Building Trustworthy Systems on seL4

Ihor Kuz

seL4 Summit: 25 September 2019

www.data61.csiro.au

Overview

- What is a Trustworthy System?
- What does seL4 provide?
  - seL4 kernel
  - Other seL4 platform tools
- How to build a Trustworthy System
  - Steps to trustworthiness
- Example

What is a Trustworthy System?

A system where the the Trusted Computing Base is worthy of the trust put into it

- Trusted Computing Base (TCB)
  - Parts of system that ***must be trusted*** to maintain ***safety and security properties***
  - If ***TCB fails*** then safety and security of the ***system can be compromised***
  - Consists of *trusted (critical) components*
- Trustworthiness = Confidence that components:
  - Do what they are supposed to do
  - Cannot be compromised or subverted
  - Will not fail
- Assurance
  - What assurance do you have that system is trustworthy?
  - Ideal: have high-assurance that all trusted components are trustworthy

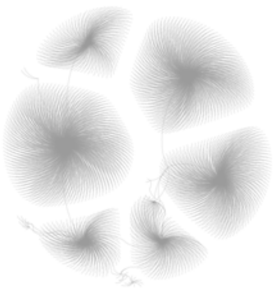
What Does seL4 Provide?

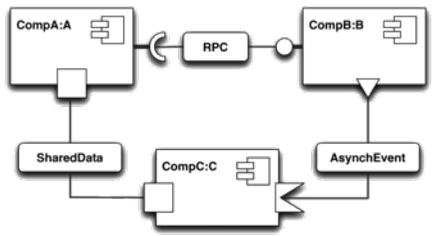

seL4
seL4
seL4
seL4
seL4
seL4
seL4
seL4
seL4

seL4 Kernel

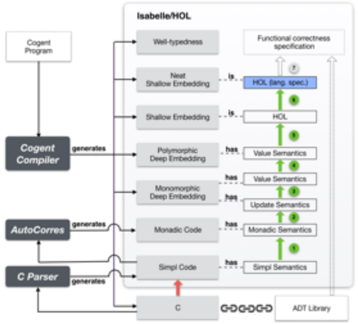
- Functional correctness
  - The implementation does exactly what the specification says
    - No more, no less
  - C code, Binary
- Security properties
  - Confidentiality
  - **Integrity**
  - Availability
- WCET
  - Research quality
- Restrictions
  - Assumptions (including hardware model and correctness)
  - Kernel init
  - Platforms and features
  - Correct user-level setup

Other seL4 Platform Tools





- CapDL
  - User-level initialisation
  - Formally verified (in progress)
- CAmkES
  - Component platform
  - Verified (in progress) CAmkES to CapDL mapping
    - CapDL has same security (data flow) properties as componentised system model
  - Verified glue code (in progress)
- Cogent programming language (in progress)
  - Type system, proof generation
- CakeML programming language support (in progress)
  - Verified compiler
- Rust programming language support
  - Type system, memory safety



What (useful) Guarantees do we get?

- Integrity
  - Requires correct user-level setup
- Confidentiality
  - Basic confidentiality (others can't read your memory)
  - Non-interference (specific setup: domain scheduler, strict partitioning)
  - Time protection (in progress)
- No unintended data flows
  - CAmkES architecture correctly implemented
- Kernel protections can't be bypassed
  - No bugs (in verified part & assumptions)
  - No kernel vulnerabilities to exploit to bypass Integrity, confidentiality guarantees

