Formally Verified Operating Systems

SINGULARITY AND SEL4

Outline

Formal Verification & Type Systems

Singularity

- Software-Isolated Processes
- Contract-Based Channels
- Manifest-Based Programs
- Formal Verification

seL4

- Assumptions
- Design Path
- Costs of Verification

Formal Verification in a nutshell

Create a collection of **rules**

Claim/Prove that those rules describe certain **properties**

Check whether/Prove that something adheres to those rules
- If yes, then that something has the above properties

Properties may be very weak or very strong
- Weak properties: easy to prove
- Strong properties: may not be provable
  - Rice's theorem: it is impossible to prove anything non-trivial for arbitrary programs

Formal Verification Example

Hoare Logic:

$$\{P\}\ s\ \{Q\}$$

```
fun tenmod (mod) { mod ≠ 0 }
returns ret { ret = 10 % mod }
is
    return 10 % mod;
end;
```

$$\{P_1\}\ x \coloneqq 5; \{P_1 \setminus (x = \cdots) \cup (x = 5)\}$$

Type Systems

"The world's best lightweight formal method" (Benjamin Pierce)

Mainly for safety properties

Static type-checking

- Proving properties of your program
- May need annotations from the programmer

Almost all programming languages have type systems

- But the static guarantees vary a lot

Annotations

```
fun factorial(n : int) { n > 0 }
returns r : int { r == n! } is
      if ( n == 1 )
            return 1;
      else
            return n * factorial( n - 1 );
end;
```

Note

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;
}
```

Postcondition: $A_{post} = B_{pre} \wedge B_{post} = A_{pre}$

Singularity – Takeaway Goal

PL techniques can make kernel & programs a lot safer

Safe programs can run in kernel-space

IPC is really fast when programs run in kernel-space

(Reasonable?) restrictions on programs make the job of the OS much easier

Singularity - Authors

Galen Hunt

- University of Rochester (PhD, 1998)
- Created prototype of Windows Media Player
- Led Menlo, Experiment 19 and Singularity projects

Jim Larus

- UC Berkeley (PhD, 1989)
- University of Wisconsin-Madison (1989-1999)
- University of Washington (2000-)
- Microsoft Research (1997-)
    - eXtreme Computing Group (2008-2012)

Singularity – Design Goals

- A dependable system
  - Catch errors as soon as possible

Compile Time > Installation Time > Run Time

Design Time Load Time

Singularity - 3 Core Ideas

Software-Isolated Processes (SIPs)

Contract-Based Channels

Manifest-Based Programs

Software-Isolated Processes

Programs written in a memory-safe language

- Cannot access data of other processes

Cannot dynamically load code

Can only communicate with other processes via messages

- Sender and receiver always known

Kernel respects the above limitations, too

Programs run in kernel-space

Every process has its own runtime and GC

Contract-enforcing channels

The only way of inter-process communication

Endpoints always belong to specific threads

- Can be passed to other programs via channels

Sending data also transfers ownership of data

- Process cannot access data anymore after sending it

Adherence to communication protocol statically verifiable

Contract-enforcing channels

```
contract C1 {
   in message Request(int x) requires x>0;
   out message Reply(int y);
   out message Error();
   state Start: Request?
      -> (Reply! or Error!)
      -> Start;
}
```

Source: Singularity Technical Report, Hunt et al. (MSR-TR-2005-135)

Manifests

Manifests describe :

- the complete program code
  - The program itself
  - All dependencies
- the resources a program might access
- the communication channels it offers

Can be statically verified

Guide install-time compilation

Manifests

