Practical real-time operating system security for the masses

ARM

Milosch Meriac

Principal Security Engineer

github.com/ARMmbed/uvisor

ARM TechCon
25th October 2016

Why is microcontroller security so hard ?

Security + time = comedy

If a product is successful, it will be hacked

Who can't suppress snickering on seemingly dumb security bugs in other people's product? It's easy and delightful to declare developers of failed products as incompetent.
As old stuff has old bugs – we won't run out of entertainment anytime soon.

Device lifetime

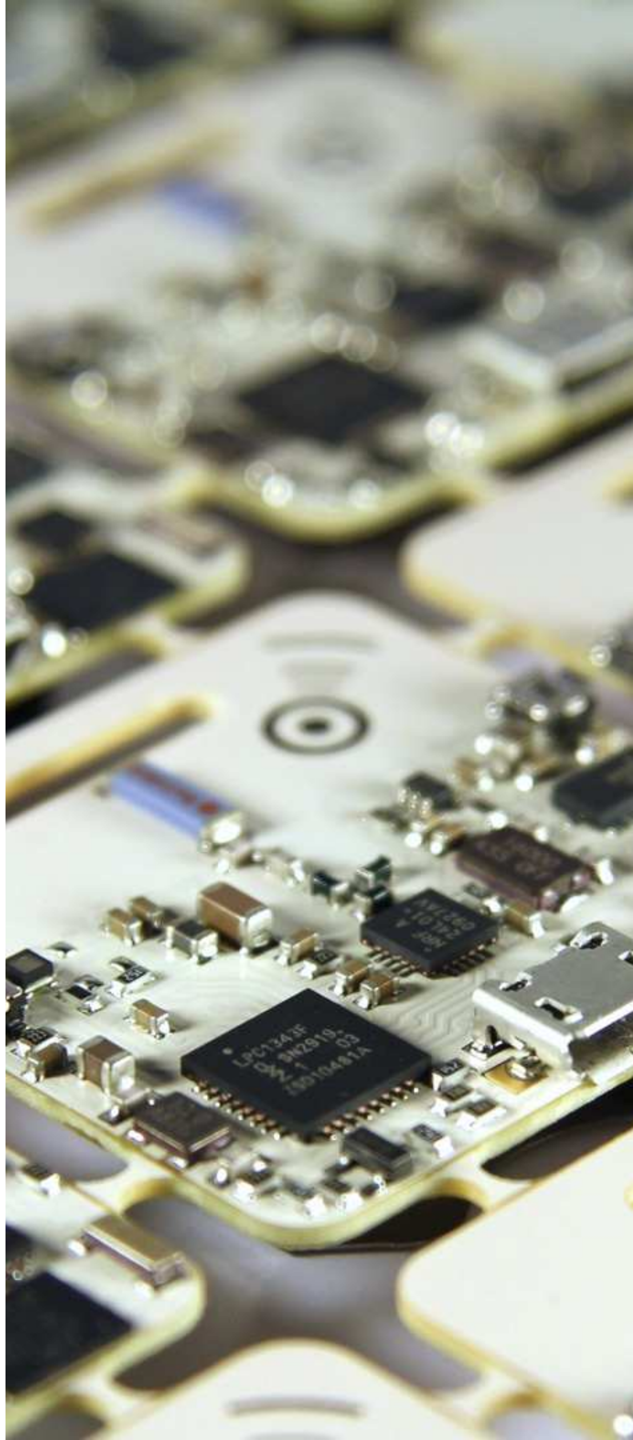
The security of a system is dynamic over its lifetime. Lifetimes of home automation nodes can be 10+ years.

Attack scale well

The assumption of being hacked at some point requires a solid mitigation strategy for simple, reliable and inexpensive updates.
Do our defenses scale with our attackers?

Assume you can't prevent it

Value of bugs is expressed by value = number_of_installations x device_value. Increased value and large deployments drive attackers - especially in the IoT.
Massively parallelized security researchers/attackers vs. limited product development budgets and time frames.



Flat memory models

The 80's called, and they want their security model back

A flat address space as the result of lack of a memory management units (MMU) does not justify the absence of security. Many microcontrollers like ARM Cortex-M3/M4 provide a hardware memory protection unit (MPU) instead.

No separation

Flat memory models and ignorance of MPUs blocks vital security models like "least privilege".