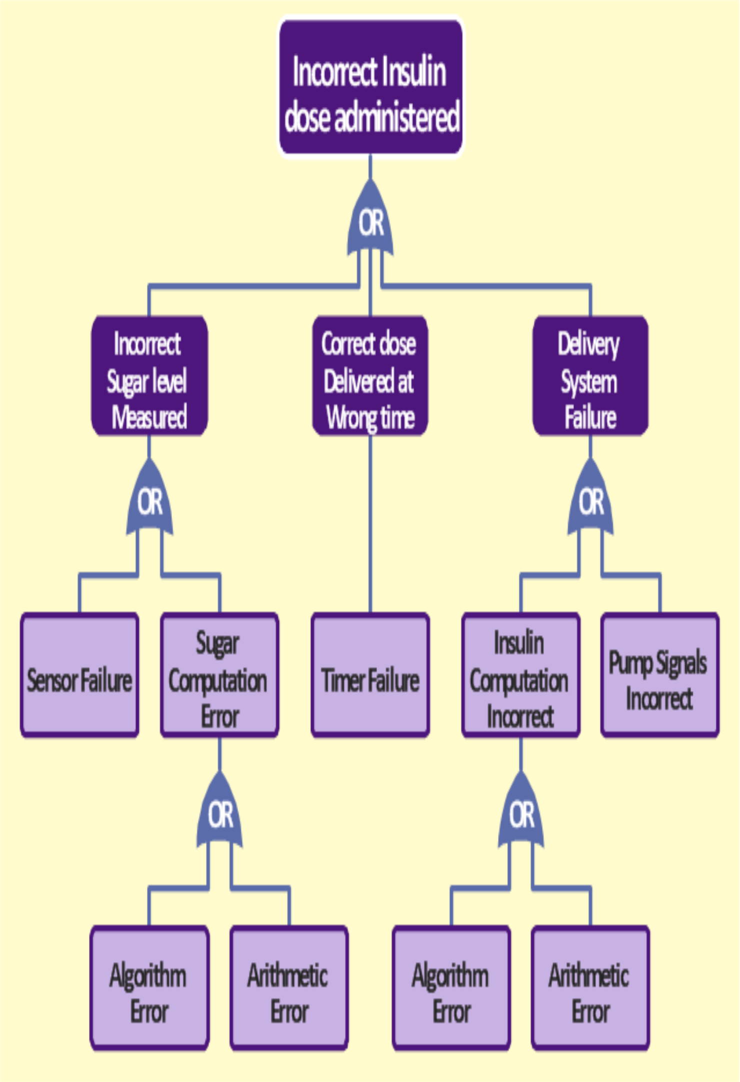
How to Build a Trustworthy System

Steps to Trustworthiness

1. Determine Safety and/or Security Requirements
2. Architect
3. Implement
4. Validate and Verify
5. Repeat
6. Profit!

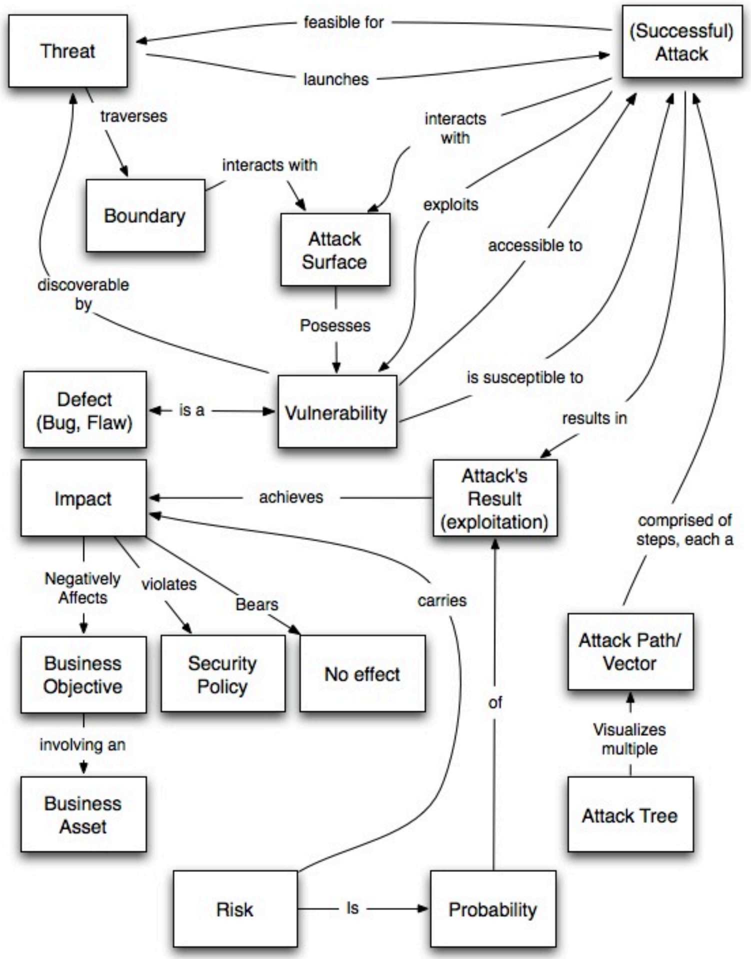
1. Determining Safety Requirements

- Functional Safety
  - Functions required for system safety (to mitigate risks and hazards)
  - Must have correct execution and behaviour
- Identify Hazards
  - Hazard analysis
  - Fault Trees
- Determine safety-critical functions
  - To control hazards
  - Address failure modes (hardware, software, human, system)
  - Additional components in the system
  - Functional path requirements
  - Functionality and sub-system boundaries