```
<manifest>
<application identity="S3Trio64" />
<assemblies>
<assembly filename="S3Trio64.exe" />
<assembly filename="Namespace.Contracts.dll" version="1.0.0.2299"/>
<assembly filename="Io.Contracts.dll" version="1.0.0.2299" />
<assembly filename="Corlib.dll" version="1.0.0.2299" />
<assembly filename="Corlibsg.dll" version="1.0.0.2299" />
<assembly filename="System.Compiler.Runtime.dll“ version="1.0.0.2299" />
<assembly filename="Microsoft.SingSharp.Runtime.dll“ version="1.0.0.2299" />
<assembly filename="ILHelpers.dll" version="1.0.0.2299" />
<assembly filename="Singularity.V1.ill" version="1.0.0.2299" />
</assemblies>
<driverCategory>
<device signature="/pci/03/00/5333/8811" />
<ioMemoryRange index="0" baseAddress="0xf8000000"
rangeLength="0x400000" />
<ioMemoryRange baseAddress="0xb8000" rangeLength="0x8000“ fixed="True" />
<ioMemoryRange baseAddress="0xa0000" rangeLength="0x8000“ fixed="True" />
<ioPortRange baseAddress="0x3c0" rangeLength="0x20" fixed="True" />
<ioPortRange baseAddress="0x4ae8" rangeLength="0x2" fixed="True" />
<ioPortRange baseAddress="0x9ae8" rangeLength="0x2" fixed="True" />
<extension startStateId="3" contractName="Microsoft.Singularity.Extending.ExtensionContract"
endpointEnd="Exp“ assembly="Namespace.Contracts" />
<serviceProvider startStateId="3" contractName="Microsoft.Singularity.Io.VideoDeviceContract"
endpointEnd="Exp"assembly="Io.Contracts" />
</driverCategory>
...
</manifest>
```

Source: Singularity Technical Report, Hunt et al. (MSR-TR-2005-135)

Verification

Mostly safety properties

- Safe memory access
- Guaranteed by the type system

Support for contract-based verification

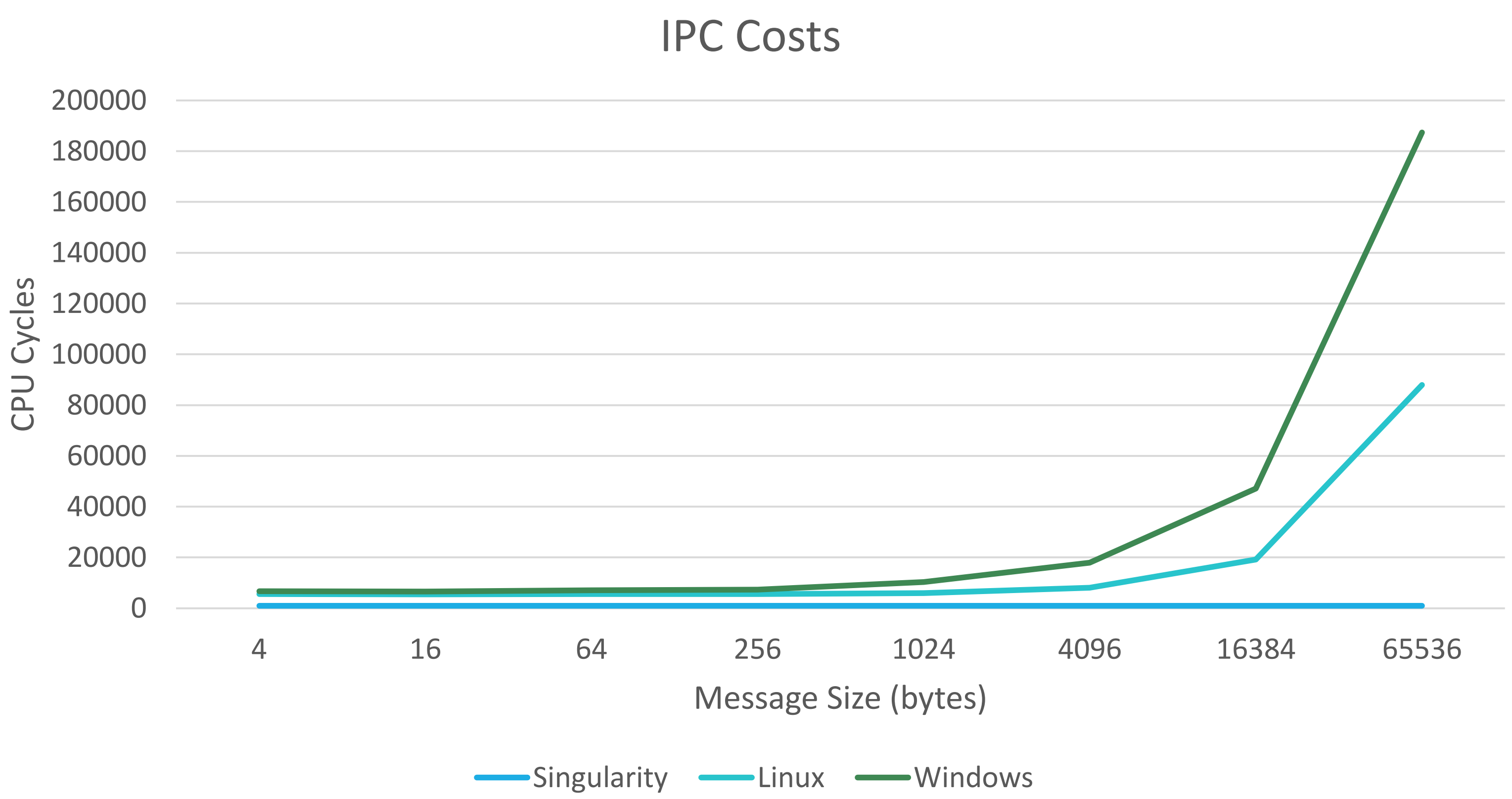
- Enables verification of functional correctness
- Not ubiquitously applied in kernel
- Some parts are checked
  - Channel contracts
  - Manifests

Benefits of safety properties

	Cost (CPU Cycles)			
	Singularity	FreeBSD	Linux	Windows
Read cycle counter	8	6	6	2
ABI call	87	878	437	627
