Trusted Build

Why Trusted Build?

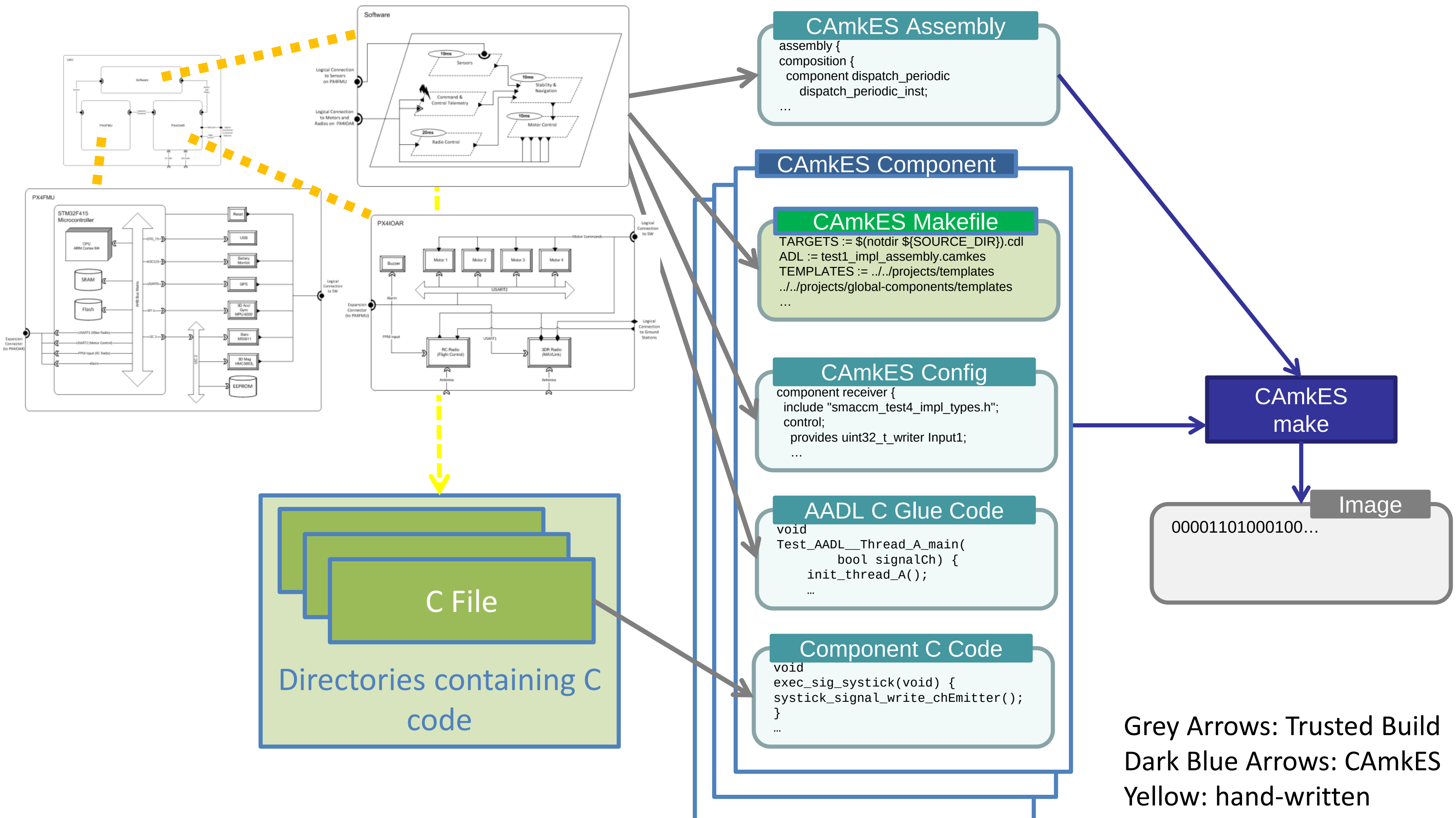
Why Trusted Build?

- Ensure fidelity between models and system image
  - Proofs are over architectural models
    - Information flow between processes and threads
    - Well-formedness of architecture: scheduling, memory limits and safety, etc.
  - Trusted build *generates* system image from architectural model
    - Prevents stupid errors
      - Mismatches on unit types between modules [Mars Polar Lander]
      - Mismatches on alignment of data, data representation, and data location

Trusted Build Features

- OS Support: eChronos, CAmkES/seL4, vxWorks, linux
- Process/thread support
  - Periodic, sporadic, hybrid dispatch models
  - User provides entrypoints which are invoked by middleware
- Comm Support: Shared Memory, event-, event data-, data-port communications, RPC.
  - Cross thread / process
  - For data-port, cross-VM
  - Interrupts
- Support for interaction with external code
  - External types
  - External threads

CAmkES/seL4 Build process

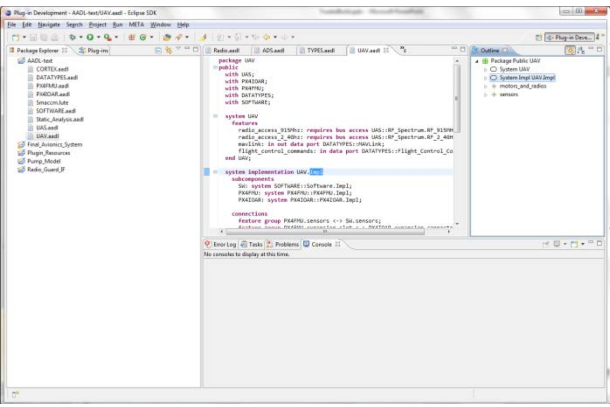


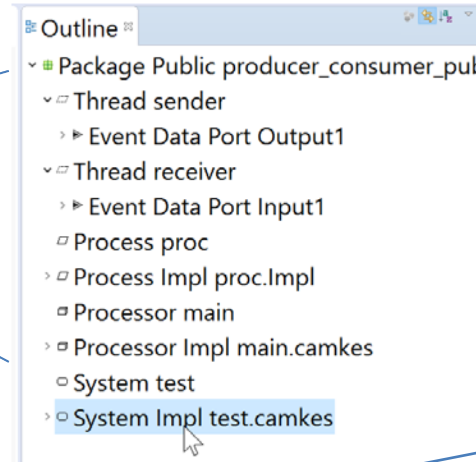
Grey Arrows: Trusted Build
Dark Blue Arrows: CAmkES
Yellow: hand-written code/models

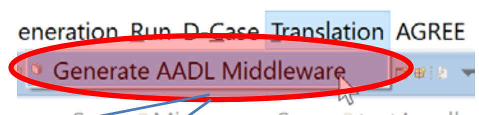
Why *Trusted* Build?

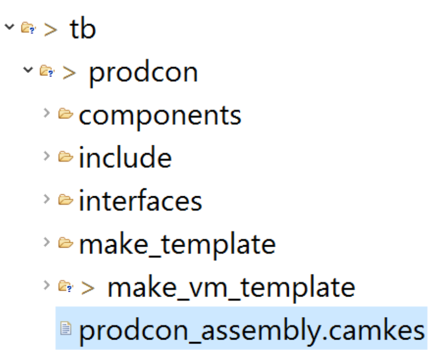
- OSATE plug-in for Eclipse
- Written in Java using String Template library
- No high-assurance pedigree, except:
  - Implementations of some connectors (Mailboxes for Dataports (seL4/Linux), Connector components for data/event ports (seL4) have associated proofs
  - Code has been inspected by multiple groups, including Red Team, looking for security flaws
- Eventual goal is to tie into CAmkES/seL4 connector proofs
  - Isabelle/HOL

Generating Code:









Components/IDL Files

```
...
assembly {
   composition {
      component dispatch_periodic dispatch_periodic_inst;

      component TimeServerKZM time_server;
      // Component instances for all AADL-defined threads
      component sender sender_inst;
      component receiver receiver_inst;

      // Port declarations for active threads
      connection seL4TimeServer tb_sender_periodic_dispatcher_timer(from 
sender_inst.tb_timer, to time_server.the_timer);
      connection seL4Notification tb_sender_periodic_dispatcher_echo_int(from 
dispatch_periodic_inst.sender_periodic_dispatcher, to sender_inst.tb_timer_complete);
```

Assembly File

AADL Modeling Issues

- Proper way of modeling interrupts / drivers
  - Interrupts have 1st and 2nd level-handlers
  - Strictly speaking, 1st level handler does not run on associated AADL thread.
- "External" threads and types that do not match the AADL dispatch paradigm
  - external types / threads
- VMs / Containers
  - Maintaining VM memory / device mappings
  - Schedulability?

ISR ports and Drivers

thread sender
features
Input1: **in event port {**
  Source_Text => ("user_code/f1.c", "user_code/f2.c", "user_code/f3.h");
  TB_SYS::Is_ISR => **true;**
  TB_SYS::First_Level_Interrupt_Handler => "timer_flih";
  Compute_Entrypoint_Source_Text => "timer_slih";
  TB_SYS::Signal_Name => "irq";
  TB_SYS::Signal_Number => 27;
  TB_SYS::Memory_Pages => ("mem", "0x53F98000:0x1000");
  TB_SYS::Sends_Events_To => "{{2 Output1}}";
};
...
properties
    Dispatch_Protocol => Sporadic;
    Initialize_Entrypoint_Source_Text => "initialize_timer";
    TB_SYS::Thread_Type => Active ;
 ...
end sender ;

ISRs have a first-level and second-level handler

Each ISR must have a unique signal name and number

Each ISR port is associated with a list of named, sized memory regions:
“mem” is the name
0x53F98000 is the location
0x1000 is the size

AADL Engineering Issues

- Aligning with Code Generation Annex
  - We started developing our tools in 2012 – concurrent with CGA development.
- Factoring tool to allow multiple implementations of comm primitives
  - OS-specific
  - Team(!) specific (mailboxes, queue-servers)
  - Specializations for cross-VM communications
- Running OSATE "headless"
  - For code generation: want single command build
  - For AGREE / Resolute analysis
  - I think that this will be a standard mechanism for use.

AADL Code Gen Annex Questions

Is there a complete description of the desired interactions between entrypoints and communication functions or complete examples?

