Security economics

CS642:
Computer Security

Topics

- What are the economic incentives for security
- What are the disincentives?
- How can economics help security?
- Hope for the future

Penetrate and Patch

- Usual approach to software development
  - Develop product as quickly as possible
  - Release it without adequate testing
  - Patch the code as flaws are discovered
- In security, this is "penetrate and patch"
  - A **bad** approach to software development
  - An **even worse** approach to secure software!

Why Penetrate and Patch?

- First to market advantage
  - First to market likely to become market leader
  - Market leader has huge advantage in software
  - Users find it safer to “follow the leader”
  - Boss won’t complain if your system has a flaw, as long as everybody else has same flaw…
  - User can ask more people for support, etc.
- Sometimes called “network economics”

Why Penetrate and Patch?

- Secure software development is hard
  - Costly and time consuming development
  - Costly and time consuming testing
  - Cheaper to let customers do the work!
- No serious economic disincentive
  - Even if software flaw causes major losses, the software vendor is not liable
  - Is any other product sold this way?
  - Would it matter if vendors were legally liable?

Penetrate and Patch Fallacy

- **Fallacy:** If you keep patching software, eventually it will be secure
- Why is this a fallacy?
- Empirical evidence to the contrary
- Patches often add new flaws
- Software is a moving target: new versions, features, changing environment, new uses,…

Open vs Closed Source

- Open source software
  - The source code is available to user
  - For example, Linux
- Closed source
  - The source code is not available to user
  - For example, Windows
- What are the security implications?

Open Source Security

- Claimed advantages of open source is
  - **More eyeballs:** more people looking at the code should imply fewer flaws
  - A variant on Kerchoffs Principle
- Is this valid?
  - How many "eyeballs" looking for security flaws?
  - How many "eyeballs" focused on boring parts?
  - How many "eyeballs" belong to security experts?
  - Attackers can also look for flaws!
  - Evil coder might be able to insert a flaw

Open Source Security

- Open source example: wu-ftp
  - About 8,000 lines of code
  - A security-critical application
  - Was deployed and widely used
  - After 10 years, serious security flaws discovered!
- More generally, open source software has done little to reduce security flaws
- Why?
  - Open source follows penetrate and patch model!

Closed Source Security

- Claimed advantage of closed source
  - Security flaws not as visible to attacker
  - This is a form of “security by obscurity”
- Is this valid?
  - Many exploits do not require source code
  - Possible to analyze closed source code…
  - …though it is a lot of work!
  - Is “security by obscurity” real security?

Open vs Closed Source

- Advocates of open source often cite the **Microsoft fallacy** which states
  1. Microsoft makes bad software
  2. Microsoft software is closed source
  3. Therefore all closed source software is bad
- Why is this a fallacy?
  - Not logically correct
  - More relevant is the fact that Microsoft follows the penetrate and patch model

Open vs Closed Source

- No obvious security advantage to either open or closed source
- More significant than open vs closed source is software development practices
- Both open and closed source follow the "penetrate and patch" model

Open vs Closed Source

- If there is no security difference, why is Microsoft software attacked so often?
  - Microsoft is a big target!
  - Attacker wants most “bang for the buck”
- Few exploits against Mac OS X
  - **Not** because OS X is inherently more secure
  - An OS X attack would do less damage
  - Would bring less “glory” to attacker
- Next, we consider the theoretical differences
  - See this paper

Security and Testing

- The fundamental problem
  - Good guys must find (almost) all flaws
  - Bad guy only needs 1 (exploitable) flaw
- Software reliability far more difficult in security than elsewhere
- How much more difficult?
  - See the next slide…

Security Testing: Do the Math

- Suppose 10^6 security flaws in some software
  - Say, MacOS X
- Suppose each bug has MTBF of 10^9 hours
- Expect to find 1 bug for every 10^3 hours testing
- Good guys spend 10^7 hours testing: **find 10^4 bugs**
  - Good guys have found 1% of all the bugs
- Trudy spends 10^3 hours of testing: **finds 1 bug**
- Chance good guys found Trudy's bug is only **1%** !!!

Cost vs. benefit

- Rational attackers compare the cost of an attack with the gains from it
  - Attackers look for the weakest link; thus, little is gained by strengthening the already strong bits
- Rational defenders compare the risk of an attack with the cost of implementing defenses
  - Lampson: “Perfect security is the enemy of good security”
- But human behavior is not always rational:
  - Attackers follow each other and flock all to the same path
  - Defenders buy a peace of mind; avoid personal liability by doing what everyone else does

  → Many events are explained better by group behavior than rational choice

System Purchasing Economics

- What are incentives for person recommending a software purchase?
  - Get the best product

Or

  - Don’t get fired?

- Buying from big companies (IBM, Microsoft) easy to defend even if less secure

Who pays for security?

- US banks:
  - System assumes when customer complains they are correct unless *bank proves fraud*
- British banks
  - System assumes banks are secure: when customer complains they can be accused of fraud

Which leads to better security, and by whom?