Thread yield	394	911	906	753
2 thread wait-set ping pong	1,207	4,707	4,041	1,658
2 message ping pong	1,452	13,304	5,797	6,344
Create and start process	300,000	1,032,000	719,000	5,376,000

Source: Singularity Technical Report, Hunt et al. (MSR-TR-2005-135)

Singularity's Money Graph



Source of Data: Sealing OS Processes to Improve Dependability and Safety, Hunt et al., EuroSys 2007

Takeaway

PL techniques can make kernel & programs a lot safer

Safe programs can run in kernel-space

IPC is really fast when programs run in kernel-space

(Reasonable?) restrictions on programs make the job of the OS much easier

Discussion

Can systems programmers live without C?

Is the sharing of data between processes really not important?

seL4 – Takeaway Goal

Functional verification of microkernels is possible

Performance of verified kernels can be OK

BUT:

Verification is a colossal effort

Still needs to assume compiler correctness (➔ huge trusted base)

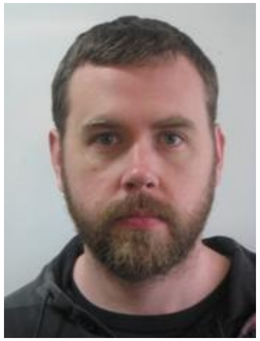
seL4 - Authors


Gerwin Klein


Kevin Elphinstone


Gernot Heiser

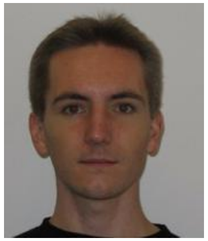

June Andronick


David Cock


Philip Derrin


Dhammika Elkaduwe


Kai Engelhardt


Rafal Kolanski


Michael Norrish


Thomas Sewell


Harvey Tuch


Simon Winwood

seL4 – Project Leaders

Gerwin Klein
- TU Munich (PhD)
- University of New South Wales
- Does not put a CV on his webpage

Gernot Heiser
- ETH Zurich (PhD, 1991)
- University of New South Wales
- Created Startup “Open Kernel Labs” to sell L4 technology
- Collaborated with Jochen Liedtke (L4)

Kevin Elphinstone
- University of New South Wales
- Does not put a CV on his webpage
- Collaborated with Jochen Liedtke (L4)

Secure L4 – Design Goal

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

How to design kernel + spec?

Bottom-Up-Approach:

Concentrate on low-level details to maximize performance

Problem:

Produces complex design, hard to verify

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;
}
```