Escalation

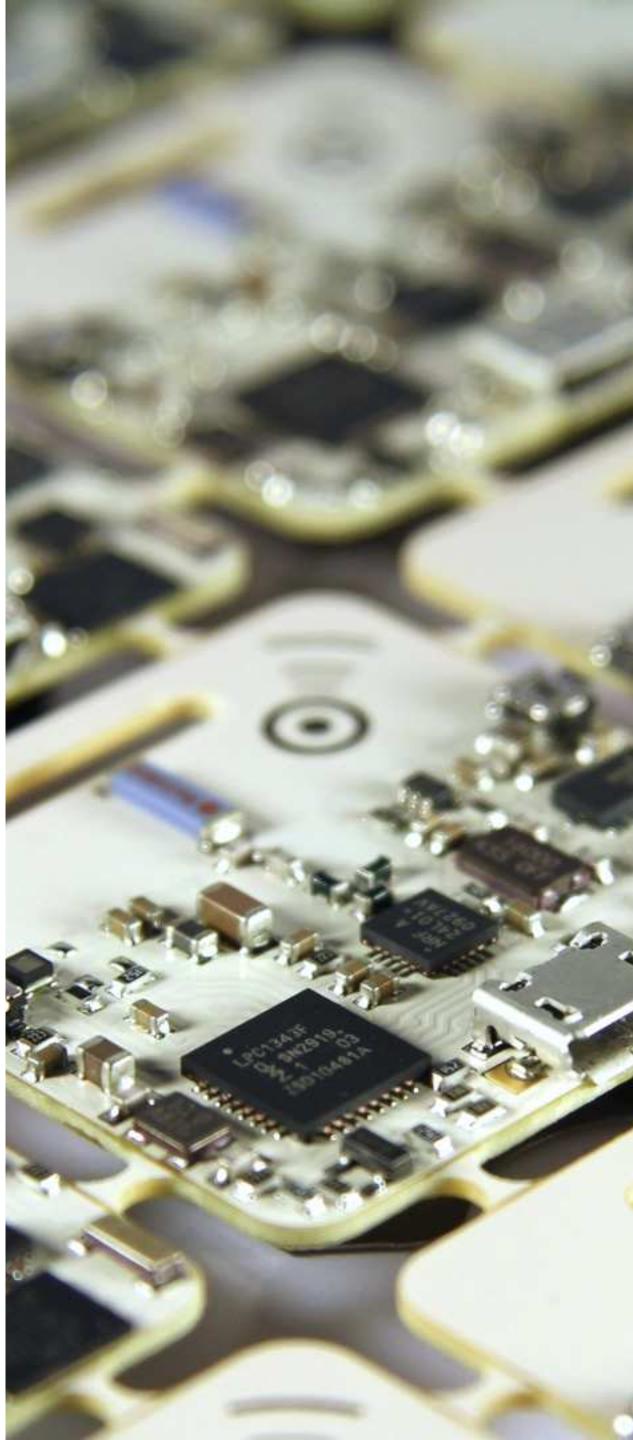
Flat memory models enable escalation & persistence of bugs by uncontrolled writing to flash memories.

Verification

Security verification impossible due to the immense attack surface and lack of secure interfaces between system components.

Leakage

All your bases are belong to us – thanks to leakage of device secrets like identity keys or even class secrets.



Resources matter

Lack of abstraction is expensive

Point-solutions are easy to build for microcontrollers, but hard to maintain and to reason about. Abstractions are hard to build, but allow to amortize security verification across a large range of devices.
For crypto APIs highly portable abstractions are common now – yet OS independence is not common for many other security critical components.

