Proof Engineering Considered Essential

FM’14 Singapore
Gerwin Klein
An exception 06 has occurred at 0028:C11B3ADC in VxD DiskTSD(03) + 00001660. This was called from 0028:C11B40C8 in VxD voltrack(04) + 00000000. It may be possible to continue normally.

* Press any key to attempt to continue.
* Press CTRL+ALT+RESET to restart your computer. You will lose any unsaved information in all applications.

Press any key to continue
A problem has been detected and Windows has been shut down to prevent damage to your computer.

A process or thread crucial to system operation has unexpectedly exited or been terminated.

If this is the first time you've seen this stop error screen, restart your computer. If this screen appears again, follow these steps:

1. Check to make sure any new hardware or software is properly installed.
2. If this is a new installation, ask your hardware or software manufacturer for any Windows updates you might need.
3. If problems continue, disable or remove any newly installed hardware or software. Disable BIOS memory options such as caching or shadowing. If you need to use Safe Mode to remove or disable components, restart your computer, press F8 to select Advanced Startup options, and then select Safe Mode.

Technical information:

STOP: 0x0000000F (0x80000003, 0x85866DA0, 0x85866FA4, 0x800C8CAB)
Isolation
Isolation is the Key

Trustworthy Computing Base
• message passing
• virtual memory
• interrupt handling
• access control

Applications
• fault isolation
• fault identification
• IP protection
• modularity

Trusted next to Untrusted
Isolation is the Key

**Trustworthy Computing Base**
- message passing
- virtual memory
- interrupt handling
- access control

**Applications**
- fault isolation
- fault identification
- IP protection
- modularity

**Trusted next to Untrusted**

Untrusted
- Legacy Apps
- Linux Server

Trusted
- Sensitive App
- Trusted Service
- seL4

Hardware
Functional Correctness Possible

Proof

Specification

Code
**Functional Correctness Possible**

### What

**Specification**

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

### Proof

**Code**
22 Threads and TCBs

theory Tcb_A
imports CSpace_A ArchVSpace_A Schedule_A Ipc_decls_A
begin

constdefs
  set_thread_state :: obj_ref ⇨ thread_state ⇨ unit s_monad
  set_thread_state ref ts
  do
tcb ⇨ assert_opt_get t get_tcb ref;
  set_object ref yTCB ytcb (| tcb_state := ts |) zz od

defs
  suspend_def:
  suspend lazy thread
  do
  ipc_cancel thread;
  set_thread_state thread Inactive od

constdefs
  restart :: obj_ref ⇨ unit s_monad
  restart thread
  do
  state ⇨ get_thread_state thread;
  when y ¬ runnable state z t do
  ipc_cancel thread;
  od
  OR switch_to_idle_thread

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

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

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

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

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

end
*conditions apply

- Specification
- Proof
- Code
*conditions apply
*conditions apply

Assume correct:
- compiler + linker (wrt. C op-sem)
- assembly code (600 loc)
- hardware (ARMv6)
- cache and TLB management
- boot code (1,200 loc)
Proof Architecture [SOSP’09]

- Specification
- Design
- C Code Semantics

- Isabelle

- Haskell Prototype
- C Code
Proof Architecture Now

High-level properties:
- functional correctness
- integrity
- authority confinement
- non-interference
- termination
- worst-case execution time (by static analysis)
seL4: Unique Assurance

First and only general-purpose OS kernel with full functional-correctness proof – at binary level
seL4: Unique Assurance

First and only general-purpose OS kernel with full functional-correctness proof – at binary level

First and only kernel with proof of integrity and confidentiality enforcement – at binary level
seL4: Unique Assurance

First and only general-purpose OS kernel with full functional-correctness proof – at binary level

First and only kernel with proof of integrity and confidentiality enforcement – at binary level

World’s fastest microkernel on ARM architecture
seL4: Unique Assurance

First and only general-purpose OS kernel with full functional-correctness proof – at binary level

First and only kernel with proof of integrity and confidentiality enforcement – at binary level

World’s fastest microkernel on ARM architecture

Predecessor deployed on 2 billion devices
seL4: Unique Assurance

First and only general-purpose OS kernel with full functional-correctness proof – at binary level

First and only kernel with proof of integrity and confidentiality enforcement – at binary level

World’s fastest microkernel on ARM architecture

Predecessor deployed on 2 billion devices

First and only protected-mode operating-system with complete and sound timing analysis
seL4: Unique Assurance

Open Source in 2014

First and only general-purpose OS kernel with full functional-correctness proof – at binary level

First and only kernel with proof of integrity and confidentiality enforcement – at binary level

World’s fastest microkernel on ARM architecture

First and only protected-mode operating-system with complete and sound timing analysis

Predecessor deployed on 2 billion devices
Refinement Pays Off

- **Functional correctness**: 12 person years
- **Integrity+Confinement**: 10 person months
- **Non-interference**: 48 person months
- **Binary Verification**: automatic
Cost of Assurance
Cost of Assurance

• **Industry Best Practice:**
  
  • High assurance (Common Criteria EAL 6+):
    $1,000/LOC, model verification + testing, unoptimised
  