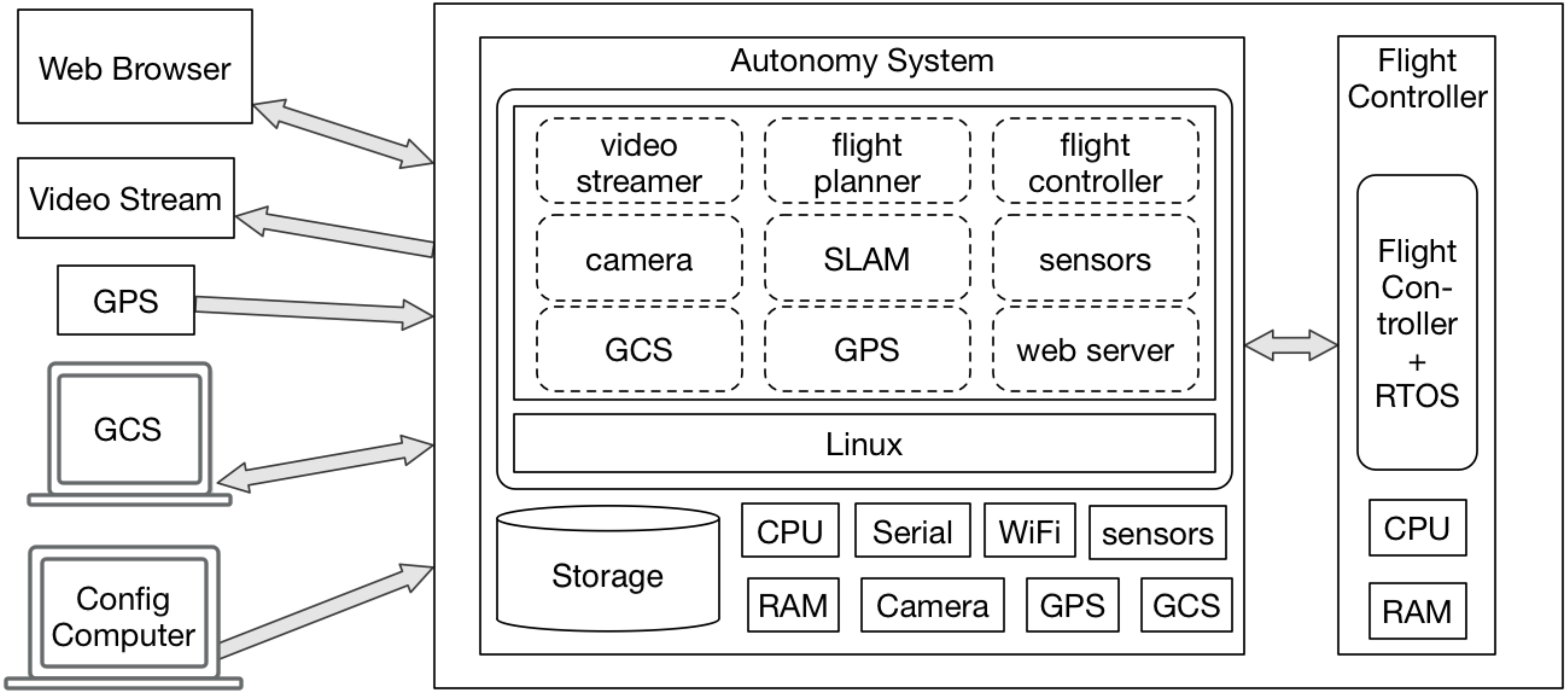
1. Determining Security Requirements

- Threat modeling
- Identify threats
  - Bad things that can happen to assets
- Discover vulnerabilities and exploits
  - Vulnerability: weakness in system or code
  - Exploit: way to take advantage of vulnerability
- Develop attacks
  - Realisation of a threat
  - Apply exploits
- "Shall not" requirements
  - Ensure that attacks are not possible



Example: Autonomous UAV

Example: Autonomous UAV



Example: Threat Model: Vulnerabilities/Exploits

- Communication:
  - Vulnerabilities: plain-text, poor authentication, poor encryption, poor resource management
  - Exploits: sniff messages, man-in-the-middle, spoof messages, replay messages , DoS
- Data:
  - Vulnerabilities: plain-text, poor encryption, poor integrity controls, poor isolation
  - Exploits: access data (read/write), bypass authentication/authorisation
- Code
  - Vulnerabilities: poor integrity controls, poor isolation
  - Exploits: load malicious code, modify code, read code
- General
  - Vulnerabilities: code bugs, poor authentication, poor authorisation
  - Exploits: access data, modify control flow, run arbitrary code, crash, bypass authentication/authorisation