Ports:

- Are send/receive interfaces typesafe? What happens if the wrong type is injected into a call?
- What happens if you call send_output() and receive_input() multiple times in a dispatch?
- What happens if you use a dataport that does not support tracking number of updates?
  - E.g. shared memory

Threads:

- How do you manage interrupts?
- How are periodic dispatches managed?

Wrap Up

- Trusted Build is a tool for generating system implementation skeletons from AADL models
- Goal is to make AADL analysis meaningful against generated code
- It supports a subset of AADL
  - This is intentional: goal is to have small enough code base to have confidence in result
- Can target multiple Oses: seL4, VxWorks, eChronos, linux
  - Focus for this talk was on CAmkES/seL4

Backup

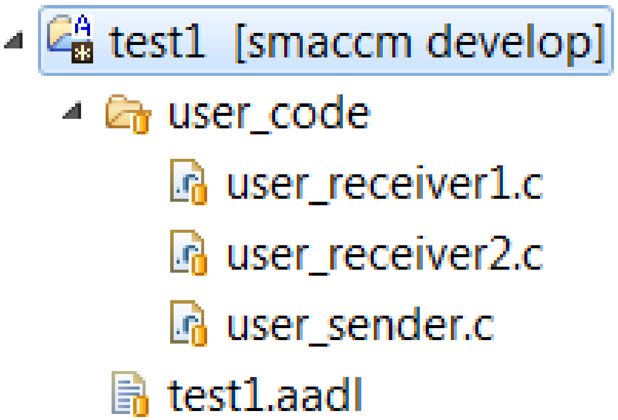
DEMO: Producer / Consumer

- Producer / Consumer
  - Deployed to CAmkES/seL4
  - Producer periodically emits int32_t
  - Consumer is event-dispatched and consumes it.
- To compile the code:
  - Copy "template" makefile to top-level tb directory
  - Copy all code to camkes/apps directory
  - Add application directory to Kconfig.
- To run the image:
  - Type:
    qemu-system-arm -M kzm -nographic -kernel images/capdl-loader-experimental-image-arm-imx31

Structure of Generated Code

- CAmkES OS Configuration
  - .camkes component files
  - CAmkES assembly file for top-level system
  - IDL files describing RPCs
- Generated C Middleware Code
  - tb_<component>.c
    - Implementation of middleware services
  - tb_<component>.h
    - Description of API between user-level AADL code and TB code.
- Template Makefile for system build
  - Works for simple build procedures
  - However, for more complex system generations, likely you will want to replace it with a more “full-featured” make

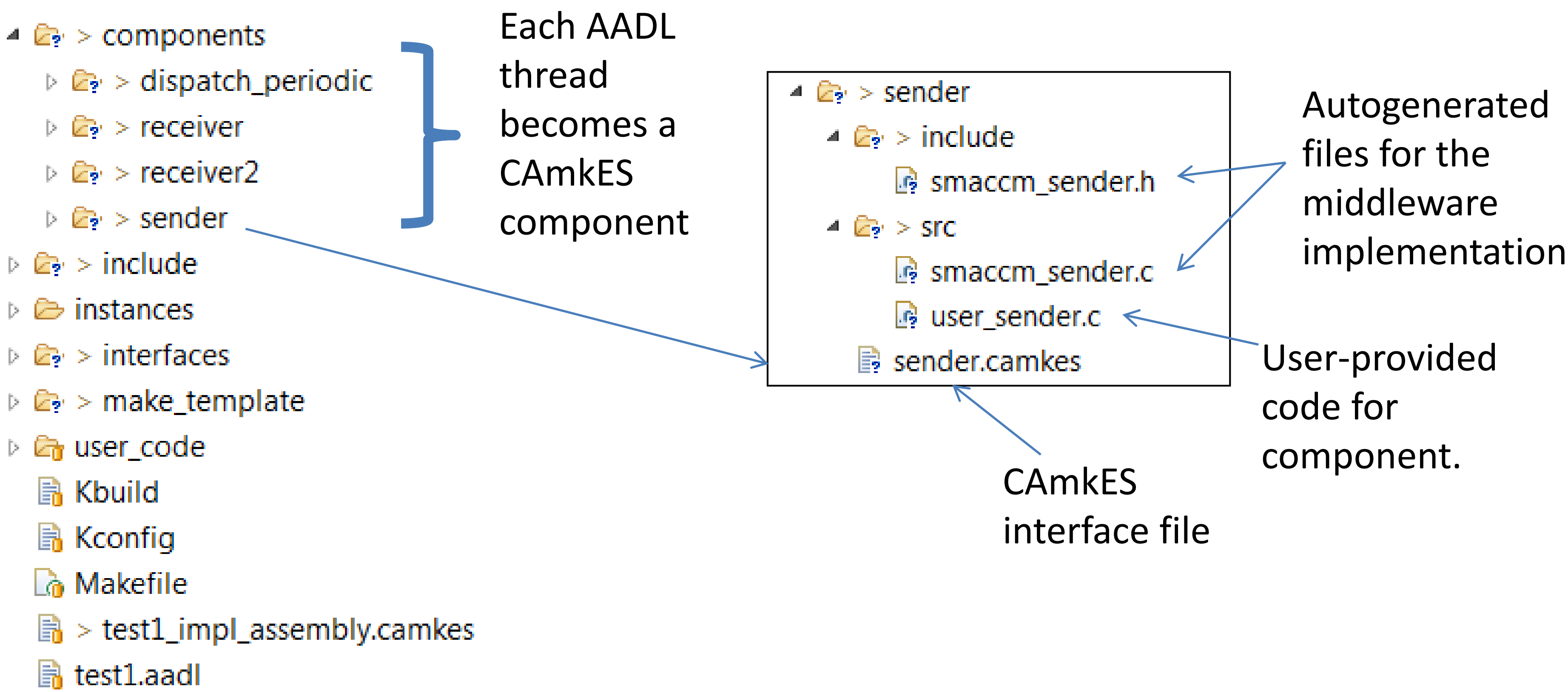
Simple Model Initial Structure



C files describe the component behavior; in this case, there is one C file per component.

AADL files describe the system structure

Structure of Generated Code



Structure of Generated Code

- > components
  - > dispatch_periodic
  - > receiver
  - > receiver2
  - > sender
- > include
  - smaccm_test1_impl_types.h
- instances
- > interfaces
- > make_template
- user_code
- Kbuild
- Kconfig
- Makefile
- > test1_impl_assembly.camkes
- test1.aadl

Common include files for all components. In this case, only the AADL type definitions.

Structure of Generated Code

- > components
  - > dispatch_periodic
  - > receiver
  - > receiver2
  - > sender
- > include
- instances
  - test1_test1_impl_Instance.aaxl2
- > interfaces
- > make_template
- user_code
- Kbuild
- Kconfig
- Makefile
- test1_impl_assembly.camkes
- test1.aadl

OSATE/AADL-generated XML file for the system instance. This file can be ignored.

Structure of Generated Code

- > components
  - > dispatch_periodic
  - > receiver
  - > receiver2
  - > sender
- > include
- instances
- > interfaces
  - receiver_interface.idl4
  - receiver2_interface.idl4
  - sender_interface.idl4
  - uint32_t_writer.idl4
  - uint64_t_writer.idl4
  - void_writer.idl4
- > make_template
- user_code
- Kbuild
- Kconfig
- Makefile
- test1_impl_assembly.camkes
- test1.aadl

Interface definitions for components.

Structure of Generated Code

```
> components
    > dispatch_periodic
    > receiver
    > receiver2
    > sender
> include
instances
> interfaces
> make_template
    Kbuild
    Kconfig
    Makefile
user_code
Kbuild
Kconfig
Makefile
test1_impl_assembly.camkes
test1.aadl
```

Autogenerated templates for makefiles and related build files required by CAmkES. To use them, copy them to the parent directory.

Structure of Generated Code

- > components
  - > dispatch_periodic
  - > receiver
  - > receiver2
  - > sender
- > include
- instances
- > interfaces
- > make_template
- user_code
- Kbuild
- Kconfig
- Makefile
- > test1_impl_assembly.camkes ← Top level assembly file for CAmkES ADL.
- test1.aadl

Supported AADL Constructs: Types

- Types from Base_Types.aadl
    - Integer_{8, 16, 32, 64}
    - Unsigned_{8, 16, 32, 64}
    - Float_32, Float_64
- N-Dimensional Arrays
- Structures
- Externally-defined types:

```
data can_message
  properties
    TB_SYS::Is_External => true;
    TB_SYS::CommPrim_Source_Header => "canDriverTypes.h";
end can_message;
```

Supported AADL Constructs: Ports

- **Example models found in smaccm github repository: smaccm/models/Trusted_Build_Test**
- Data ports
  - Reader gets copy of last write
  - Can be “lossy”
  - Examples: test8, test9, test10
- Event Data ports
  - Queued communications
  - Lossless (unless queue fills up)
  - If queue fills, write call returns **false**
  - Examples: test4, test5, test6, test10, test11, test12
- Event ports
  - Signal only
  - Examples: test13