  • Low assurance (traditional embedded kernels):
    $100–200/LOC, 1–5 faults/kLOC, optimised
Cost of Assurance

• **Industry Best Practice:**
  - High assurance (Common Criteria EAL 6+):
    $1,000/LOC, model verification + testing, *unoptimised*
  - Low assurance (traditional embedded kernels):
    $100–200/LOC, 1–5 faults/kLOC, *optimised*

• **State of the Art – seL4:**
  - $400/LOC, binary-level formal proof, *optimised*
  - Estimate repeat would cost half
    - about as much as unverified predecessor Pistachio!
  - Aggressive optimisation [APSys’12]
    - much faster than traditional high-assurance kernels
    - as fast as best-performing low-assurance kernels
Cost of Assurance

- **Industry Best Practice:**
  - High assurance (Common Criteria EAL 6+):
    - $1,000/LOC, model verification + testing, optimised
  - Low assurance (traditional embedded kernels):
    - $100–200/LOC, 1–5 faults/kLOC, optimised
- **State of the Art – seL4:**
  - $400/LOC, binary-level formal proof, optimised
  - Estimate repeat would cost half
    - about as much as unverified predecessor Pistachio!
  - Aggressive optimisation [APSys’12]
    - much faster than traditional high-assurance kernels
    - as fast as best-performing low-assurance kernels
Cost of Assurance

• **Industry Best Practice:**
  - High assurance (Common Criteria EAL 6+): $1,000/LOC, model verification + testing, unoptimised
  - Low assurance (traditional embedded kernels): $100–200/LOC, 1–5 faults/kLOC, optimised

• **State of the Art – seL4:**
  - $400/LOC, binary-level formal proof, optimised
  - Estimate repeat would cost half about as much as unverified predecessor Pistachio!

  - Aggressive optimisation [APSys'12]
    - much faster than traditional high-assurance kernels
    - as fast as best-performing low-assurance kernels

Formal verification getting close to traditional kernel development.

Still too expensive for large-scale user code development.

Automation, Synthesis, Proof Generation
Next Step: Full System Assurance

DARPA HACMS Program:
- Provable vehicle safety
- Red Team must not be able to divert vehicle

Boeing Unmanned Little Bird (AH-6)

SMACCMcopter Research Vehicle
Scale
Size distribution of AFP entries in lines of proof, sorted by submission date
A Sense of Scale

Lines of Proof in Comparison

- AFP entries by submission date
- four-color theorem, Isabelle/HOL, CompCert
- Odd Order Theorem, L4.verified, Verisoft
Proof Introspection

- 500 files
- 22,000 lemmas stated
- 95,000 lemmas proved
Proof Introspection

- 500 files
- 22,000 lemmas stated
- 95,000 lemmas proved

Raf’s Observation

The introspection of proof and theories is an essential part of working on a large-scale verification development.
Proof Introspection

- 500 files
- 22,000 lemmas stated
- 95,000 lemmas proved

Raf’s Observation

The introspection of proof and theories is an essential part of working on a large-scale verification development.

- Learning Isabelle? Easy.
- Learning microkernels? Not too bad.
- Finding your way in the 400kloc proof jungle? Hard!
Maintenance

- Development of seL4 code + spec artefacts (sloc)

![Graph showing maintenance phases and development milestones.](image-url)
Maintenance

- Development of seL4 proofs (sloc)

![Graph showing maintenance phases and proof development](image_url)
Proof Development

– proof development

• decomposition of proofs over people,
• custom proof calculus,
• automating mechanical tasks, custom tactics
• proof craft
Proof Development

– proof development

• decomposition of proofs over people,
• custom proof calculus,
• automating mechanical tasks,
• proof craft

Tim’s Statement

Automating “donkey work” allows attention and effort to be focussed where most needed – but it must be done judiciously.
Proof Development

– proof development
  • decomposition of proofs over people,
  • custom proof calculus,
  • automating mechanical tasks,
  • proof craft

– challenges
  • non-local change,
  • speculative change,
  • distributed development

Tim’s Statement

Automating “donkey work” allows attention and effort to be focussed where most needed – but it must be done judiciously.
Proof Development

– proof development
  • decomposition of proofs over people,
  • custom proof calculus,
  • automating mechanical tasks,
  • proof craft

– challenges
  • non-local change,
  • speculative change,
  • distributed development

Tim’s Statement
Automating “donkey work” allows attention and effort to be focussed where most needed – but it must be done judiciously.