Example: Threat Model: Some Attacks

- Steal Vehicle
  - exploit vulnerability in web server to run arbitrary code on web server,
  - **exploit vulnerability in OS to run code in privileged mode**,
  - modify code to cause the vehicle to fly to incorrect location,
  - wait for it there, then take vehicle.
- Steal collected data
  - exploit vulnerability in GCS component by sending malicious RF communication
  - causing it to run arbitrary code in the GCS component
  - **exploit vulnerability to elevate privilege to root**,
  - run code to read collected mission data,
  - send it out over WiFi to third party
- Steal encryption keys
  - modify 3rd party library used in GPS component to include specific attack code,
  - **exploit vulnerability to access storage at elevated privilege level**,
  - read keys from storage,
  - insert them into web server content files,
  - monitor web server and read keys from web server when they appear.

2. Architecting: Principles

- Minimal TCB
  - Minimise number of trusted components
  - Minimise size of trusted components
- Minimise attack surface
  - Minimise (superfluous) functionality
- Least privilege
  - Let components access only what they need
- Separation of Duties
- Defense in depth
  - Number of exploits needed to reach a goal
- Auditing

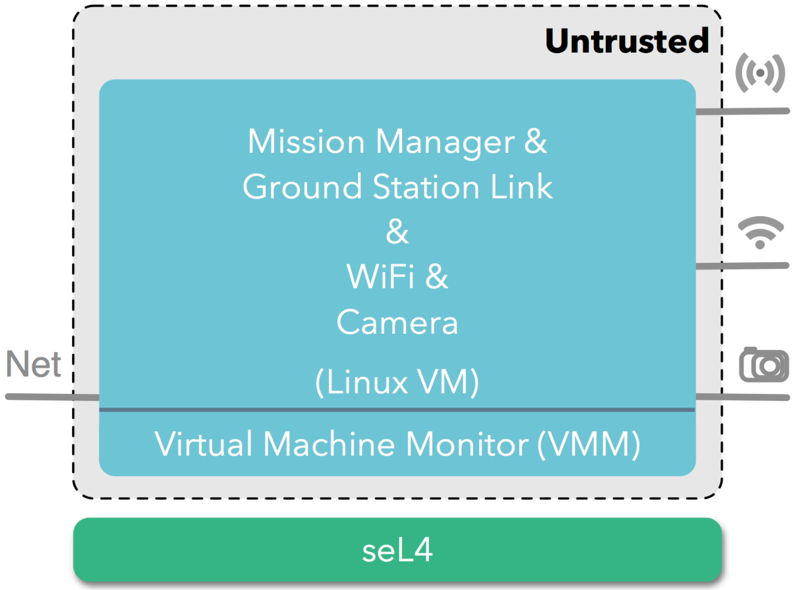
2. Architecting: Patterns

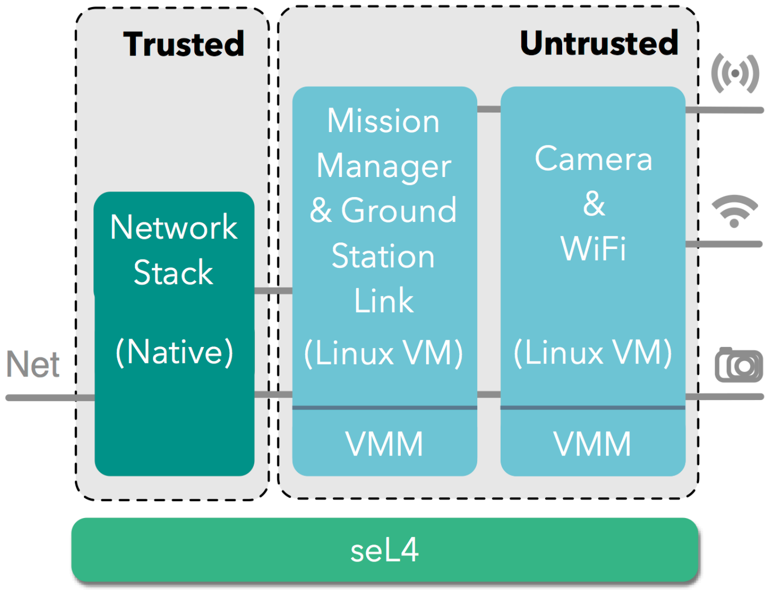
- Partition data
  - Based on who accesses it (e.g. not a single file-system)
- Split stacks
  - Horizontal (e.g. FS vs storage, network stack vs ethernet)
- Split data flows
  - Vertical (e.g. separate network streams)
- Encrypt data
  - At rest (e.g. collected data)
  - In motion (e.g. telemetry data sent to ground station)
- Isolate Cryptography
  - Isolated component storing and accessing keys

