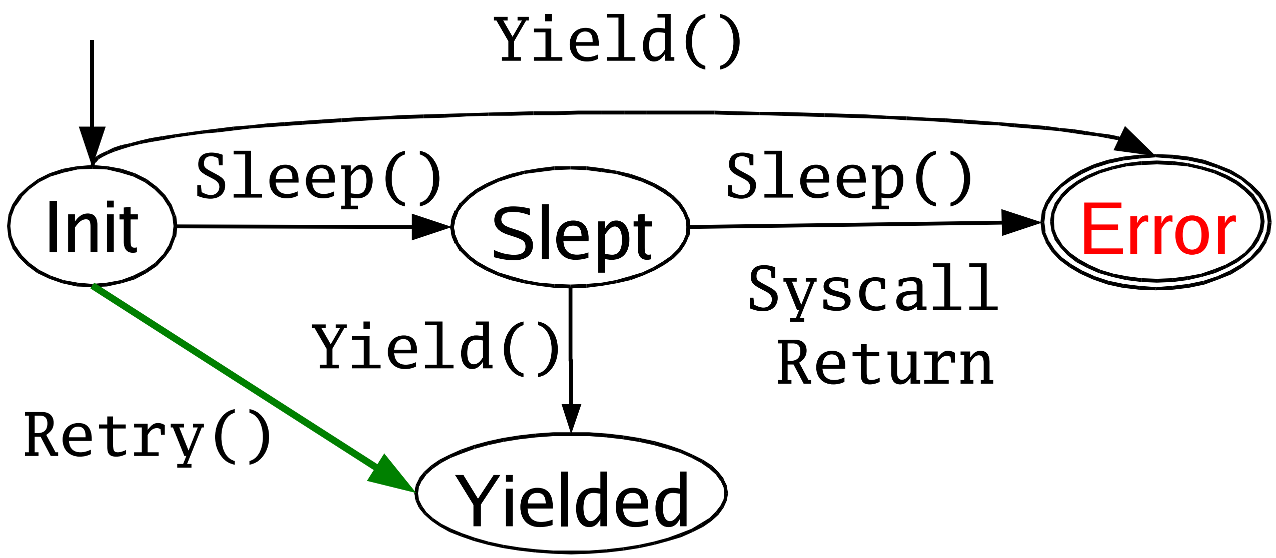
- No kernel path can invoke Sleep() more than once.
- After Sleep(), the kernel must call Yield().
- Before Yield(), the kernel must call Sleep().

Sleep() and Yield(): problem

- Problem
  - Occasionally, it is allowable to invoke Yield() without invoking Sleep() first
  - Reason: needs to abort and retry the current system call immediately
- Result: false positive errors

Sleep() and Yield(): solution

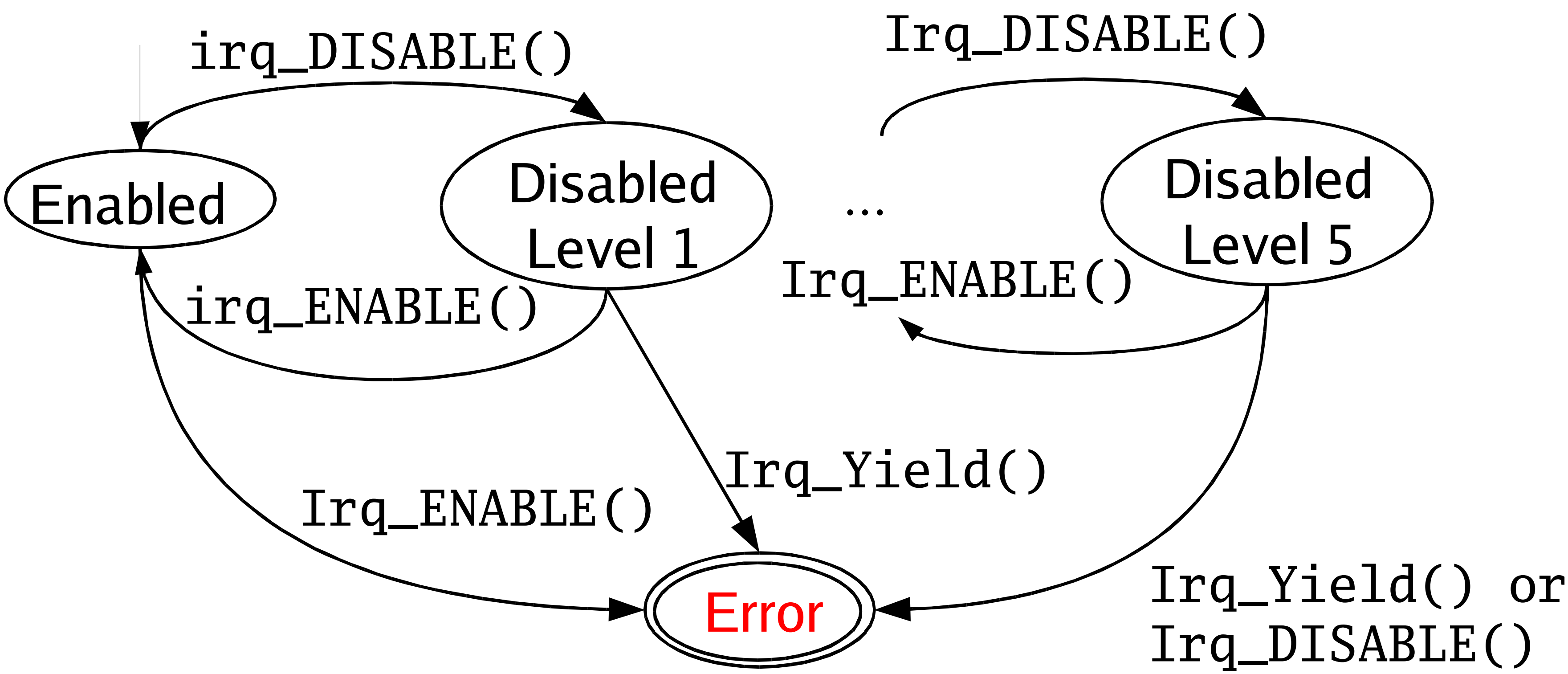
- Wrap those special Yield() in Retry()
  - Avoid false positives
  - Result in cleaner code



Property: interrupt enable and disable

- Property
  - Properly nest interrupt enables and disables
  - Do not invoke Yield() while interrupt is disabled
- Problem
  - Property needs a counter, so cannot be accurately described by an FSA
  - Solution: approximate the property using a guard state

Property:
interrupt enable and disable



Evaluation: Usability

- Setup: cooperation between
  - A MOPS developer
  - An EROS developer
- Experience
  - EROS developer wrote specifications by himself
    - took 16 hours spanning several conference trips
  - Only a few iterations is needed for each property
  - Found a few false positives

Integration and Performance

- Integrating MOPS into EROS
  - Took less than an hour
- Performance
  - EROS kernel: 26K lines of code
  - Checking five properties took 100 seconds
  - Fast enough to be part of every major build
  - Could be improved

Related work

- Static analysis tools
  - For temporal safety properties
    - SLAM, BLAST, ESP, MC
  - For other properties
    - Cqual, ESC/Java, Splint
- We expect that our conclusion applies to many these tools as well.

Conclusions

- Requirements for an effective tool for preventing temporal safety errors
  - Be sound
  - Have specifications that typical testers can write
  - Require no invasive change to the code base
  - Be efficient enough to be incorporated into the build process
- Should incorporate these tools into the development of critical software more broadly