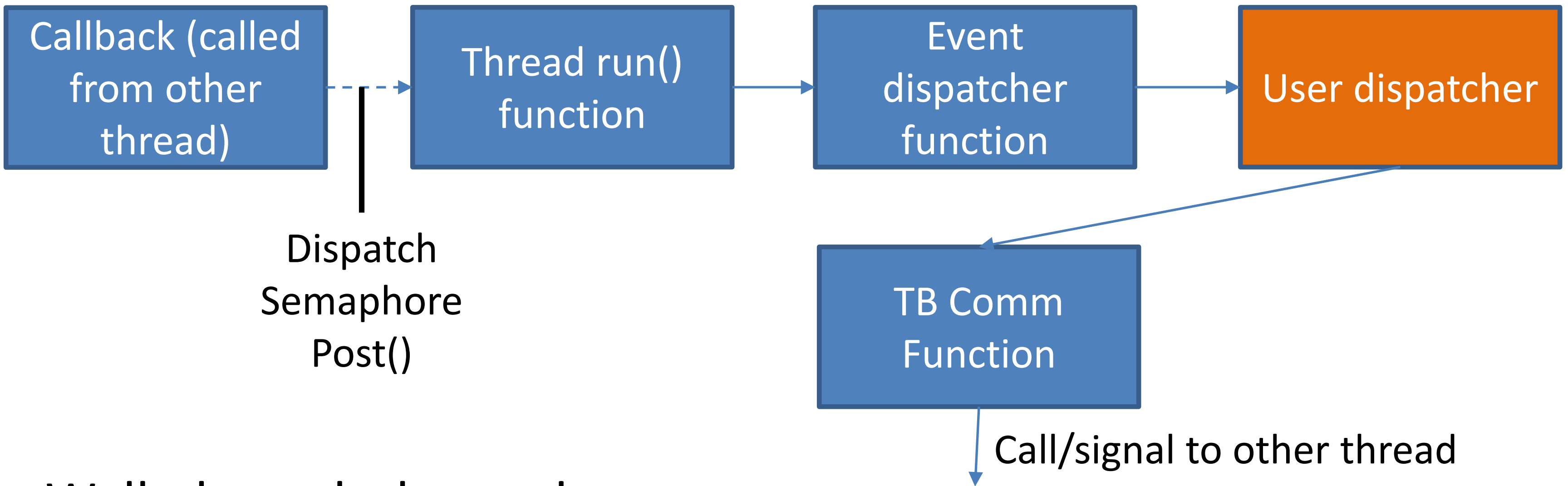
Supported AADL Constructs: Shared Memory

- Provides “raw” interface to shared memory between tasks – pass-through to CAmkES shared memory support (CAmkES dataport)
- Useful for passing large amounts of data between tasks
- Example: test16

Supported AADL Constructs: RPCs

- Provides direct support for RPCs between threads
  - Pass-through to native RPC support in CAmkES
  - In CAmkES, RPCs are organized into “procedures”
  - Analogous concept in AADL is “subprogram group”
- Synchronization
  - CAmkES synchronizes all entry points in same “procedure”
  - However, thread may have several “procedures”
  - Possibility for race conditions
- Example: test_rpc_native

Structure of Threads



- Walk through tb_sender.c
- Thread main function:
  - Waits on a semaphore to dispatch
  - When semaphore "wakes up", calls user dispatchers for events that have occurred
  - User code can call communications primitives provided by TB

DEMO: Periodic Producer / Consumer

- Modify producer/consumer so that consumer is periodically dispatched
  - Switch to the producer_consumer_periodic project
  - Remove the dispatcher from the receiver input port
  - Add a period of 1 second and a periodic dispatcher to the receiver
  - Modify the periodic dispatcher in user_receiver.c file to drain the queue of messages.
- To compile the code:
  - Copy "template" makefile to top-level tb directory
  - Copy all code to camkes/apps directory
  - Add application directory to Kconfig.
- To run the image:
  - Type:
    qemu-system-arm -M kzm -nographic -kernel images/capdl-loader-experimental-image-arm-imx31

Interfacing with External Code

- Drivers and other low-level systems code may not fit the AADL dispatch paradigm
- You can designate a thread as “external”
  - TB generates a template .camkes file matching the communications ports defined in AADL
  - It *does not* generate the C code; you can choose to implement this in any fashion that you like.
  - You can use existing mechanisms in CAmkES to map external thread to *composite* components

Using ISR Ports

- Example in smaccm/models/Trusted_Build_Test/test_irq1
- ISR ports allow you to build "native" drivers
- Interrupt triggers AADL thread
- Memory-mapped IO for interacting with the hardware.

Using ISR ports

```
thread sender
features
Input1: in event port {
  Source_Text => ("user_code/f1.c", "user_code/f2.c", "user_code/f3.h");
  TB_SYS::Is_ISR => true;
  TB_SYS::First_Level_Interrupt_Handler => "timer_flih";
  Compute_Entrypoint_Source_Text => "timer_slih";
  TB_SYS::Signal_Name => "irq";
  TB_SYS::Signal_Number => 27;
  TB_SYS::Memory_Pages => ("mem", "0x53F98000:0x1000");
  TB_SYS::Sends_Events_To => "{{2 Output1}}";
};
...
properties
   Dispatch_Protocol => Sporadic;
   Initialize_Entrypoint_Source_Text => "initialize_timer";
   TB_SYS::Thread_Type => Active ;
 ...
end sender ;
```

ISRs have a first-level and second-level handler

Each ISR must have a unique signal name and number

Each ISR port is associated with a list of named, sized memory regions:
“mem” is the name
0x53F98000 is the location
0x1000 is the size

Interacting with Shared Memory

```
#include <autoconf.h>
#include <stdio.h>
#include <clock_driver.h>
#include <sender.h>
```

CAmkES-generated file that contains the shared buffers

Name is right here.

```
#define KZM_EPIT_BASE_ADDR  (unsigned int)mem
#define KZM_EPIT_CTRL_ADDR  (KZM_EPIT_BASE_ADDR + 0x00)
#define KZM_EPIT_STAT_ADDR  (KZM_EPIT_BASE_ADDR + 0x04)
#define KZM_EPIT_LOAD_ADDR  (KZM_EPIT_BASE_ADDR + 0x08)
#define KZM_EPIT_COMP_ADDR  (KZM_EPIT_BASE_ADDR + 0x0C)
#define KZM_EPIT_CNT_ADDR   (KZM_EPIT_BASE_ADDR + 0x10)
…
```

User ISR Port Code

```
void timer_flih() {
epit_irq_callback();
}
```

First level handler for fast response

```
void timer_slih() {
smaccm_thread_calendar();
}
```

Second level handler for longer processing

CAmkES Code

Sender.camkes:

```
import "../../interfaces/sender_interface.idl4";
import "../../interfaces/uint32_t_writer.idl4";
import "../../interfaces/receiver_interface.idl4";

component sender {
control;
  has semaphore smaccm_dispatch_sem;

  uses receiver_interface receiver_inst;

  // interfaces for IRQ port dispatchers (if any).
  consumes DataAvailable irq;
  dataport Buf mem;
}
```

Test_irq1_impl_assembly.camkes:

```
component irq_hw {
   hardware;
   dataport Buf mem;
   emits DataAvailable irq;
}
…
assembly {
  composition {
     component irq_hw irq_obj;
     …
     // IRQ connections (if any)
      connection seL4HardwareMMIO irq_mem(
        from sender_inst.mem, to irq_obj.mem);
      connection seL4HardwareInterrupt irq_irq(
        from irq_obj.irq, to sender_inst.irq);
   }
   configuration {
      irq_obj.mem_attributes = "0x53F98000:0x1000";
      irq_obj.irq_attributes = 27 ;
   }
}
```

Wrapping Existing Drivers / External Code

- NICTA has existing drivers and external code.
- How do we use them?
- Declare a thread in AADL as “external”
  - It is usually an active thread
  - This leaves a placeholder in the model
  - It will be “wired up” to other threads as usual in the assembly
  - You can fill it in.
    - In most cases, best way is to wrap the driver and HW components in a CAmkES **composite component.**
    - You need to write the C and CAmkES Code
    - Trusted build provides .template files to get you started.

Example: UART

- Available at github at: smaccm/models/Trusted_Build_Test/test_uart_active
- CAmkES Interacts with rest of the system using two methods:

```
// reader
void pilot_recv(int32_t uart_num, int32_t c);

// writer
int32_t uart_write(int32_t uart_num, int32_t wsize);
```

- Uses shared buffer for write.

On the AADL Side

Packet with 1 byte

```
thread uart
features
send : in event data port uart_packet.impl {
Compute_Entrypoint_Source_Text => "send_handler";
    SMACCM_SYS::Sends_Events_To => "{{}}";
};

recv: out event data port uart_packet.impl {
      SMACCM_SYS::CommPrim_Source_Text => "recv_data";
};

properties
    SMACCM_SYS::Is_External => true;
    Dispatch_Protocol => Sporadic;
    -- Source_Text => ("user_code/user_sender.c");
    Priority => 10;
    Stack_Size => 256 bytes;
    SMACCM_SYS::Thread_Type => Active ;
    Compute_Execution_Time => 10 us .. 50 us;
end uart ;
```

This says “external”

CAmkES Composite Components

- Allows you to wrap other components, their connections, and their configurations, in one component.
- Looks like a CAmkES Assembly file except that it has top-level interface for communication with the rest of the system.
- We'll have to write this by hand; an example is on the next slide

CAmkES Composite Components

```
....
component uart {
    uses test_uart__uart_packet_impl_writer uart_recv;
    provides test_uart__uart_packet_impl_writer send;

   composition {
     component uartbase uartbase;
     component uart_driver uart;
     connection seL4HardwareMMIO uart_mem
            (from uart.uart0base, to uartbase.mem);
     connection seL4HardwareInterrupt uart_irq
            (from uartbase.irq, to uart.interrupt);

     connection ExportRPC conn1(from send, to uart.send);
     connection ExportRPC conn2(from uart.recv, to uart_recv);
   }

  configuration {
      uartbase.mem_attributes  = "0x12C10000:0x1000";    //UART1
      uartbase.irq_attributes  =  84;                //UART1 interrupt
      uart.ID = 3;
   }
}
```

```
component uartbase {
   hardware;
   dataport Buf mem;
   emits DataAvailable      irq;
}
```

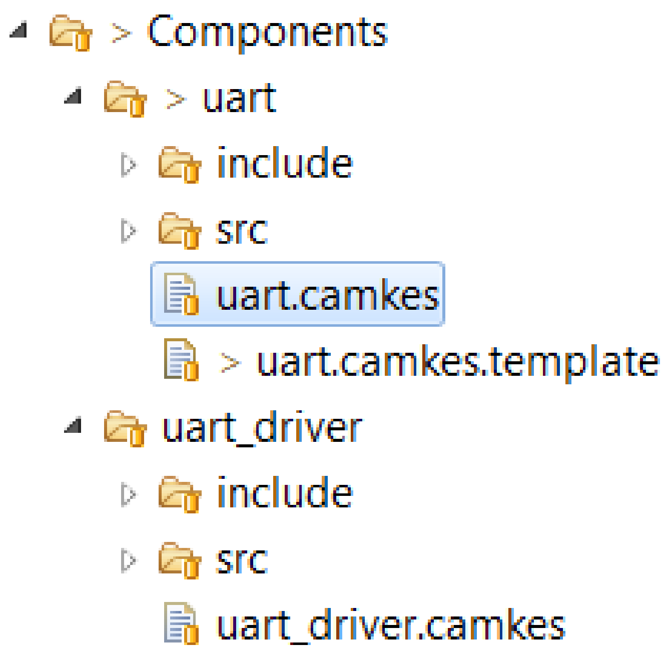
Original driver code goes in here.