Public key cryptography

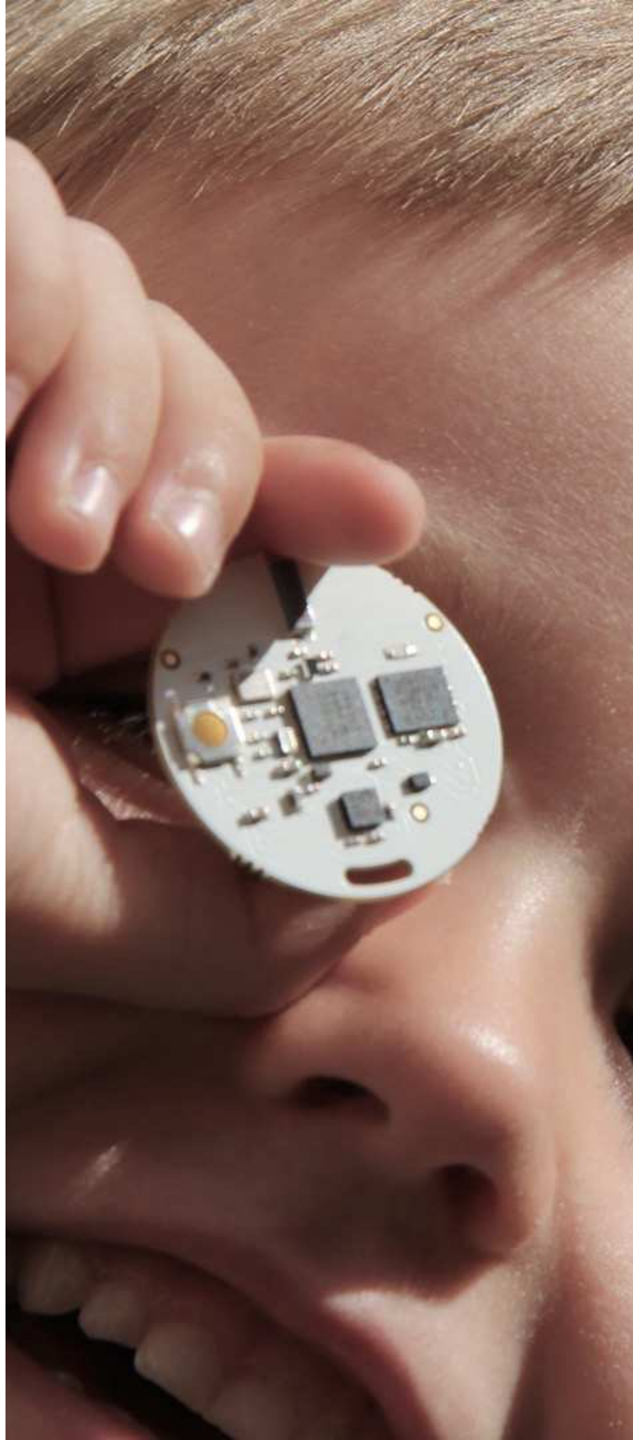
Public-key crypto is absent from many low end devices due to memory and speed constraints.

Shortcuts

Vendors take compromises on security to reduce footprint. Think for example of commonplace class key usage versus supporting key provisioning and per-device secrets.

Communication

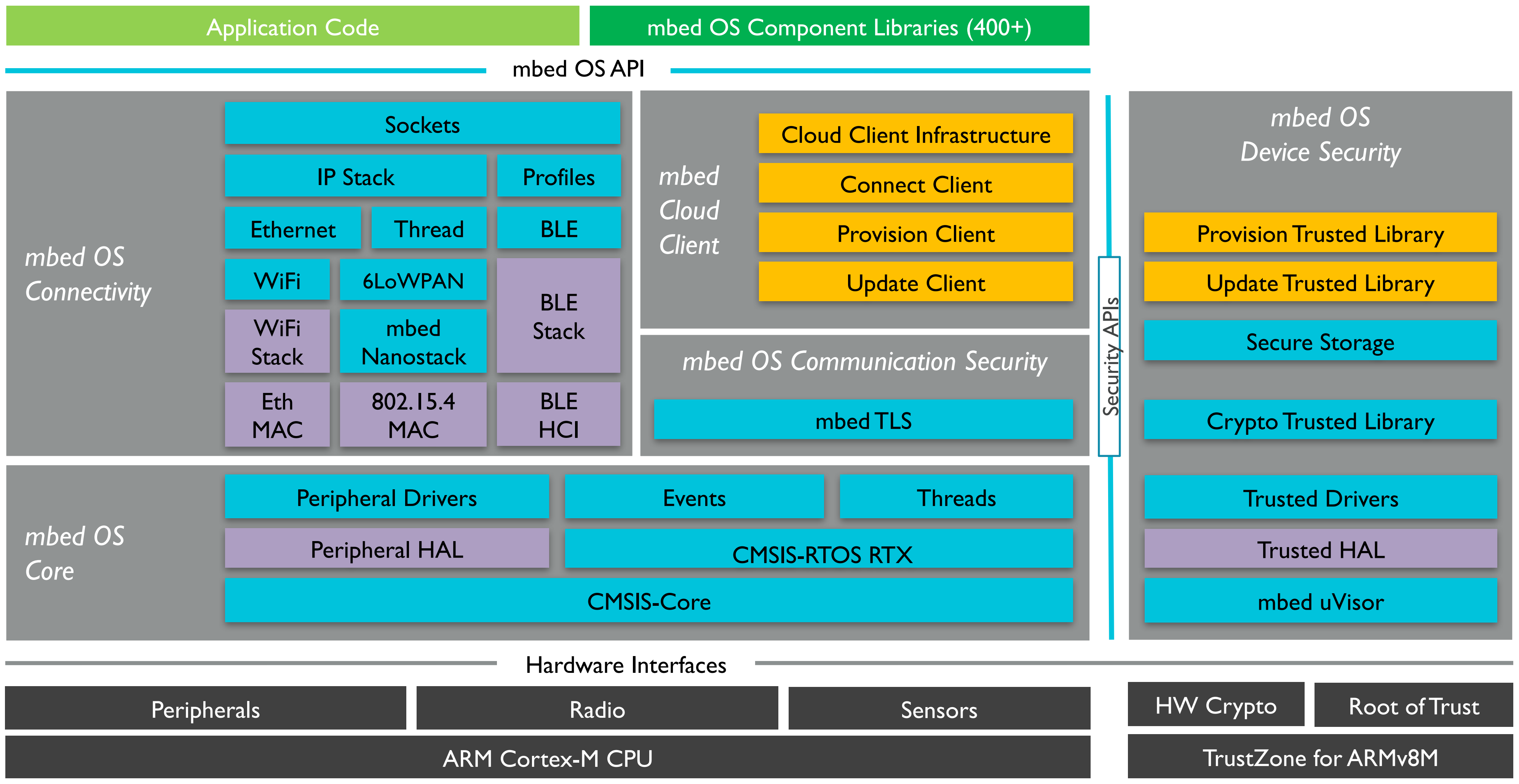
Less abstraction requires more communication of limitations to developers.
This usually results in higher level components having wrong assumptions about low level components. Think of flash config writing in the presence of a local attacker. Be scared.