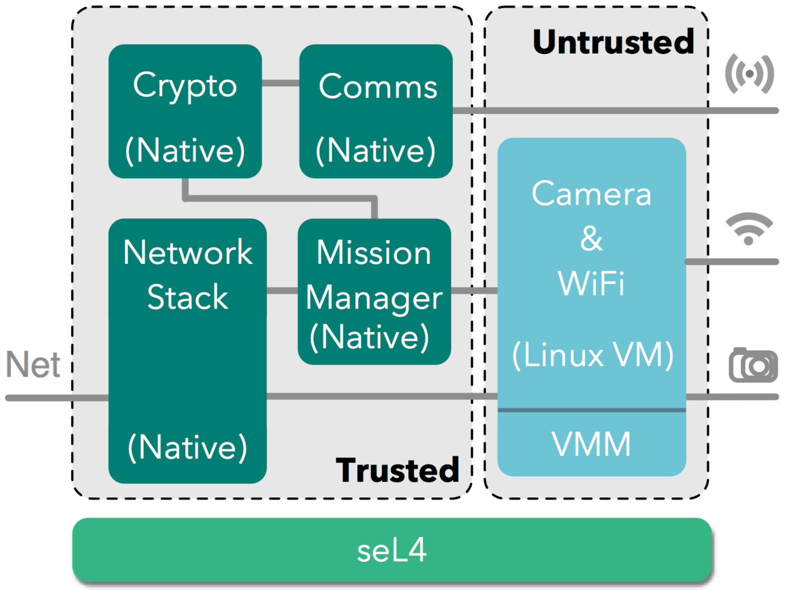
2. Architecting: Mechanisms

- Isolation
  - Decompose system into components
  - Use seL4 isolation to protect components from each other
  - Critical vs non-critical components
- Filter
  - Sanitise inputs and outputs
- Monitor
  - Monitor component outputs
  - Corrective action when there’s a problem
- Static vs dynamic systems
  - We know how to architect static systems to provide isolation guarantees
  - Dynamic systems are still “Research in progress”

3. Implementing: Cyber Retrofit







First put all of the existing software inside a VM running on seL4

⬇

No security benefit yet, simply showing that seL4 runs on the target platform and that all the software can run virtualised

Then start pulling some trusted components out of the VM to run natively on seL4

⬇

Some security benefit: compromise in VM cannot propagate to trusted component

Full security architecture, with all trusted components running as a seL4 components

⬇

Important security benefit: All components run isolated in a container, only the VM is still vulnerable