Security-based Costs

- US Electronic payments (EMV)
  - Cost of a transaction varies based on security practices
    - Did you check a PIN
    - Did you get a signature
  - Liability for fraud lies partially on seller
    - Encourage them to have better practices

Botnets

- Mirai botnet: millions of IoT security cameras compromised to form a botnet
  - Launches massive denial of service attacks.

Who pays to make security cameras more secure?

Why pays cost of a botnet?

Secure Software Evaluation

- US orange book: criteria for military-grade security
  - US government pays for evaluation of software
  - Cheaper for companies
- European Common Criteria: criteria for military-grade software
  - Vendor hires auditor to evaluate software
  - Cheaper for government

Which one has better aligned incentives?

Principle: assign security to the correct entity

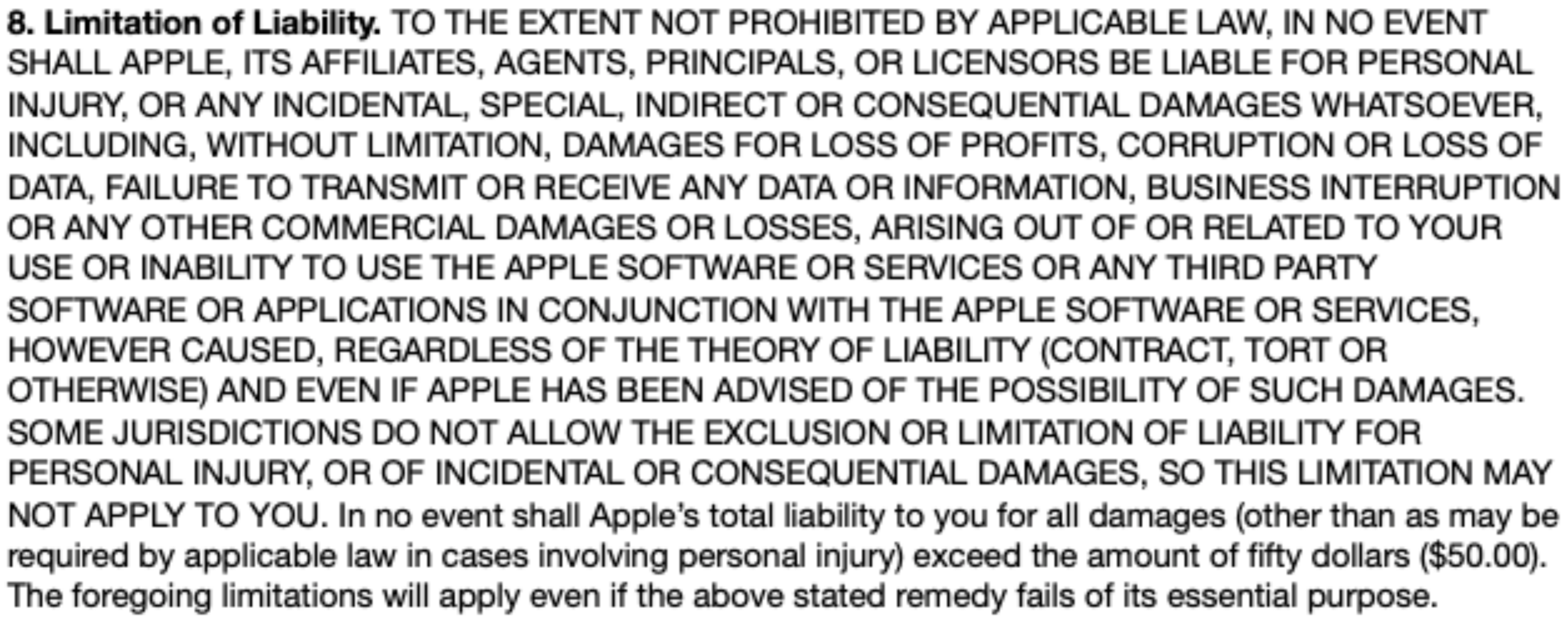
- Make the entity that is charged or benefits from security do the evaluation
  - Hire your own evaluators
- Charge the entity that is best able to do something about it
  - Banks can do a lot more for security than customers
  - Device vendors can do more than users

Software Liability

- Without an agreement, company liable if:
  - The software vendor owed the user a duty to provide functioning software;
  - The software did not live up to that standard;
  - The user suffered harm;
  - The software caused that harm.
- Should software vendors be liable for harm from (security) bugs?