Connections to the hardware.

Connections in/out to AADL

This comes from original CAmkES VM Assembly.

How it splits into files



…

"AADL Facing" composite component

Driver code (not autogenerated)

uart_driver CAmkES Code

```
import "../../interfaces/uart_interface.idl4";
import "../../interfaces/test_uart__uart_packet_impl_writer.idl4";

component uart_driver {
control;

// provided interfaces for input event / event data ports
provides test_uart__uart_packet_impl_writer send;
uses test_uart__uart_packet_impl_writer recv;

// information from native UART driver
consumes DataAvailable  interrupt;
has semaphore         read_sem;
has semaphore         write_sem;
attribute int         ID;

// buffer is for shared memory.
dataport Buf          uart0base;
}
```

This name must match the AADL CAmkES interface

```
bool send_write_test_uart__uart_packet_impl(const test_uart__uart_packet_impl *arg) {
		int32_t result = -1;

		// Other options rather than fail: re-send up to a bounded # of times?
		// int32_t retries = 0;
		result = uart_write(arg->uart_num, arg->datum);
		if (result == -1) {
				printf("send failed!.\n");
				return false;
		}
		return true;
}

void pilot_recv(int32_t uart_num, int32_t c) {
		test_uart__uart_packet_impl packet;
		packet.uart_num = uart_num;
		packet.datum = c;

		// MWW: might want to do error checking here
		recv_write_test_uart__uart_packet_impl(&packet);
}
```

Note: this is hand-written code!

This name must match the AADL CAmkES interface

Makefile modifi

You have to add the build instructions for the component that is not visible to AADL

```
...
uart_driver_CFILES :=  \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/src/*.c)) \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/src/plat/${PLAT}/*.c)) \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/src/arch/${ARCH}/*.c))

uart_driver_HFILES := \
  $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/include/*.h)) \
  $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/include/*.h))

uart_driver_ASMFILES := \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/crt/arch-${ARCH}/crt0.S)) \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/src/*.S)) \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/src/arch/${ARCH}/*.S)) \
   $(patsubst ${SOURCE_DIR}/%,%,$(wildcard ${SOURCE_DIR}/components/uart_driver/src/plat/${PLAT}/*.S))
...
```

Issues

- Can wrap as big or small a “chunk” as we want
  - …Thanks to NICTA’s composite components
  - We just need to normalize the interface
  - I hope to wrap the entire Linux VM as one external component
- Autogenerated makefile must be extended.
  - AADL tool is not designed to manage complex makes.
  - It is designed to automate simple models and assist the designer.

VM Support

- seL4/CAmkES can be used to host Linux VMs as “components”
- Trusted Build can support VMs in two ways:
  - As external “threads” (Air Team)
    - Allows custom interfaces between seL4/CAmkES OS and VM
    - In Air Team vehicle: *vchan* was used as mechanism for communication
    - VM code is treated as “black box”
  - As *virtual processors* (Ground Team)
    - This is more principled approach
    - AADL model can describe comms between threads/processes within the VM

Cross-VM Communications

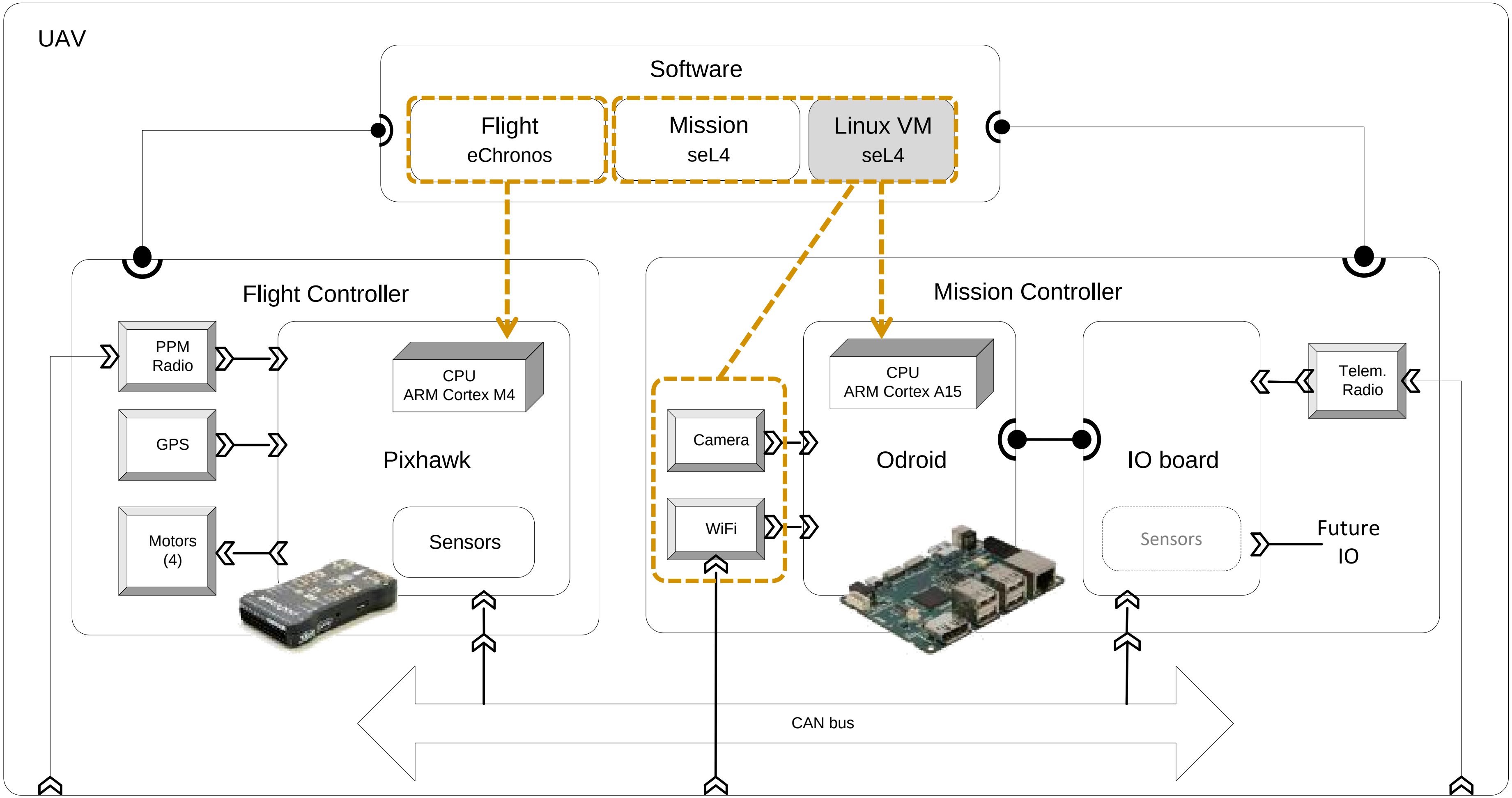
- HRL has implemented an extension of TB to support data port communications across the VM boundary
  - Automatically builds driver support and interfaces to support communications across boundary
  - It is not part of the publically released TB

Wrap Up

- Trusted Build is a tool for generating system implementation skeletons from AADL models
- Goal is to make AADL analysis meaningful against generated code
- It supports a subset of AADL
  - This is intentional: goal is to have small enough code base to have confidence in result
- Can target multiple Oses: seL4, VxWorks, eChronos, linux
  - Focus for this talk was on CAmkES/seL4