Enter ARM mbed …

ARM® mbed™

ARM mbed OS: architected platform security



ARM mbed OS: architected platform security

Lifecycle security
mbed Cloud secure identity, config and update

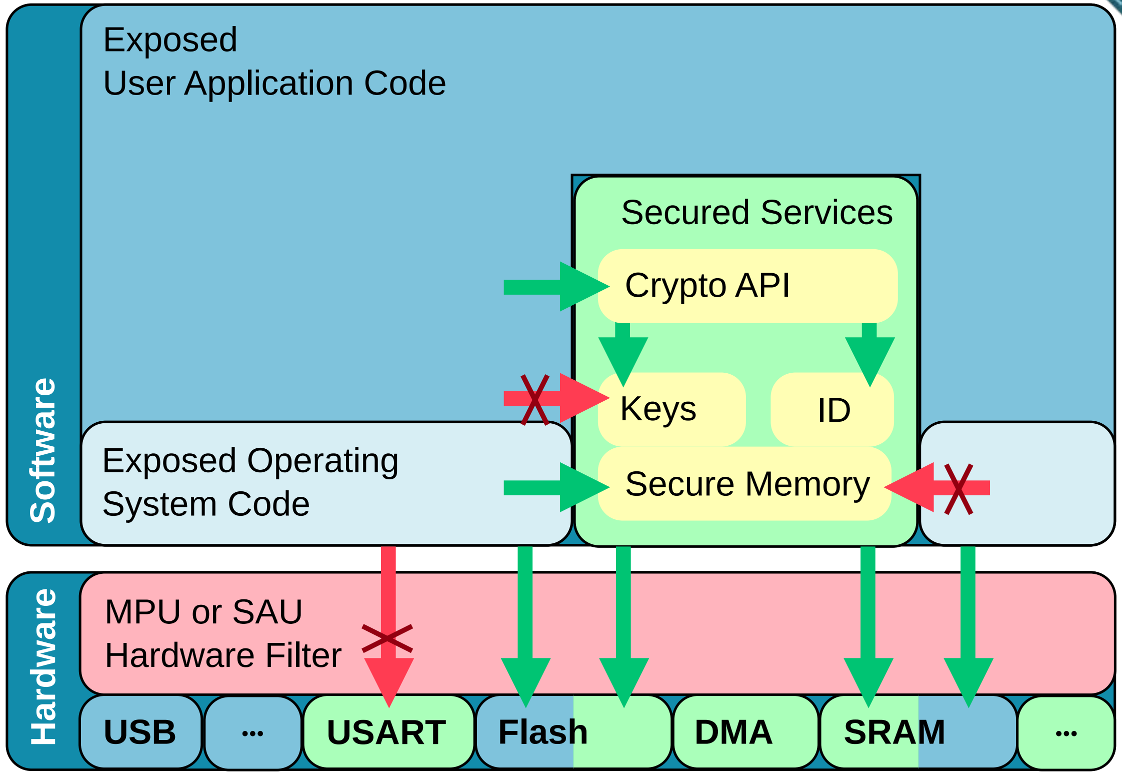
Communication security
mbed TLS

Secure code compartments
mbed uVisor on ARMv7-M & ARMv8-M MPU

ARM

mbed uVisor hypervisor: Hardware security for microcontrollers

- Enables compartmentalization of threads and processes for microcontrollers.
- Developed on github under Apache license.
- mbed uVisor initialized first in boot process
- Private stack and data sections.
- Initialization of memory protection unit (MPU) based on box permissions:
  - Whitelist approach – only required peripherals are accessible to each box.
  - Each box has private .bss data and stack sections.
- De-privileges execution, continues boot unprivileged to initialize OS and libraries.