Software liability (2)

- License agreements:



8. Limitation of Liability. TO THE EXTENT NOT PROHIBITED BY APPLICABLE LAW, IN NO EVENT SHALL APPLE, ITS AFFILIATES, AGENTS, PRINCIPALS, OR LICENSORS BE LIABLE FOR PERSONAL INJURY, OR ANY INCIDENTAL, SPECIAL, INDIRECT OR CONSEQUENTIAL DAMAGES WHATSOEVER, INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF PROFITS, CORRUPTION OR LOSS OF DATA, FAILURE TO TRANSMIT OR RECEIVE ANY DATA OR INFORMATION, BUSINESS INTERRUPTION OR ANY OTHER COMMERCIAL DAMAGES OR LOSSES, ARISING OUT OF OR RELATED TO YOUR USE OR INABILITY TO USE THE APPLE SOFTWARE OR SERVICES OR ANY THIRD PARTY SOFTWARE OR APPLICATIONS IN CONJUNCTION WITH THE APPLE SOFTWARE OR SERVICES, HOWEVER CAUSED, REGARDLESS OF THE THEORY OF LIABILITY (CONTRACT, TORT OR OTHERWISE) AND EVEN IF APPLE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY FOR PERSONAL INJURY, OR OF INCIDENTAL OR CONSEQUENTIAL DAMAGES, SO THIS LIMITATION MAY NOT APPLY TO YOU. In no event shall Apple's total liability to you for all damages (other than as may be required by applicable law in cases involving personal injury) exceed the amount of fifty dollars ($50.00). The foregoing limitations will apply even if the above stated remedy fails of its essential purpose.

Software Liability (3)

- Open source software
  - Why should be liable for security bugs in Linux?

Customer information

- How do you buy secure software?
- How do you buy a used car?
  - Suppose 20% of used cars are lemons, but you can’t tell which
  - How much would you pay for car?

Information about security

- What information is available to a purchasing decision?
  - Price - published
  - Features - observable
  - Performance - measurable
  - Reputation - available
- What is not available?
  - Security
  - Reliability

Solutions

- How do we deal with cars/drivers that can hurt other people?
  - Require insurance
  - Worse drivers pay more -> incentive to be a better driver or stop driving
- How insurance helps:
  - Charge people if their computer is involved in an attack
  - Sell them insurance to cover the cost
  - Better security practices -> cheaper insurance

Hope from Academia

- SeL4
  - Microkernel: A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality.

Secure L4 (seL4) – Design Goals

- Create a formal model of a microkernel
- Implement the microkernel
- Prove that it always behaves according to the specification

Assumptions

- Hardware works correctly
- Compiler produces machine code that fits their formalization
- Some unchecked assembly code is correct
- Boot loader is correct

Reminder

- Not all equivalent programs are equally amenable to verification

```
void swap(ptr A, ptr B)
{
       ptr C := A;
       A := B;
       B := C;
}
```

vs.

```
void swap(ptr A, ptr B)
{
       A := A xor B;
       B := A xor B;
       A := A xor B;
}
```

How to design kernel + spec?

- **Bottom-Up-Approach**: Concentrate on low-level details to maximize performance

- **Problem**: Produces complex design, hard to verify

How to design kernel + spec?

- **Top-Down-Approach**: Create formal model of kernel and generate code from it
- **Problem**: High level of abstraction from hardware

How to design kernel + spec?

- **Compromise**: build prototype in high-level language (Haskell)
- Generate “executable specification” from prototype
- Re-implement executable specification in C
- Prove refinements:
  - C ⇔ executable specification
  - Executable specification ⇔ Abstract specification (more high-level)

Sidebar: Invariants

A property that is true of every state*.

* at least at public method boundaries.

For example, inserting a node into a linked list may cause the list to become temporarily disconnected.

Invariants may need to be verified for every part of the system, not just the parts of the system that obviously manipulate the structure in question.

Example: desired file synchronizer properties?

The synchronizer doesn't eat my files.

- partial file updates work correctly
- conflicts are handled in some sane manner
- massive deletes are not propagated without warning

I always have the latest version of my files.

- what about network latency?
    - prioritize file transfers

Synchronization is idempotent.

- f(f(x)) = f(x)
- e.g., set union is idempotent

What properties do we expect from a kernel?

- Every system call terminates.
- No exceptions thrown.
- No arithmetic problems (e.g., overflow, divide by zero)
- No null pointer de-references.
- No ill-typed pointer de-references.
- No memory leaks.
- No buffer overflows.
- No unchecked user arguments.
- Code injection attacks are impossible.
- Well-formed data structures.
- Correct book-keeping.
- No two objects overlap in memory
- *etc.*