PUTTING CODEGEN IN CONTEXT

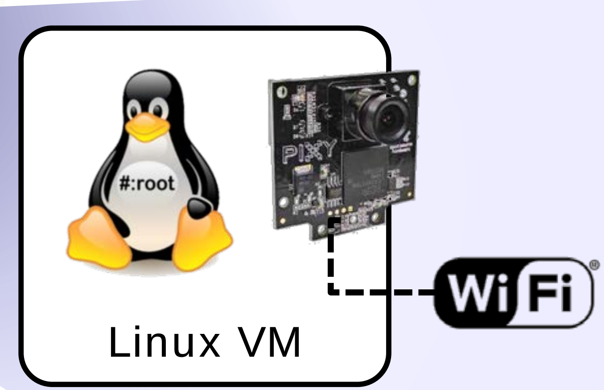
SMACCMcopter Phase 2 Architecture



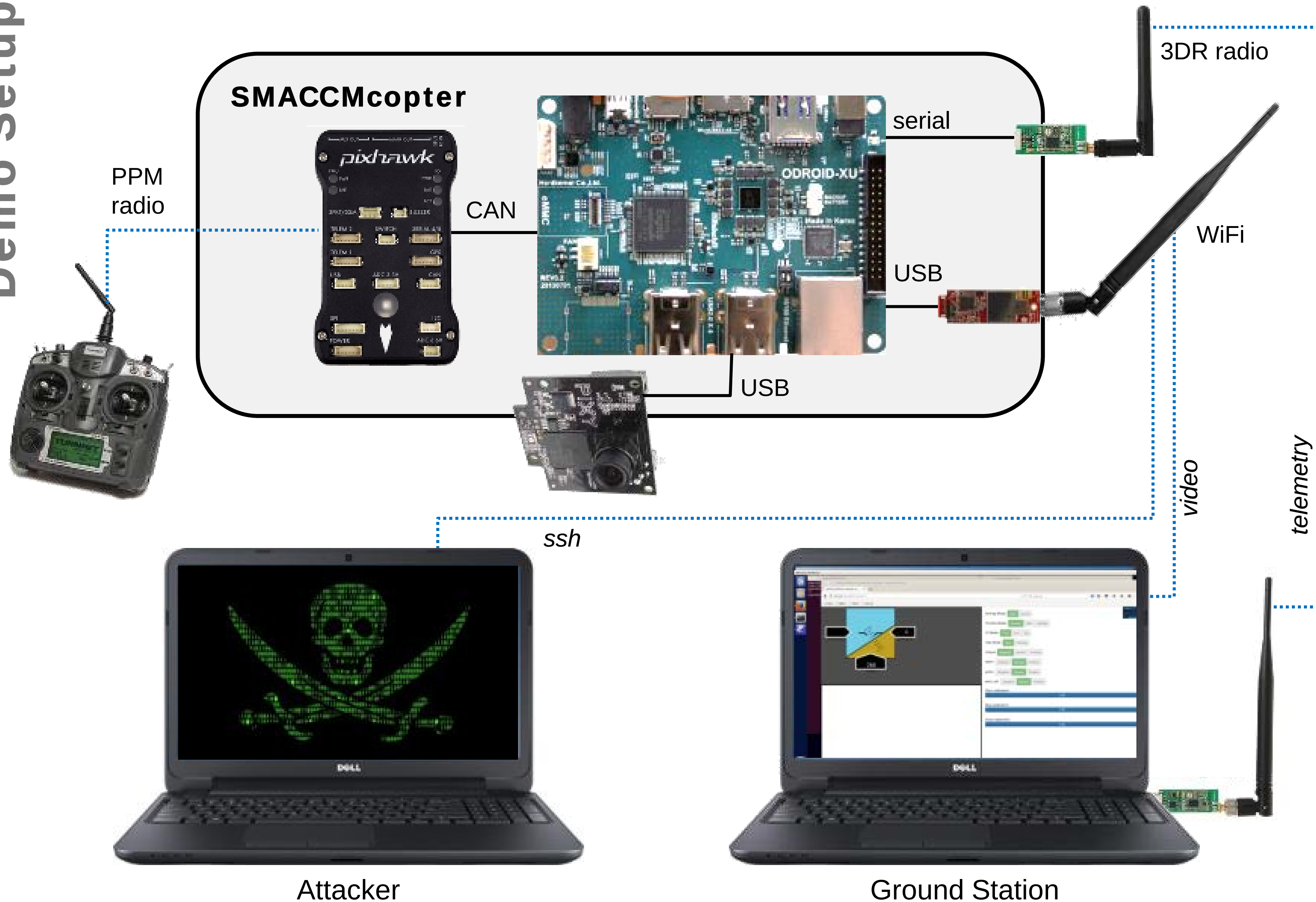
Evaluation: Attack Surface

Linux VM hosted on seL4
Can the Red Team:

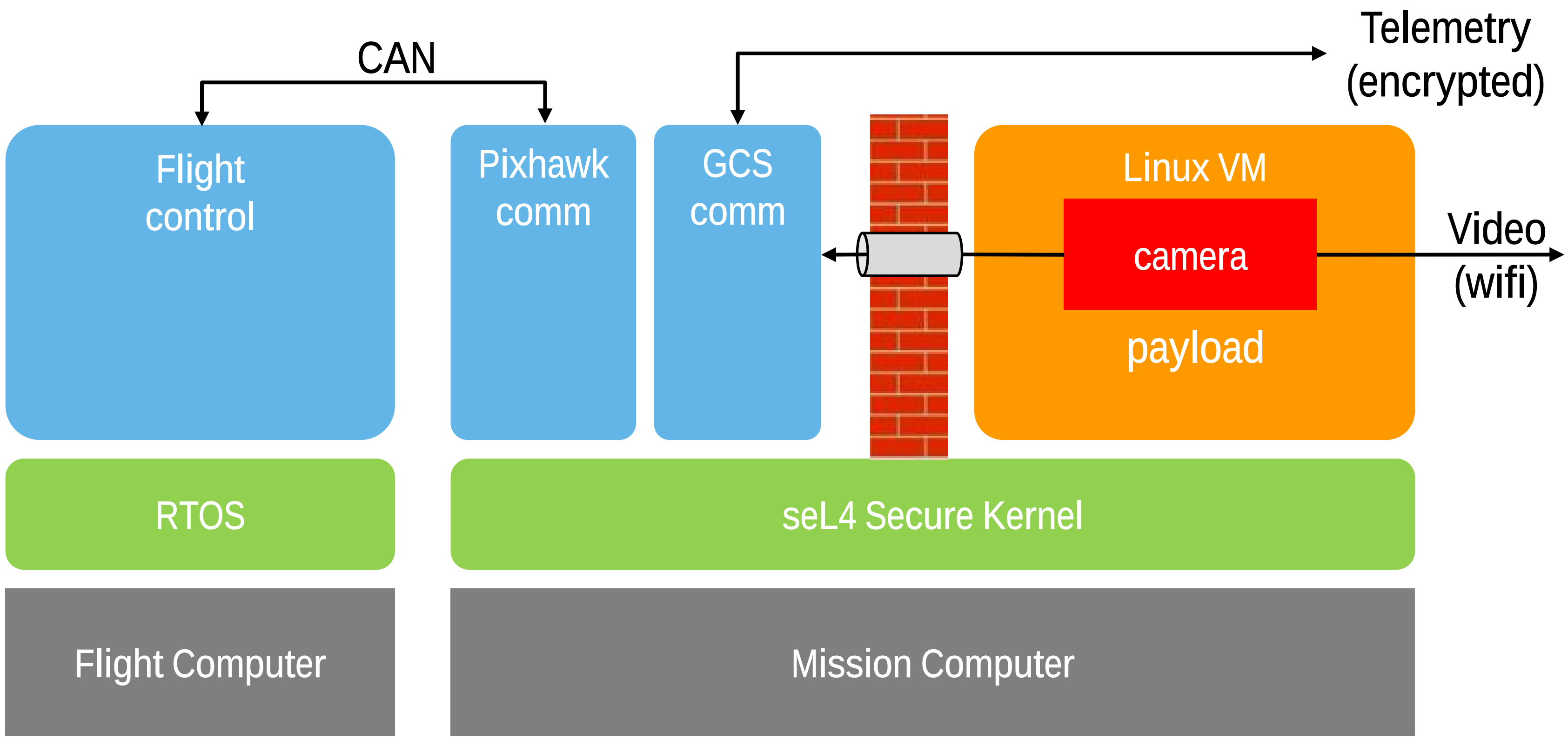
- Impact task execution?
- Access keys?
- Send on CAN bus?
- Control vehicle?



Demo Setup

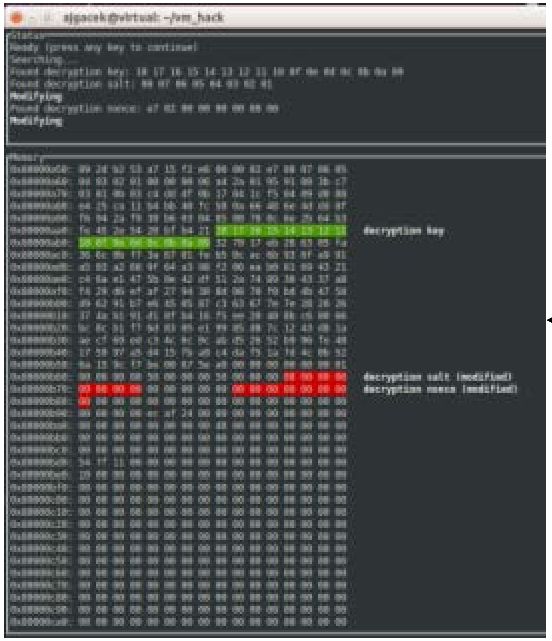

SMACCMcopter
pixhawk
ODROID-XU
CAN
serial
USB
USB
3DR radio
WiFi
PPM
radio
ssh
video
telemetry
Attacker
Ground Station