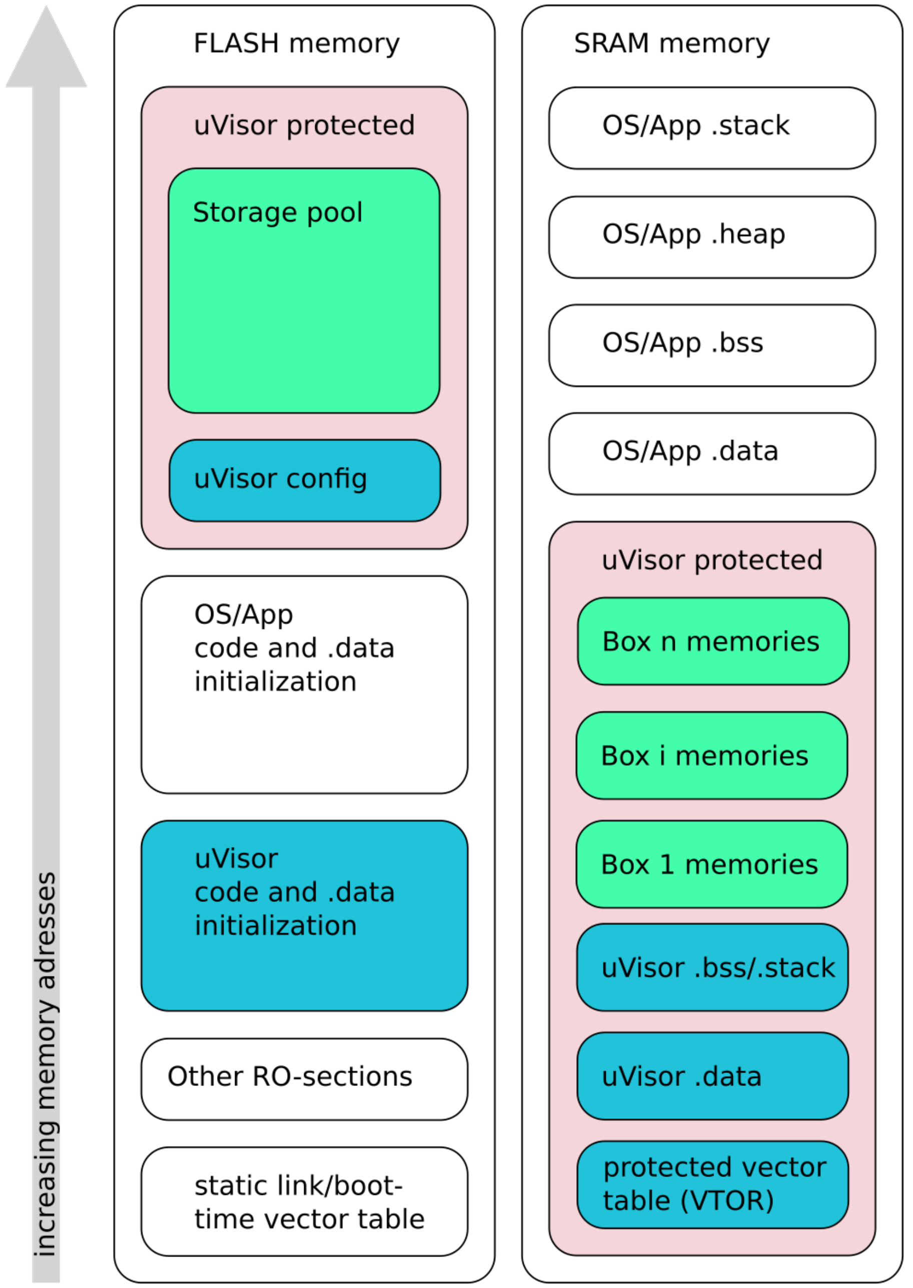


ARM

System security model for microcontrollers

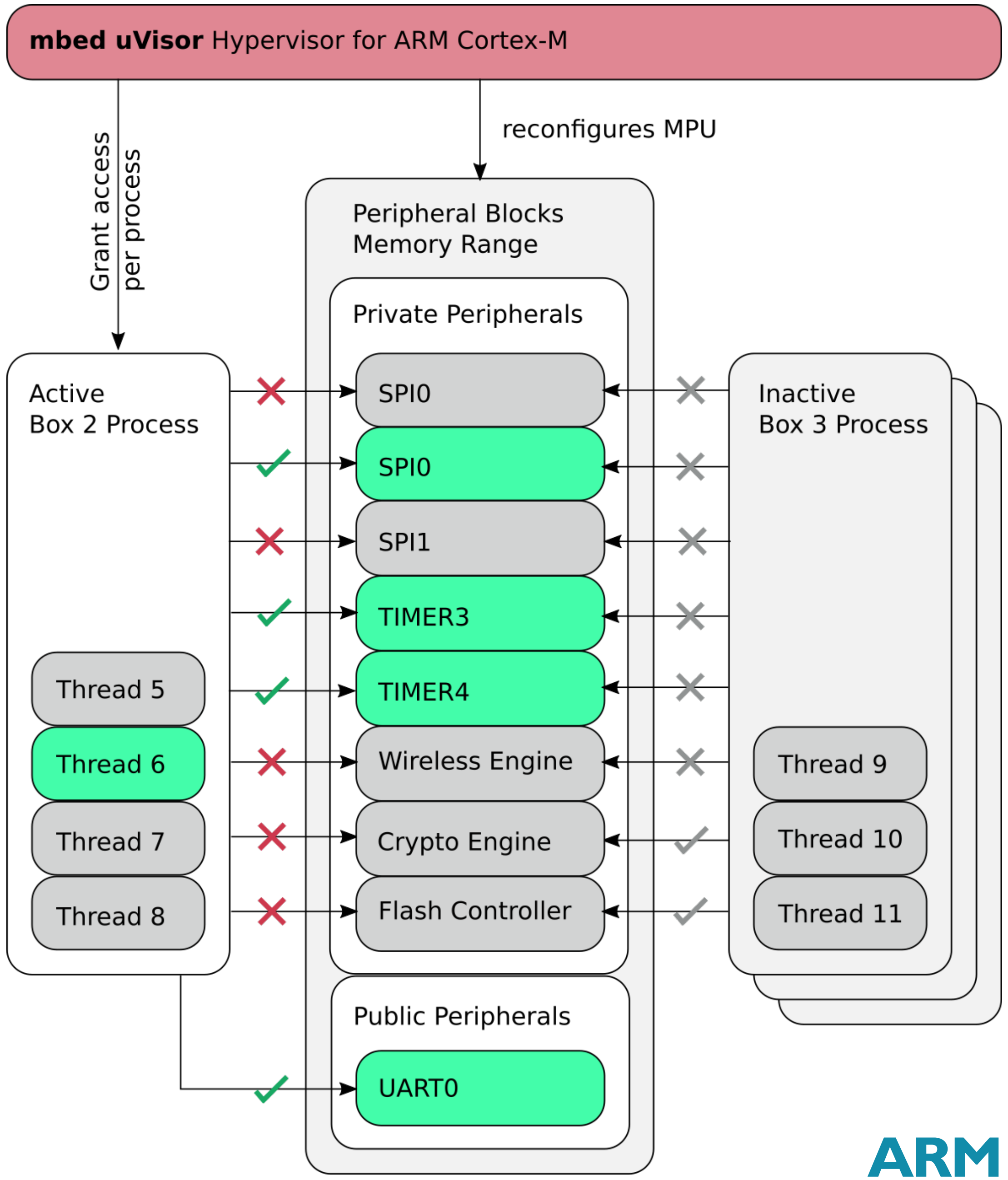
mbed uVisor security

- Security model for ARM Cortex v7M/v8M microcontrollers compatible with Cortex A-class and server operating systems like Linux.
- mbed uVisor allocates protected per-box stacks and detects under-/overflows during operation.
- Public box memories accessible to all boxes for backward compatible APIs.
- Per-Box data sections are protected by default:
  - Secure per-box context memory, stack and heap.
  - Shared data/peripherals with other boxes on demand.
- mbed uVisor code sections visible to everybody
- Remaining flash memory is made available to the system as a persistent storage pool.
- Write access to flash is only allowed through APIs of a dedicated flash-access box process.