Iterative Co-design of Kernel & Proof

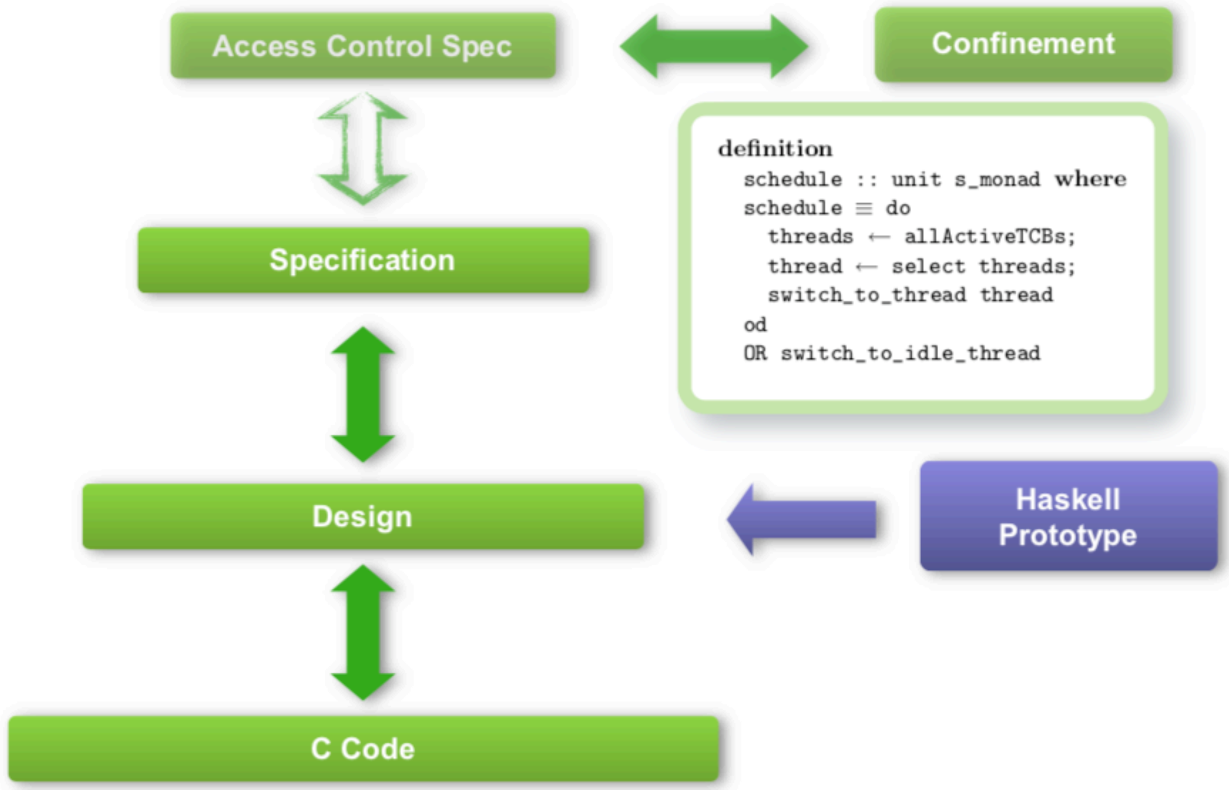
Kernel Team

1. Initial prototype (Haskell)
   - no interrupts
   - single address space
   - generic linear page table
2. Complete prototype
   - add missing functionality
3. Implementation

Proof Team

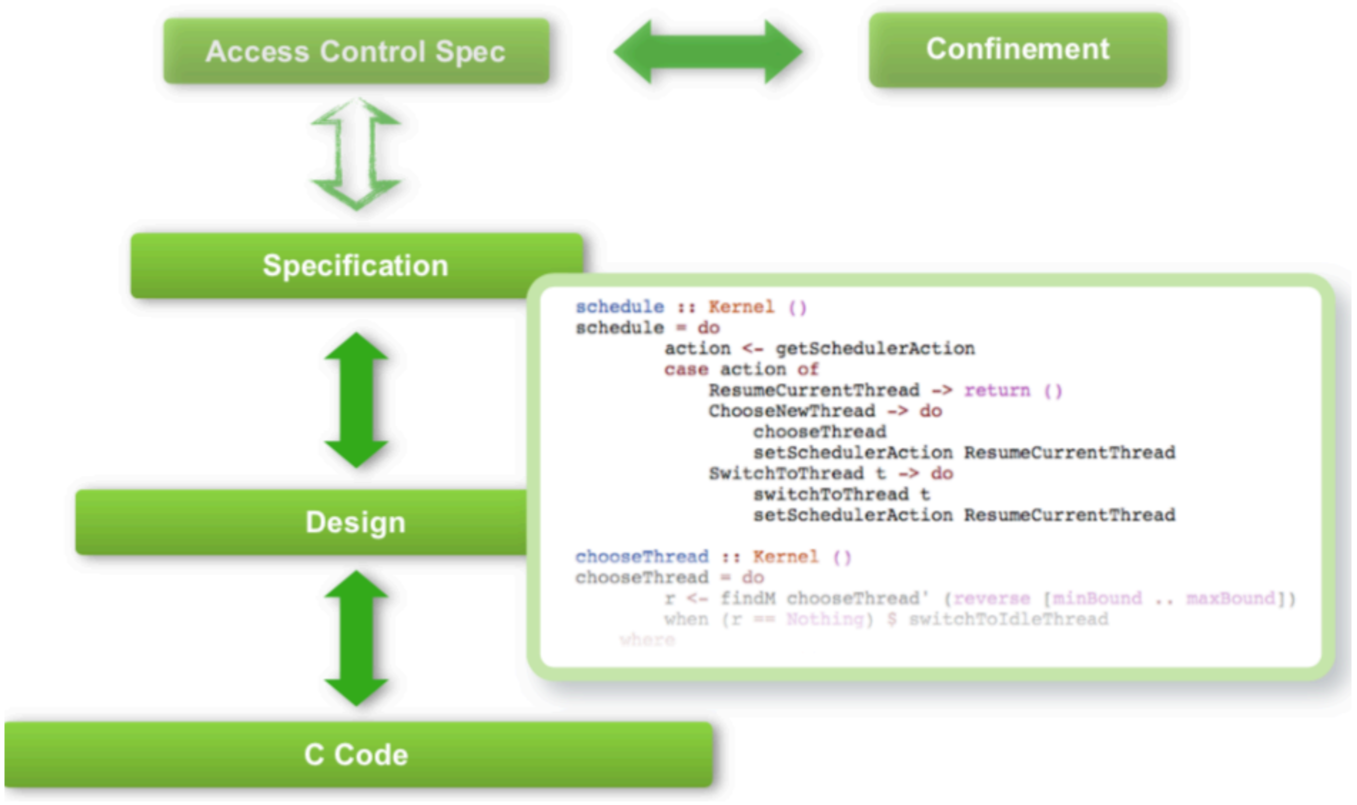
1. Infrastructure
2. Abstract Spec
   - prototype vs spec
3. Spec vs Implementation