3. Implementing: Harden Components

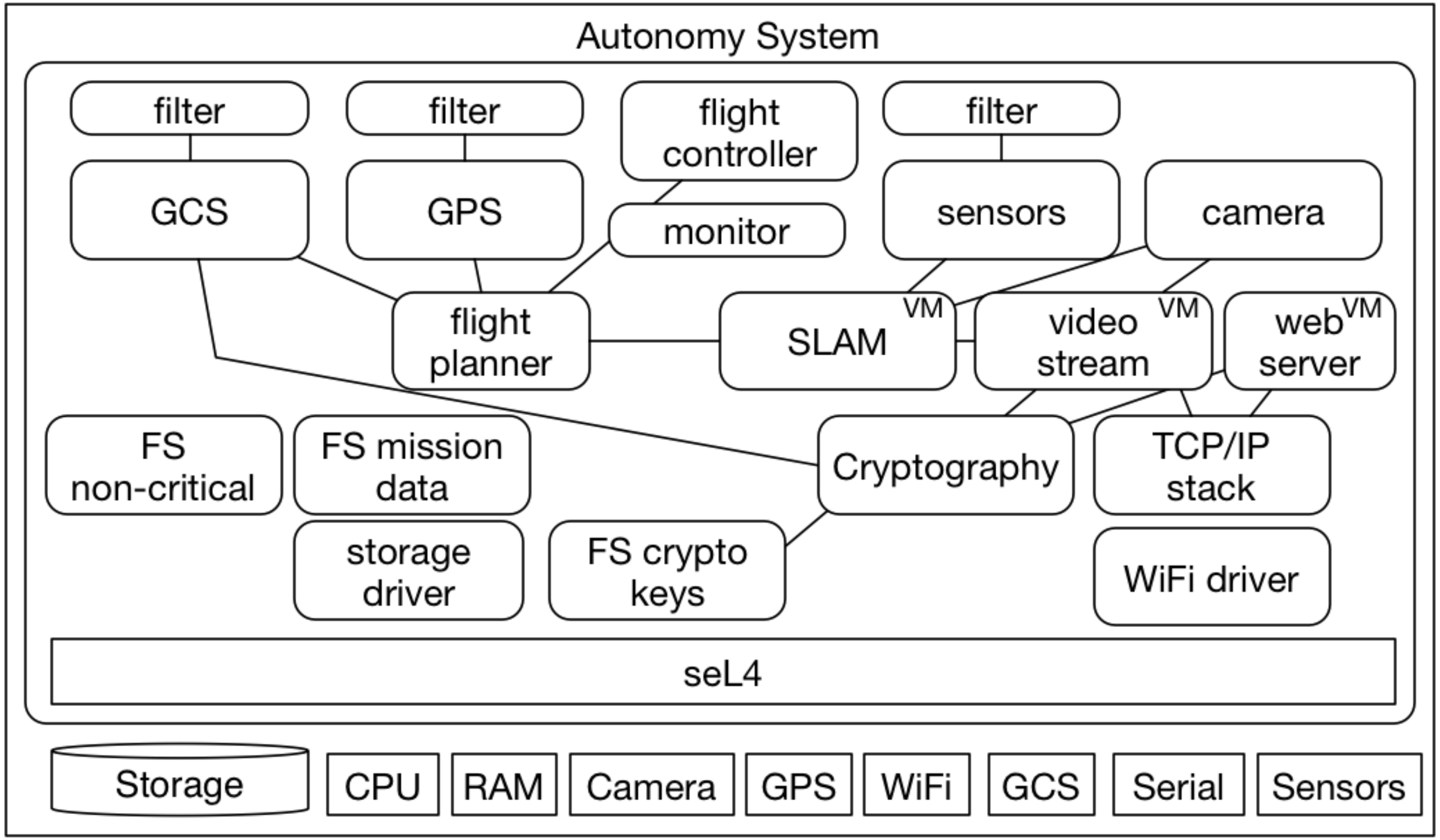
- Good Programming
  - Power of 10
  - Secure coding practices
- Tools to find weaknesses
  - Static analysis
  - Dynamic analysis
- Good Programming Languages
  - Rust, Haskell, etc.
- Verification
  - Manual
  - Semi-Automated
  - Automated

4. Validating and Verifying

- Attack and fault analysis (‘threat modeling’, attack trees, fault trees)
- Testing (including Fuzzing)
- Software assessment/auditing
- Formal verification of components
  - Theorem prover
  - Model checking
  - Generated Proofs (e.g. CakeML, CAmkES)
- Formal verification of architecture
  - Architecture level Properties
  - Architecture implementation
  - Infoflow
- Red team

Example Continued

Autonomous UAV: More Secure Architecture



Threat Model: Example Attacks

- Steal Vehicle
  - exploit vulnerability in web server to run arbitrary code on web server,
  - exploit vulnerability in OS to run code in privileged mode,
  - modify code to cause the vehicle to fly to incorrect location,
  - wait for it there, then take vehicle.
- Steal collected data
  - exploit vulnerability in GCS component by sending malicious RF communication
  - causing it to run arbitrary code in the GCS component,
  - exploit vulnerability to elevate privilege to root,
  - run code to read collected mission data,
  - send it out over WiFi to third party
- Steal encryption keys
  - modify 3rd party library used in GPS component to include specific attack code,
  - exploit vulnerability to access storage at elevated privilege level,
  - read keys from storage,
  - insert them into web server content files,
  - monitor web server and read keys from web server when they appear.

Summary

- seL4 is **not** magic security fairy dust!

1. Requirements: Understand what you need
2. Architect: Take advantage of seL4's isolation properties
3. Implement:
   - Cyber Retrofit
   - Harden critical components (verification!)
4. Verify and Validate: Make sure you got it right
5. Repeat

- Example: Autonomous UAV