Least privilege: Peripherals

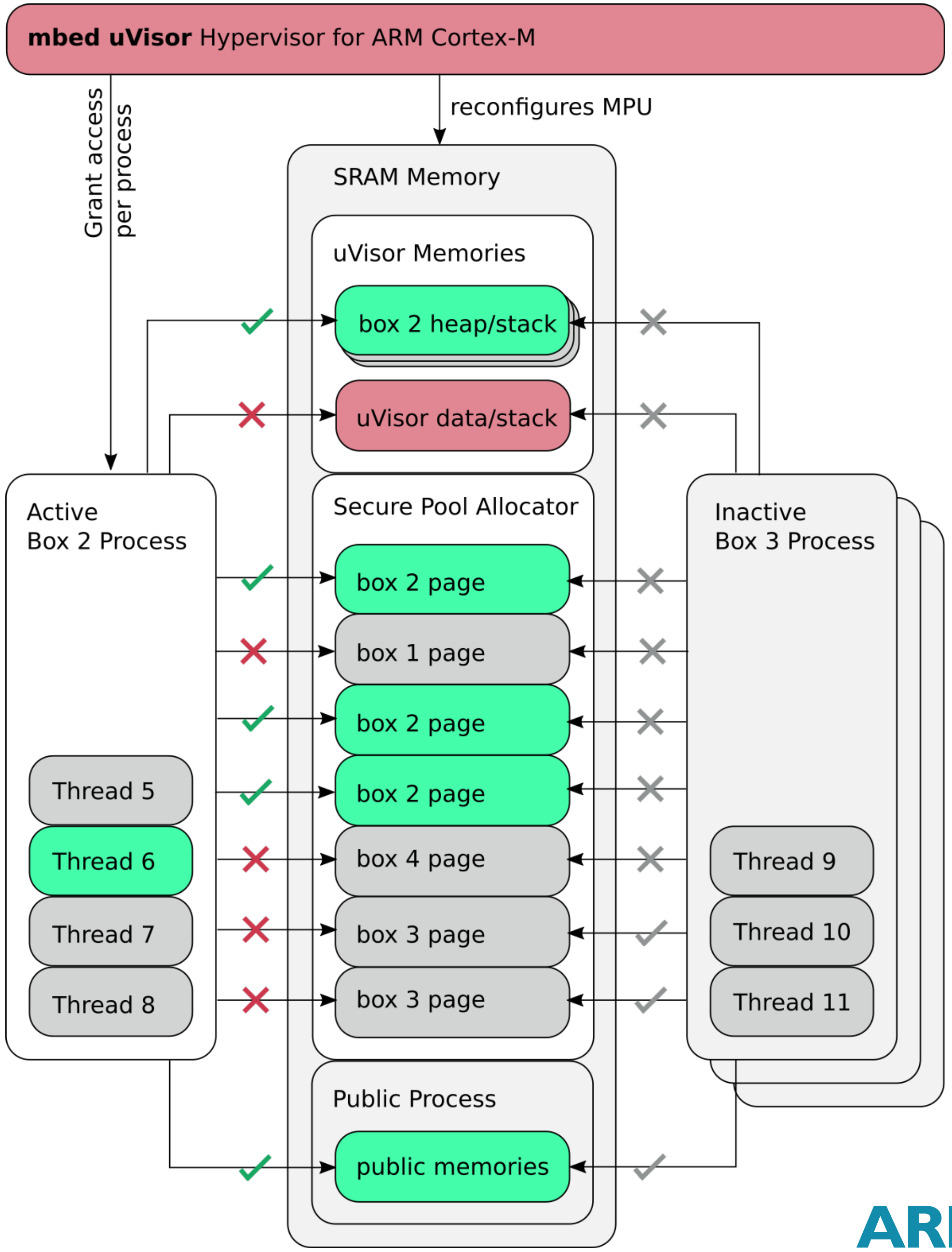
- Mutually distrustful security model:
  - "Principle of least privilege"
  - Boxes are protected against each other.
  - Boxes protected against malicious code from broken system components, driver or other boxes.
- Per-box access control lists (ACL):
  - Restrict access to selected peripherals on a "Need to use basis" for each box.
  - Public box peripherals are accessible to all boxes.
- Hardware-enforced security sandboxes:
  - mbed uVisor manages box-specific ACLs and reconfigures the MPU upon process switch.



