A Formally Verified Runtime

Bernhard Egger

2013/02/01
A Smart New World

- everything is “smart” these days
  - most noticeable feature: user-generated content on devices
  - less noticeable: increased complexity of hardware, increased complexity of software

Samsung GALAXY
A Smart New World – And a Host of New Problems

- we run more and more security- and safety-critical stuff on our smart devices
  - mobile banking
  - smart wallets
  - personal health monitoring

- and then there’s the issue of privacy
I Won’t Be Evil – Just Trust Me

- prevalent model for smartphones is based on trust
- the trust model of Android and iOS
  - limited privileges per default
  - safe programming language (Android - Java)
  - walled gardens (app stores), app permissions
- lots of problems with this approach
  - root exploits
  - native code
  - verification, permission difficulties
I Can’t Be Evil – Here’s The Mathematical Proof

- ideal world: entire system is verified
  - not (yet) possible

- realistic world: use a verified base to run software on
  - formally verified kernels/hypervisors
    - Verisoft, seL4, SecVisor
  - formally verified compilers
    - CompCert C compiler, C0 compiler

- run secure/insecure S/W alongside
Pioneers – Secure Unix, PSOS, KIT

- UCLA Secure Unix (1980)
  - microkernel-like structure
  - implemented in Pascal
  - used (an early variant of) formal refinement
  - proof for kernel only, but “tedious and painful”
  - performance up to an order of magnitude below par

- Provably Secure Operating System (1973-80)
  - considers the entire OS
  - layered approach
Pioneers – Secure Unix, PSOS, KIT

- KIT (kernel for isolated tasks, 1989)
  - first formally verified kernel
  - ~600 lines of assembler source code
  - provides (static) task switching, async I/O, exceptions, message-passing
Verisoft / VerisoftXT

- Verisoft (2003-2007)
  - government funded (~15m euro, 250 억원), partly confidential

- pervasive
  - does not rely on the correctness of the ISA or compiler
  - unbroken chain from H/W to applications

- layered approach similar to PSOS

- theorem prover: Isabelle/HOL
Verisoft / VerisoftXT

- Verisoft (2003-2007)
  - hardware layer: VAMP processor
    - formally verified down to the gate level
  - kernel layers:
    - CVM (Communicating Virtual Machines)
    - VAMOS
  - user mode layers
    - SOS (Simple Operating System) – privileged process
    - applications
Verisoft / VerisoftXT

- Verisoft (2003-2007)
  - verified compiler tool chain
    - C0, C0A language
    - bootstrapped
  - code vs proof:
    - compiler source: 1’500 lines of C0 code
    - proof: 85’000 lines of Isabelle code
- successor: VerisoftXT (2007-2010)
seL4

- ongoing joint project of people at NICTA, UNSW, and Open Kernel Labs

- production-quality, commercialized general-purpose microkernel

- formally verified from abstract specification down to C implementation
  - assumes correctness of
    - hardware, boot code, assembly code, compiler
  - everything else is proven
seL4

- interesting from the design/proof/implementation point of view
  - Haskell prototype
    - intermediate target suitable for formal methods and implementation
seL4

- formal verification
  - interactive, machine-assisted, machine-checked proof
  - theorem prover used: Isabelle/HOL

- refinement proofs
  establish correspondence between high-level and low-level representations of a system
  - correspondence: Hoare logic properties hold for both levels

source: Klein, 2009
formal verification

```
schedule ≡ do
threads ← all_active_tcbs;
thread ← select threads;
switch_to_thread thread
od OR switch_to_idle_thread

schedule = do
  action ← getSchedulerAction
  case action of
    ChooseNewThread → do
      chooseThread
      setSchedulerAction ResumeCurrentThread
    ...
    chooseThread = do
      r ← findM chooseThread’ (reverse [minBound .. maxBound])
      when (r == Nothing) $ switchToIdleThread
    chooseThread’ prio = do
      q ← getQueue prio
      liftM isJust $ findM chooseThread’’ q
    chooseThread’’ thread = do
      runnable ← isRunnable thread
      if not runnable then do
        tcbSchedDequeue thread
        return False
      else do
        switchToThread thread
        return True
```
seL4

- the proof of functional correctness
  - reduction of refinement to forward simulation
  - for each transition in $M_2$: $s \rightarrow s'$ show that there exists a corresponding transition on the abstract side $M_1$: $\sigma \rightarrow \sigma'$.
  - find a relation $R$ that holds for each possible transition between the states $s$ and $\sigma$, and $s'$ and one in $\sigma'$.

source: Klein, 2009
seL4

- kernel design and implementation
  - reduced use of global variables
    - simplifies proof
  - memory management in user level
    - separate proof
- concurrency
  - single processor
  - no exceptions
  - no yielding
  - interrupts disabled; poll-rollback-restart model
seL4

- implementation and verification effort
  - Haskell prototype: 5’700 LOC, 2 person years
  - C implementation: 8’700 LOC, 2 person months
  - proof: 200’000 LOP, > 20 person years
Okay, So What Now?

- There is still a lot of work to do
  - verified boot code, assembly code, and compiler
  - support for true concurrency (multiple cores)
  - support for H/W accelerators
References