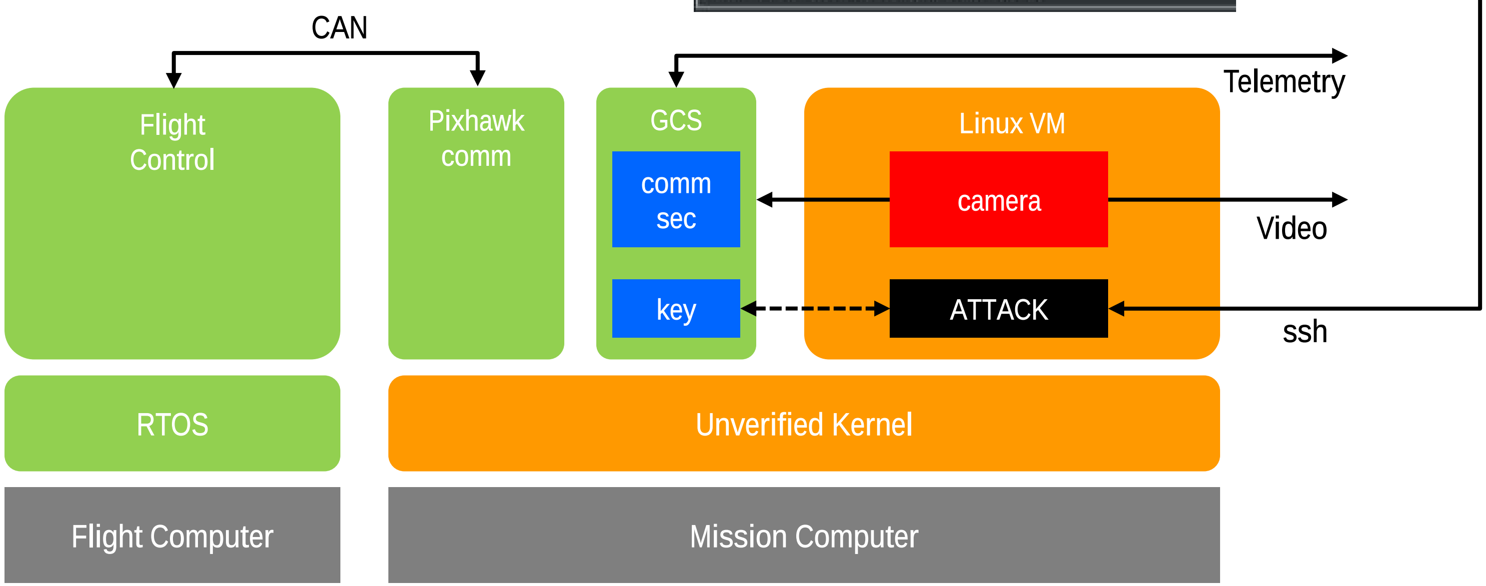
Secure Software Architecture



Key Compromise

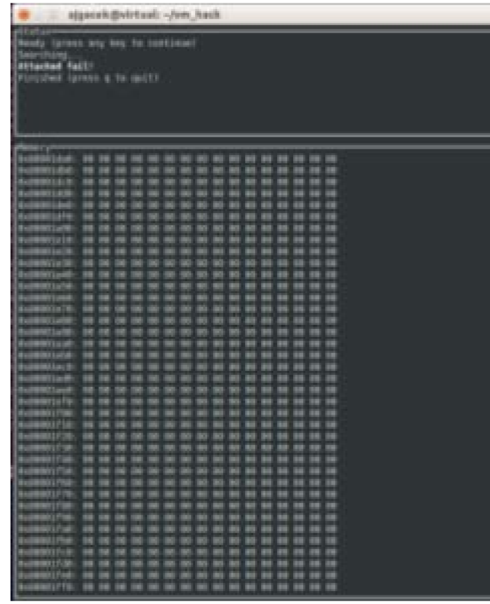
- Unverified kernel has a security vulnerability affecting memory protection
- Attacker uses vulnerability to break out of the Linux virtual machine (VM) and access the encryption process memory
- Attacker is able to overwrite the key and take control of the vehicle

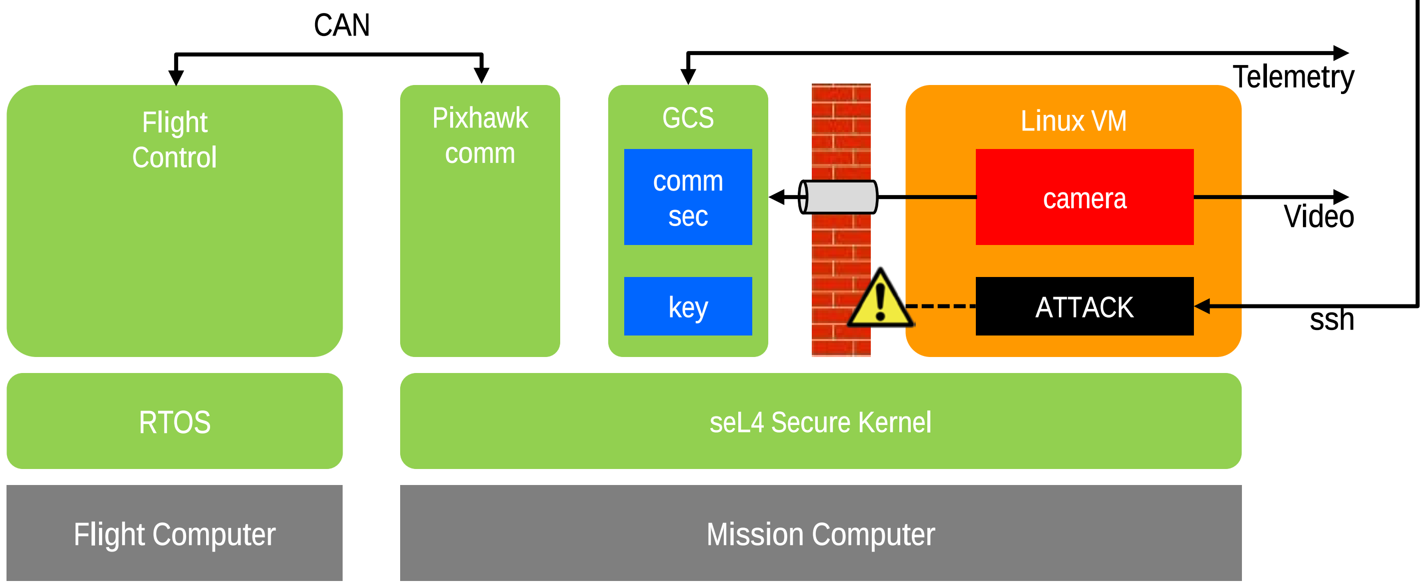




Key Compromise Blocked

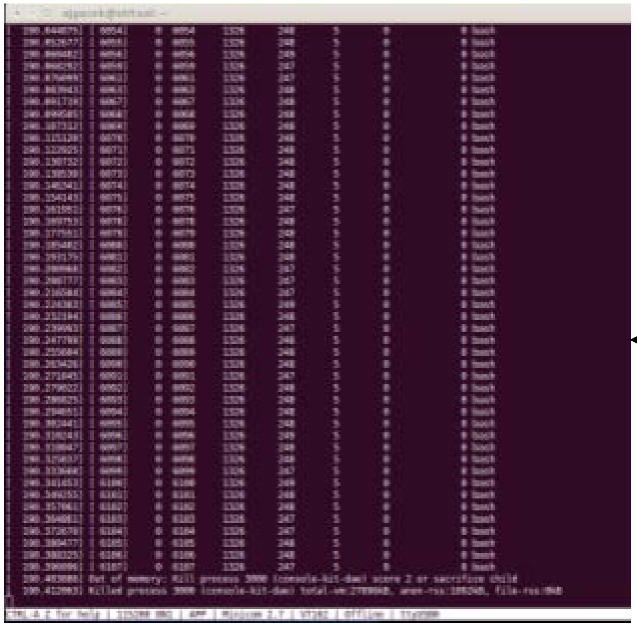
- With the seL4 kernel, the key exfiltration attack is unsuccessful
- Any attempt to read memory outside of what is allocated to the Linux VM fails

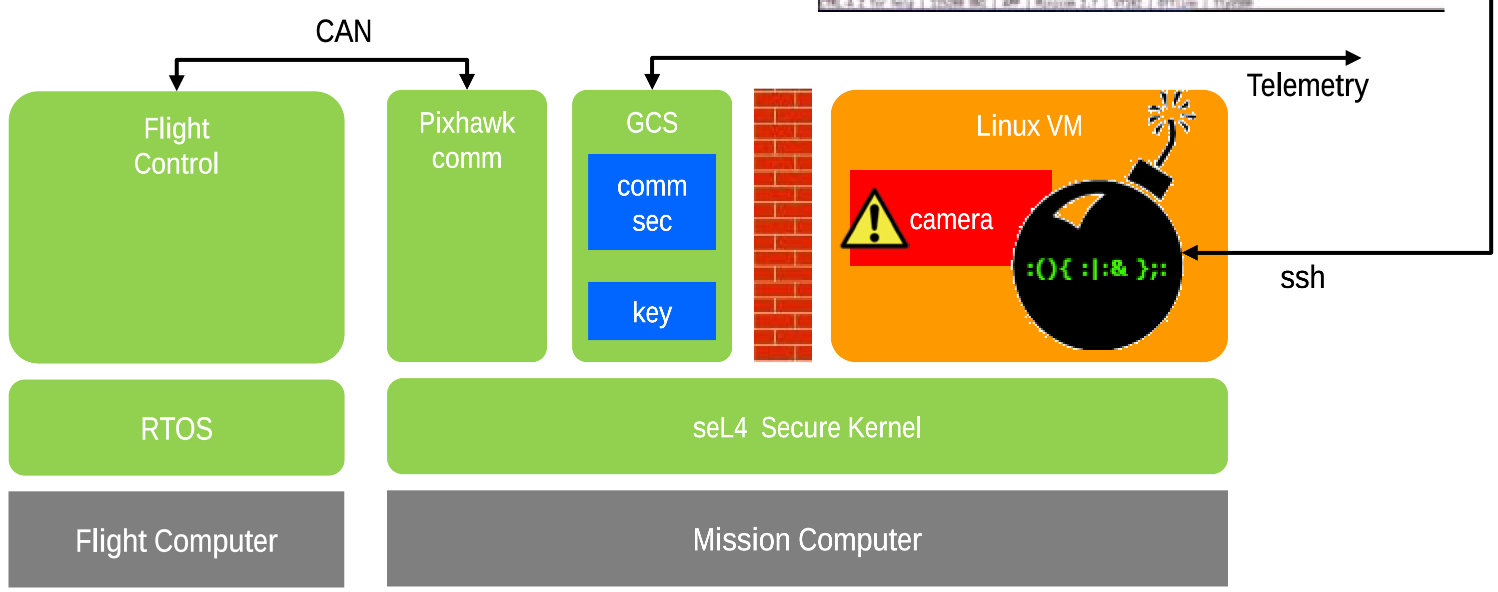




Fork Bomb Contained

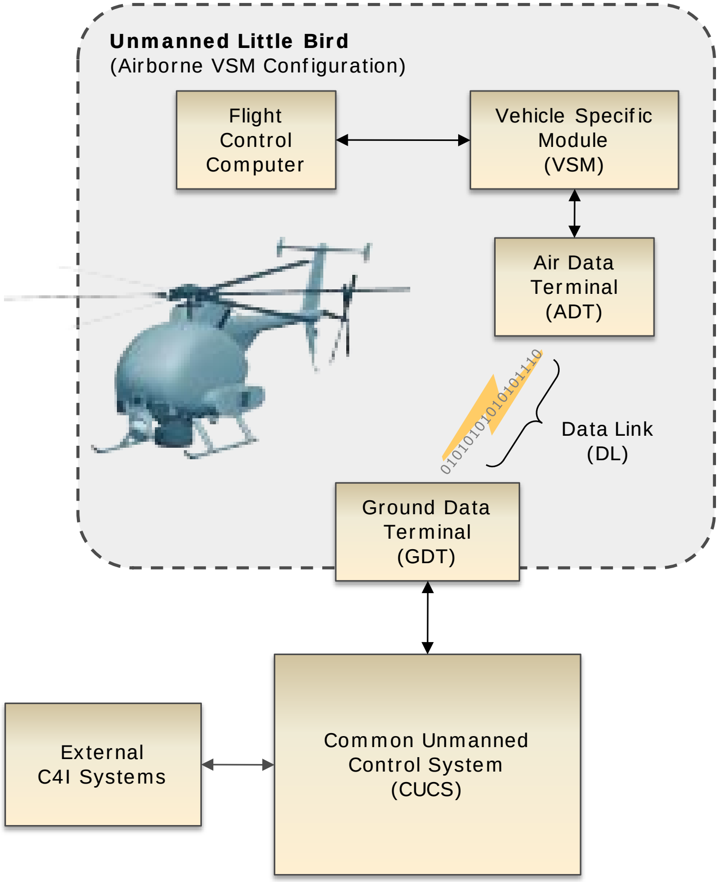
- A “fork bomb” is an attack that exhausts system resources
- seL4 confines the impact to the Linux VM
- Payload (camera) stops working but the rest of the system is unaffected