How to prove an OS?



```
definition
  schedule :: unit s_monad where
  schedule ≡ do
    threads ← allActiveTCBs;
    thread ← select threads;
    switch_to_thread thread
  od
  OR switch_to_idle_thread
```

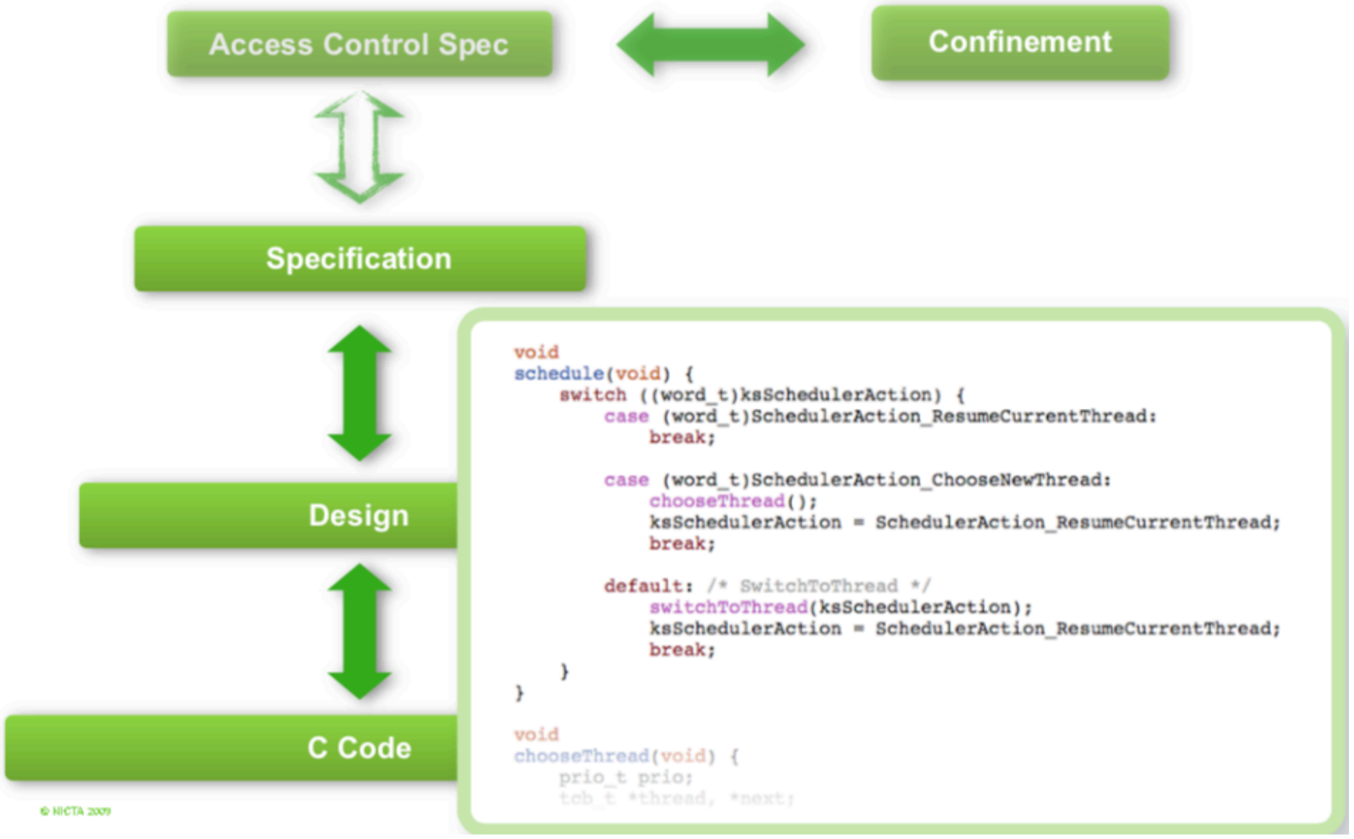
Haskell executable design specification



```haskell
schedule :: Kernel ()
schedule = do
        action <- getSchedulerAction
        case action of
            ResumeCurrentThread -> return ()
            ChooseNewThread -> do
                chooseThread
                setSchedulerAction ResumeCurrentThread
            SwitchToThread t -> do
                switchToThread t
                setSchedulerAction ResumeCurrentThread

chooseThread :: Kernel ()
chooseThread = do
        r <- findM chooseThread' (reverse [minBound .. maxBound])
        when (r == Nothing) $ switchToIdleThread
    where
```