Matthias’ Conjecture
Over the years, I must have waited weeks for Isabelle. Productivity hinges on a short edit-check cycle; for that, I am even willing to (temporarily) sacrifice soundness.
Problems of Scale

– proof maintenance

• changes, updates, new proofs, new features
• automated regression, keep code in sync
• refactoring
• simplification
Problems of Scale

– proof maintenance

• changes, updates, new proofs, new features
• automated regression, keep code in sync
• refactoring
• simplification

Dan’s Conclusion
Verification is fast, maintenance is forever.
Research Challenges
Software vs Proof Engineering

• Is Proof Engineering a thing?
  • Google Scholar:
    • “software engineering” 1,430,000 results
Software vs Proof Engineering

- Is Proof Engineering a thing?
  - Google Scholar:
    - “software engineering” 1,430,000 results
    - “proof engineering” 564 results
Software vs Proof Engineering

• **Is Proof Engineering a thing?**

  • **Google Scholar:**
    • "software engineering" 1,430,000 results
    • "proof engineering" 564 results

  **Includes**
  "The Fireproof Building" and
  "Influence of water permeation and analysis of treatment for the Longmen Grottoes"
Proof Engineering is The Same

• Same kind of artefacts:
  • lemmas are functions, modules are modules
  • code gets big too
  • version control, regressions, refactoring and IDEs apply
Proof Engineering is The Same

• **Same kind of artefacts:**
  - lemmas are functions, modules are modules
  - code gets big too
  - version control, regressions, refactoring and IDEs apply

• **Same kind of problems**
  - managing a large proof base over time
  - deliver a proof on time within budget
  - dependencies, interfaces, abstraction, etc
Proof Engineering is Different

• But: New Properties and Problems
Proof Engineering is Different

• But: New Properties and Problems
  • Results are checkable
    • You know when you are done!
  • No testing
  • 95% proof: no such thing
Proof Engineering is Different

• But: New Properties and Problems
  • Results are checkable
    • You know when you are done!
    • No testing
  • 95% proof: no such thing
  • More dead ends and iteration
Proof Engineering is Different

• But: New Properties and Problems
  • Results are checkable
    • You know when you are done!
    • No testing
    • 95% proof: no such thing
  • More dead ends and iteration
  • 2nd order artefact
    • Performance less critical
    • Quality less critical
    • Proof Irrelevance
Proof Engineering is Different

• But: New Properties and Problems
  • Results are checkable
    • You know when you are done!
    • No testing
    • 95% proof: no such thing
  • More dead ends and iteration
  • 2nd order artefact
    • Performance less critical
    • Quality less critical
    • Proof Irrelevance
  • More semantic context
    • Much more scope for automation
Deliver within Time and Budget

• **Estimation:**
  - time and effort
  - how precisely, with which confidence?
  - how early?
Deliver within Time and Budget

- **Estimation:**
  - time and effort
  - how precisely, with which confidence?
  - how early?

- **Size of artefacts**
  - easier to predict?
  - related to effort?
Deliver within Time and Budget

• **Estimation:**
  • time and effort
  • how precisely, with which confidence?
  • how early?

• **Size of artefacts**
  • easier to predict?
  • related to effort?

• **Complexity**
  • from initial artefacts?
  • which influence?
Proof Engineering Tools

• User Interface

• could proof IDEs be more powerful than code IDEs?

• more semantic information

• proof completion and suggestion?
Proof Engineering Tools

• **User Interface**
  • could proof IDEs be more powerful than code IDEs?
  • more semantic information
  • proof completion and suggestion?

• **Refactoring**
  • less constrained, new kinds of refactoring possible, e.g.
    • move to best position in library
    • generalise lemma
    • recognise proof patterns
Proof Patterns

• Large-scale Libraries
  • architecture:
    • layers, modules, components, abstractions, genericity
  • proof interfaces
  • proof patterns
Proof Patterns

• Large-scale Libraries
  • architecture:
    • layers, modules, components, abstractions, genericity
  • proof interfaces
  • proof patterns

• Technical Debt
  • what does a clean, maintainable proof look like?
  • which techniques will make future change easier?
  • readability important? is documentation?
Proof Engineering “Laws”

• Are there Proof Engineering Laws?
Proof Engineering “Laws”

• Are there Proof Engineering Laws?
  • Proofs always become larger and more complex over time. (from Cope’s rule)
Proof Engineering “Laws”

• Are there Proof Engineering Laws?
  • Proofs always become larger and more complex over time. (from Cope’s rule)
  • Adding manpower to a late proof project makes it later. (from Brooks’ law)
Proof Engineering “Laws”

• Are there Proof Engineering Laws?
  • Proofs always become larger and more complex over time. (from Cope’s rule)
  • Adding manpower to a late proof project makes it later. (from Brooks’ law)
  • You cannot reduce the complexity of a given proof beyond a certain point. Once you’ve reached that point, you can only shift the burden around. (from Tesler’s law)
Proof Engineering “Laws”

• Are there Proof Engineering Laws?
  
  • Proofs always become larger and more complex over time.  
    (from Cope’s rule)
  
  • Adding manpower to a late proof project makes it later.  
    (from Brooks’ law)
  
  • You cannot reduce the complexity of a given proof beyond a  
    certain point. Once you’ve reached that point, you can only shift  
    the burden around.  
    (from Tesler’s law)
  
  • Are they true?
Summary

**seL4**

- Full verification. Full performance.
- Already cost effective for high assurance.
- Going open source and open proof in 2014.
Summary

**seL4**

- Full verification. Full performance.
- Already cost effective for high assurance.
- Going open source and open proof in 2014.

**Proof Engineering**

- Should become a research discipline.
- Work has started. A lot more to be done.
http://sel4.systems