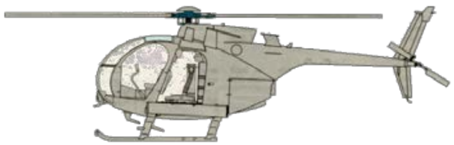
HACMS Phase 2 Demo – 11 August 2015

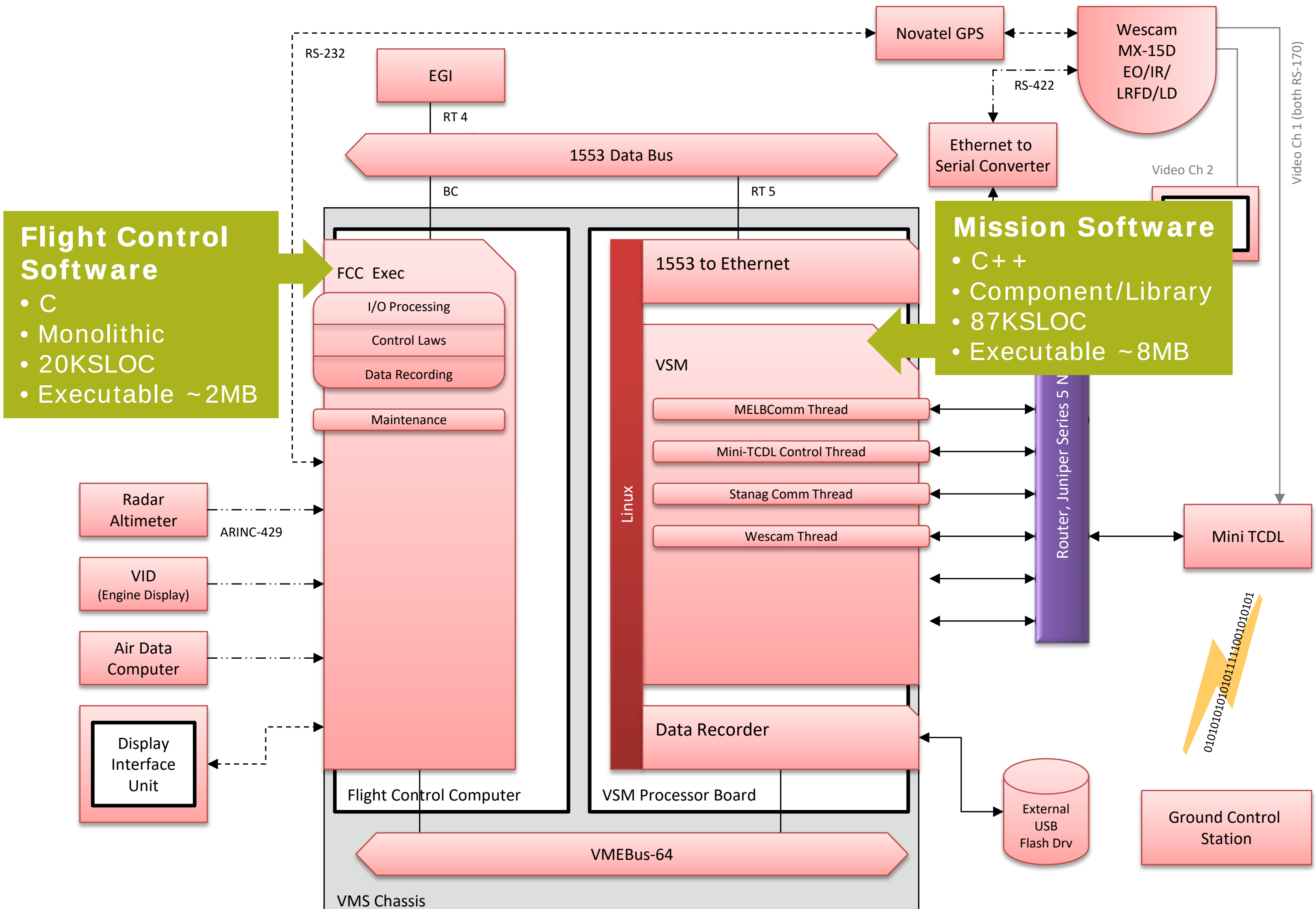
Boeing: ULB and STANAG 4586



- STANAG 4586 is an interoperability architecture designed to allow mix and match control of UAVs by ground stations
- Compatible ground station
  - STANAG 4586 defines the requirements for a compliant Ground Control Station (GCS) capable of potentially controlling multiple dissimilar UAVs.
- Compatible UAV
  - STANAG 4586 defines the requirements for a Vehicle Specific Module (VSM) that is the interoperable interface to the air vehicle
  - The standard is consistent with either an airborne or ground based VSM
- The ULB VSM is on the aircraft