Postcondition: $A_{post} = B_{pre} \wedge B_{post} = A_{pre}$

How to design kernel + spec?

Top-Down-Approach:

Create formal model of kernel

- Generate code from that

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)

Concurrency is a problem

Multiprocessors not included in the model

- seL4 can only run on a single processor

Interrupts are still there

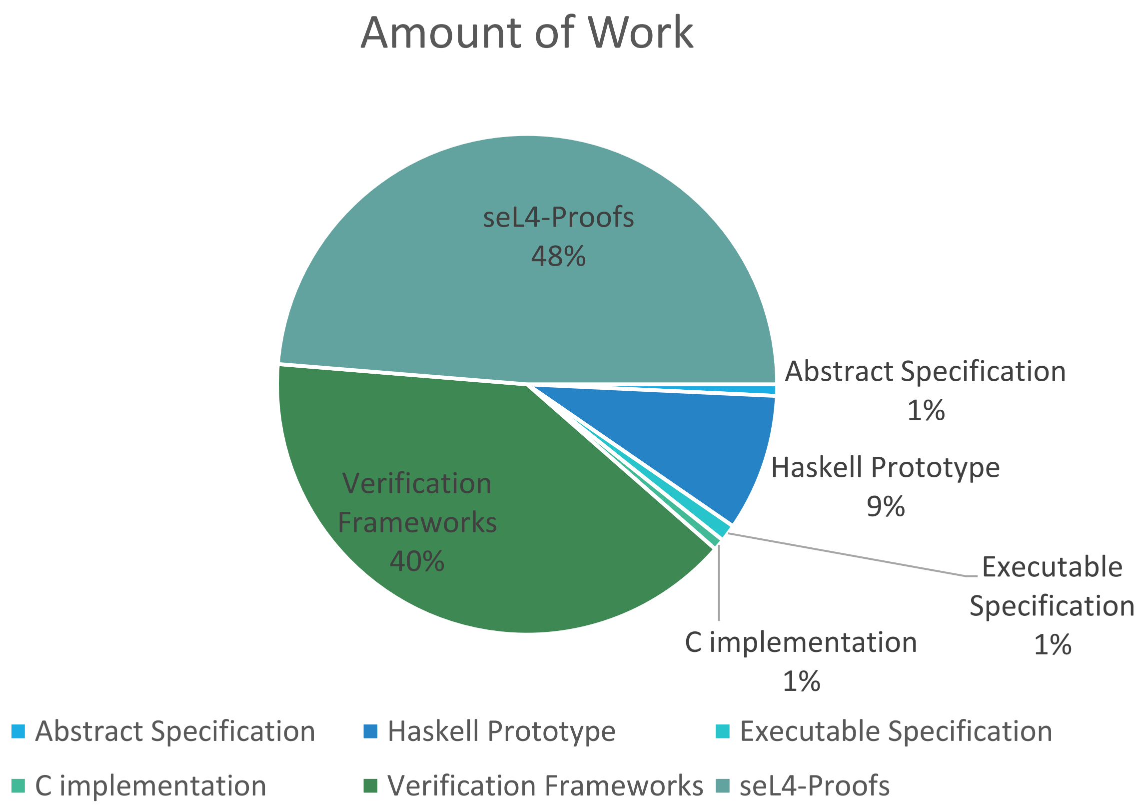
- Yield points need to establish all system invariants

Cost of Verification

	Haskell/C LOC	Isabelle LOC	Invariants	Proof LOP
abst.	—	4,900	~ 75	110,000
exec.	5,700	13,000	~ 80	55,000
impl.	8,700	15,000	0	

Source: seL4, Klein et al.

Cost of Verification



Source of Data: seL4, Klein et al.

Takeaway

Functional verification of microkernels is possible

Performance of verified kernels can be OK

BUT:

Verification is a colossal effort

Still needs to assume compiler correctness (➔ huge trusted base)

Discussion

Is proving functional correctness worth the effort?

Singularity vs. seL4

Goal

Singularity	seL4
A verifiably safe system. Kernel should fail “safely” when an error occurs.	A verifiably correct system. There just should not be any errors.

Ease of Verification

Singularity	seL4
Most guarantees come for free Annotations and contracts can give more guarantees	Several person-years just for proving about 80 invariants.

Perspective

Lots of room between Singularity and seL4

- I.e.: more parts of Singularity can be verified for functional correctness

Both are verified microkernels

- Good Isolation → additional components can be verified independently