C Implementation



```
void
schedule(void) {
    switch ((word_t)ksSchedulerAction) {
        case (word_t)SchedulerAction_ResumeCurrentThread:
            break;

        case (word_t)SchedulerAction_ChooseNewThread:
            chooseThread();
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;

        default: /* SwitchToThread */
            switchToThread(ksSchedulerAction);
            ksSchedulerAction = SchedulerAction_ResumeCurrentThread;
            break;
    }
}

void
chooseThread(void) {
    prio_t prio;
    tcb_t *thread, *next;
```

Abstract Spec (Isabelle/HOL)

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

Executable Spec (Haskell)

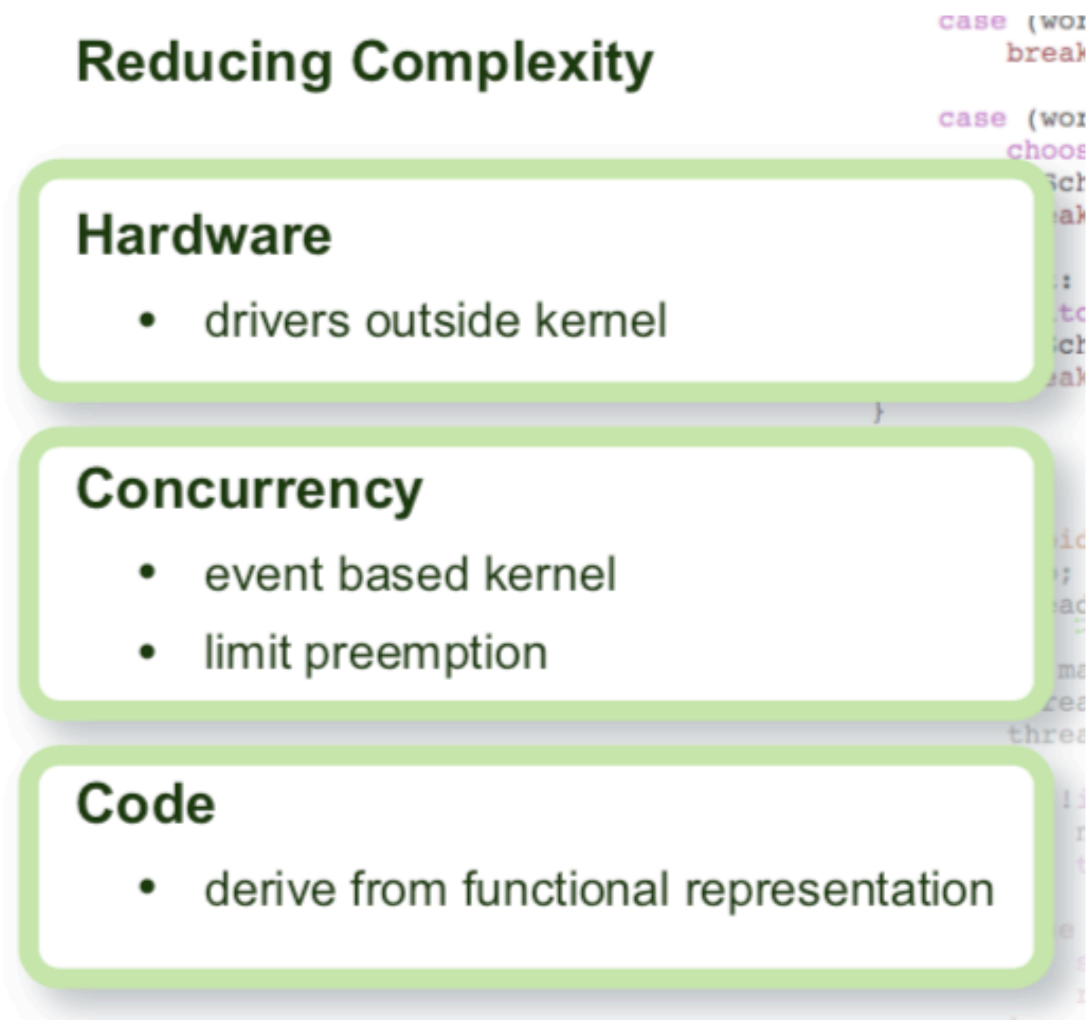
```
schedule = do
  action <- getSchedulerAction
  case action of
    ChooseNewThread -> do
      chooseThread
      setSchedulerAction ResumeCurrentThread
      ...
chooseThread = do
    r <- findM chooseThread' (reverse [minBound .. maxBound])
    when (r == Nothing) $ switchToIdleThread
chooseThread' 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
```