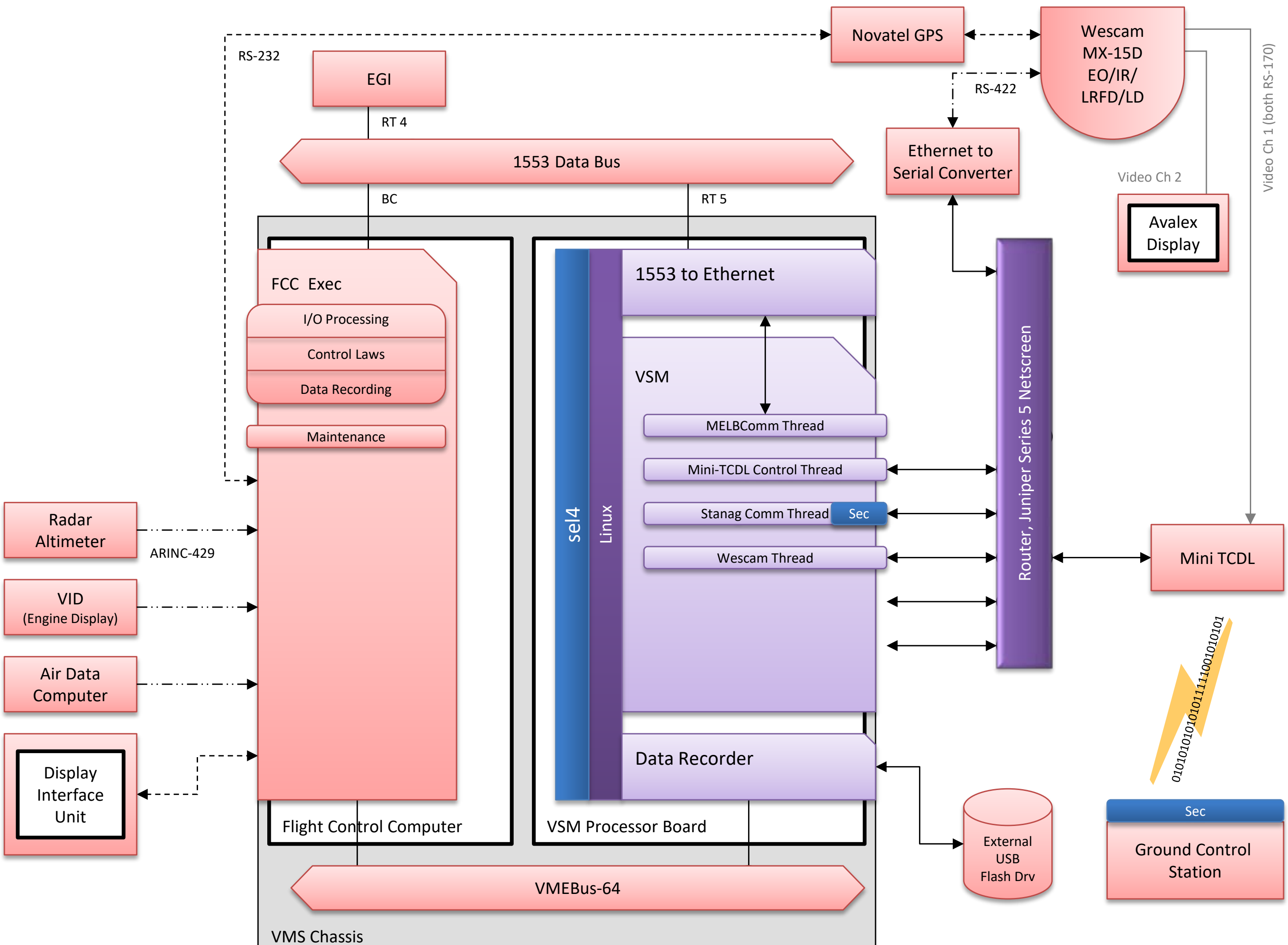
Baseline ULB Architecture





Phase 1 Architecture

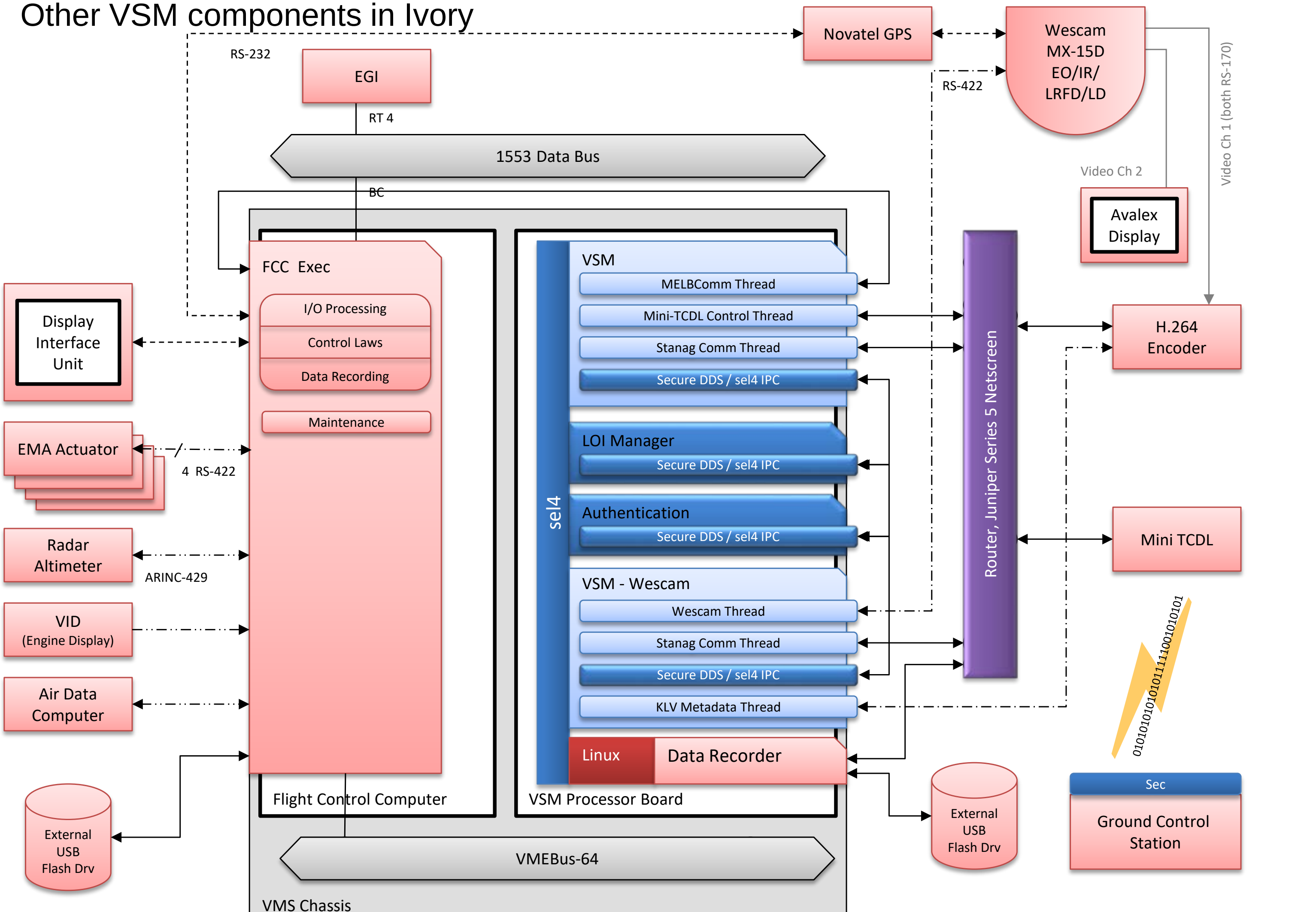
SMACCM Authentication
VSM Hosted on seL4 in a single Linux VM



Phase 2 Architecture

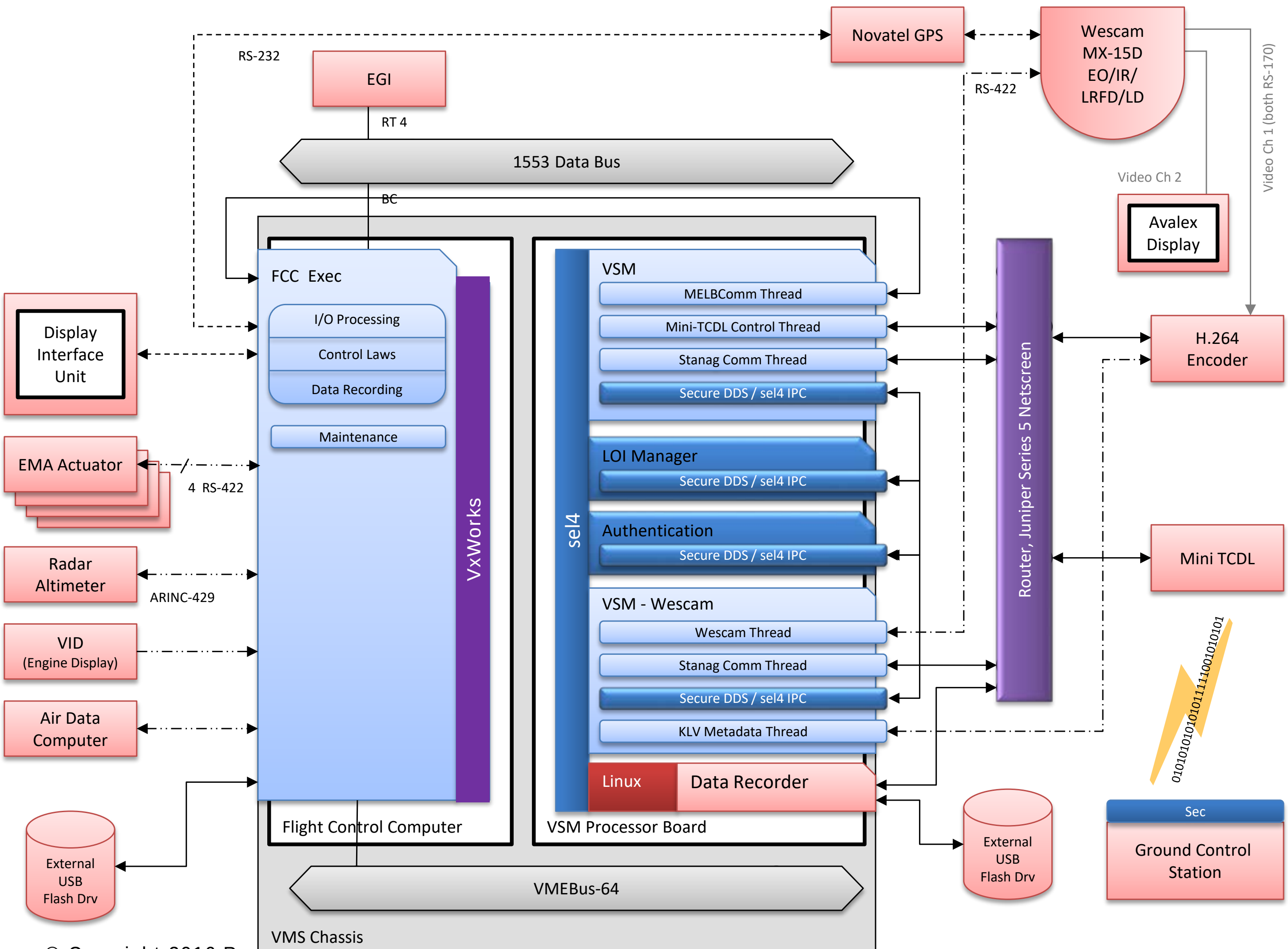
VSM split into “navigation” and camera VSMs
Authentication and LOI Ivory components
Other VSM components in Ivory

VSMs natively on seL4
Simplified VSM to FCC Communication



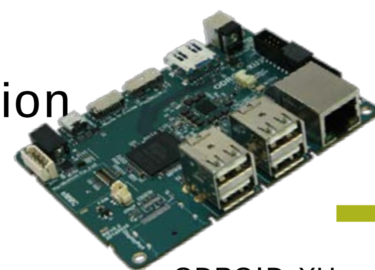
Phase 3 Architecture

FCC AADL model, running on VxWorks
Some FCC components implemented in Ivory/Tower



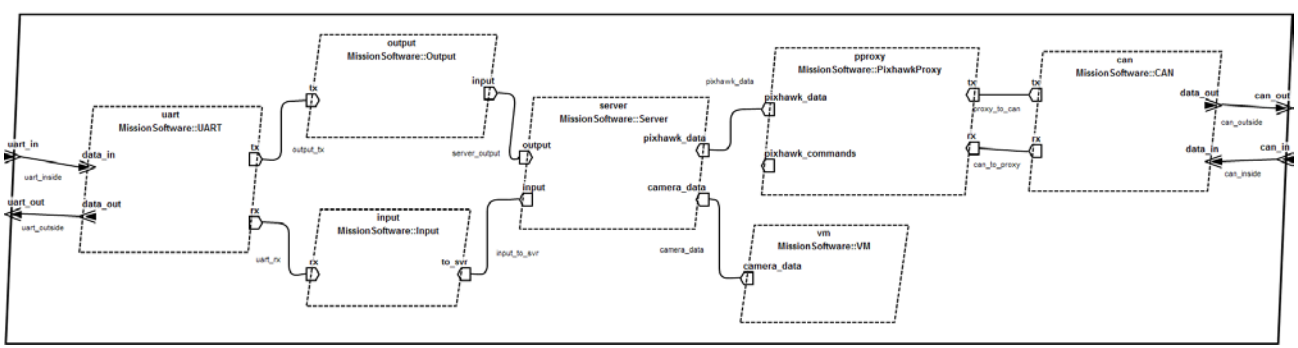
Phase 3 – SMACCMcopter

- Final version of SMACCMpilot on eChronos on Pixhawk with updated Ivory/Tower build
- SMACCMcopter mission computer hardware refresh
- Update AADL model for SMACCMcopter mission computer
- Focus on verification of system-level properties
  - E.g., Geofence
- CAN gateway and protocol verification
  - Untrusted network segment
- seL4 real-time features



ODROID-XU

NVIDIA Tegra K1 SoC
- Quad-Core ARM Cortex-A15
- IOMMU



Conclusion

- Cyber-secure: not vulnerable to a broad range of attacks
  - Secure communications with ground
  - Enforced software task separation onboard
  - Correct data flows and behaviors in system architecture
- Software components have no structural flaws
  - Analysis and construction based on *formal methods*
- Untrusted code can be safely used on the platform within trusted boundaries
  - Legacy code or COTS

Secure systems can be built without sacrificing performance

Loonwerks

More information, code, and papers available at:

Loonwerks.com