The implementation in C is much larger.

```
void setPriority(tcb_t *tptr, prio_t prio) {
  prio_t oldprio;
  if(thread_state_get_tcbQueued(tptr->tcbState)) {
    oldprio = tptr->tcbPriority;
    ksReadyQueues[oldprio] =
      tcbSchedDequeue(tptr, ksReadyQueues[oldprio]);
    if(isRunnable(tptr)) {
      ksReadyQueues[prio] =
        tcbSchedEnqueue(tptr, ksReadyQueues[prio]);
    }
    else {
      thread_state_ptr_set_tcbQueued(&tptr->tcbState,
                                     false);
    }
  }
  tptr->tcbPriority = prio;
}
```

A smidgen of C from the scheduler

Design for verification



Reducing Complexity

Hardware

- drivers outside kernel

Concurrency

- event based kernel
- limit preemption

Code

- derive from functional representation

Important Design Decisions

1. Global variables & Side Effects
2. Kernel Memory Management
3. Concurrency & Non-Determinism
4. I/O

Global Variables & Side Effects

Use sparingly. Expensive to verify because they require invariants, which need to be checked against all code. Keep them modular and under control.

Haskell prototype helped with this, since side-effects in Haskell have to be made explicit.

Kernel Memory Management

Kernel only has mechanism.

Push policy to userspace.

- => don't need to verify policy

Invariants about the state of memory book-keeping data structures mean that certain checks can be done quickly at runtime. (This would not be safe without the proof.)

Concurrency & Non-Determinism

Short system calls.

Disable interrupts during system calls.

Therefore, no concurrency in the kernel.

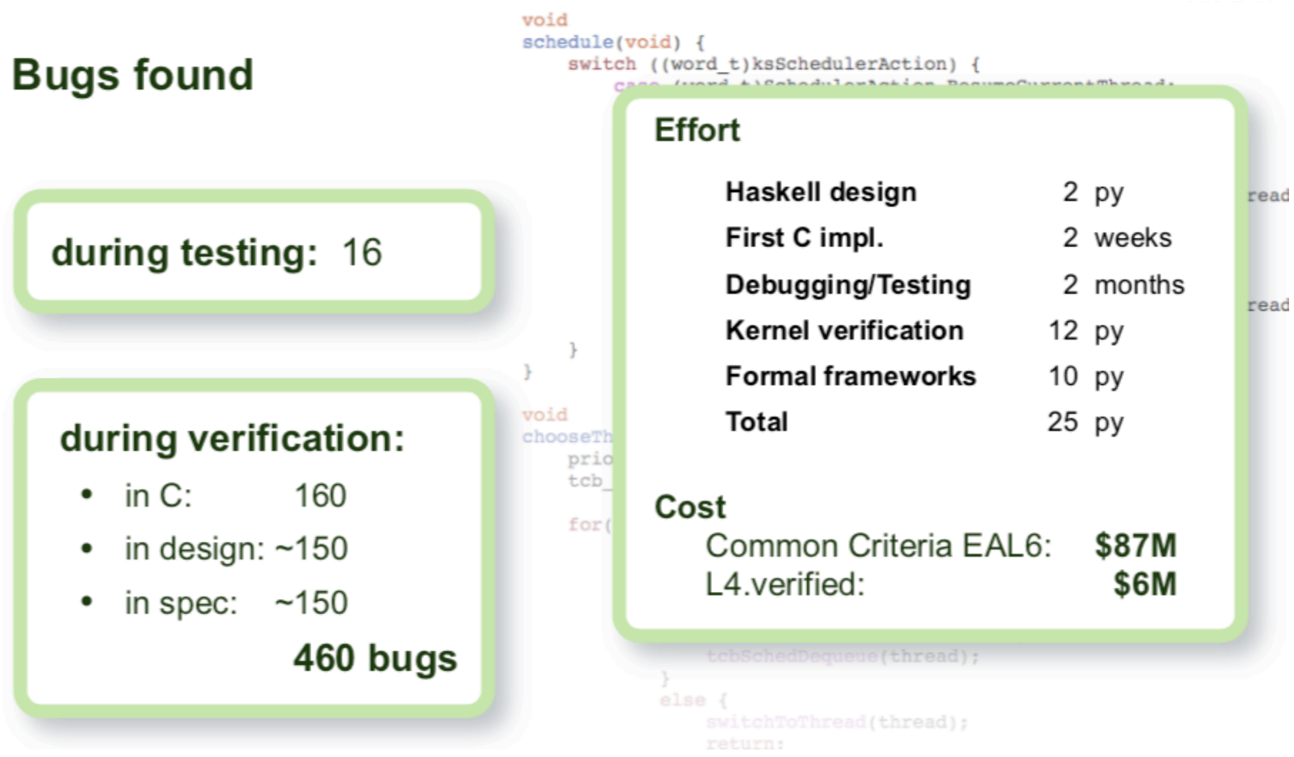
Easy.

I/O

Hardware devices generate interrupts.

These are converted to IPC messages for the userspace device drivers. Hence, much complexity removed from kernel.

Usefulness / cost



Bugs found

during testing: 16

during verification:

- in C: 160
- in design: ~150
- in spec: ~150

460 bugs

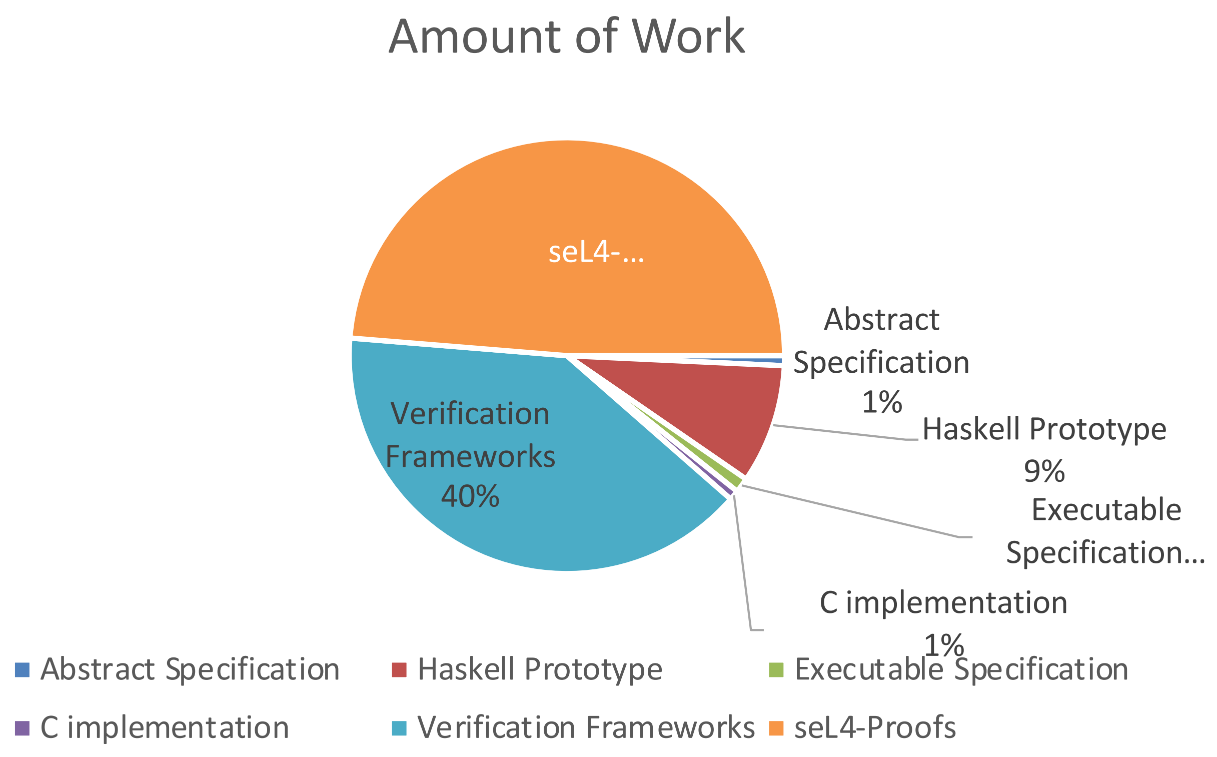
Effort

Haskell design	2 py
First C impl.	2 weeks
Debugging/Testing	2 months
Kernel verification	12 py
Formal frameworks	10 py
Total	25 py

Cost

Common Criteria EAL6:	**$87M**
L4.verified:	**$6M**

Cost of Verification



Source of Data: seL4, Klein et al.

Takeaway

- Functional verification of microkernels is possible
- Performance of verified kernels can be OK

- However:
  - Verification is a huge effort
  - Still needs to assume compiler correctness (➔ huge trusted base)

- Is proving functional correctness worth the effort?