Least privilege: Memories

Two-tier memory allocation

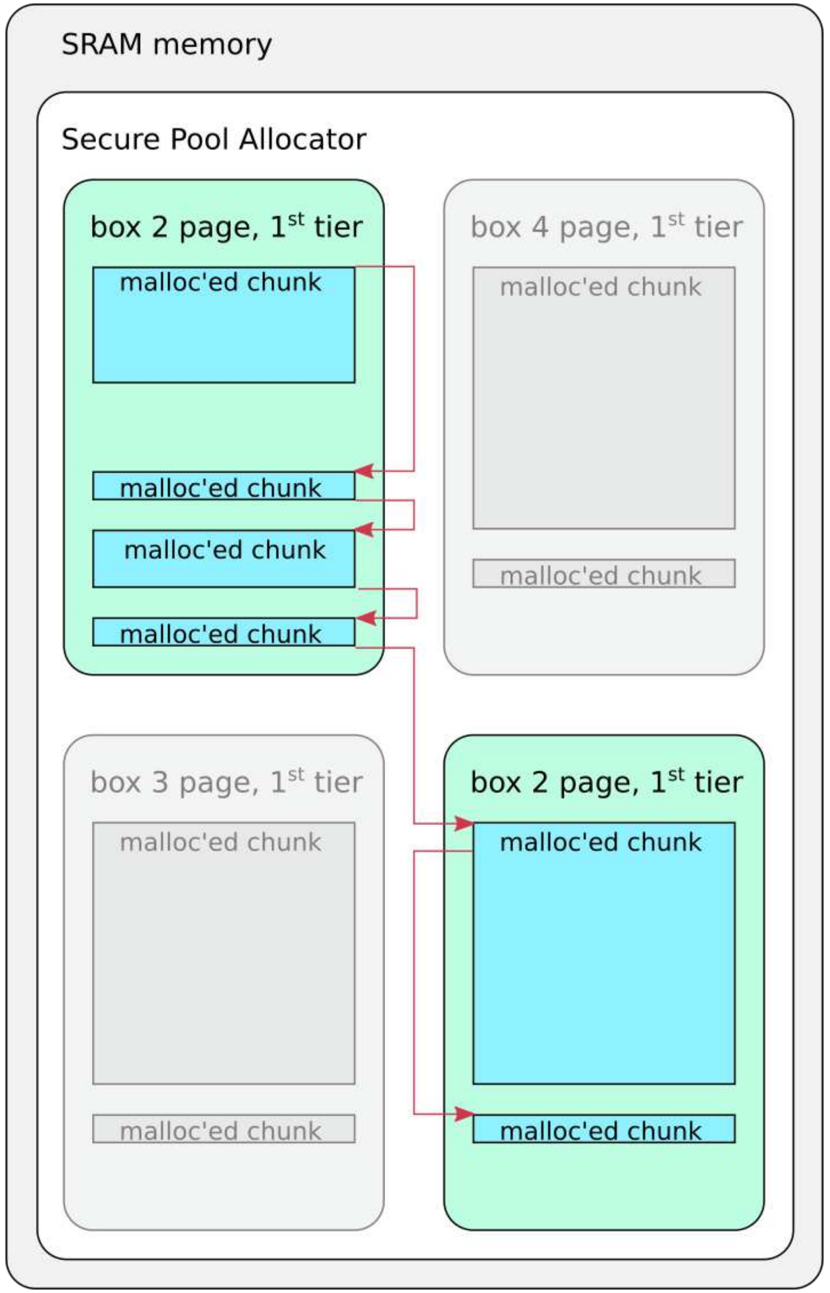
- Static box memories:
  - Boxes commit to heap/stack size of initial box thread using the link time box configuration.
  - All global variables end up in the default box, accessible to all boxes for backward compatibility.
  - The optional secure box context replaces secure global variables. The context size is specified in the box configuration macro.
- Dynamic memories:
  - Each box requests large pages from mbed uVisor.
  - After usage pages are returned to mbed uVisor.
- Secure box data must be copied across boxes or placed into the public box.



Pages for MMU-less systems

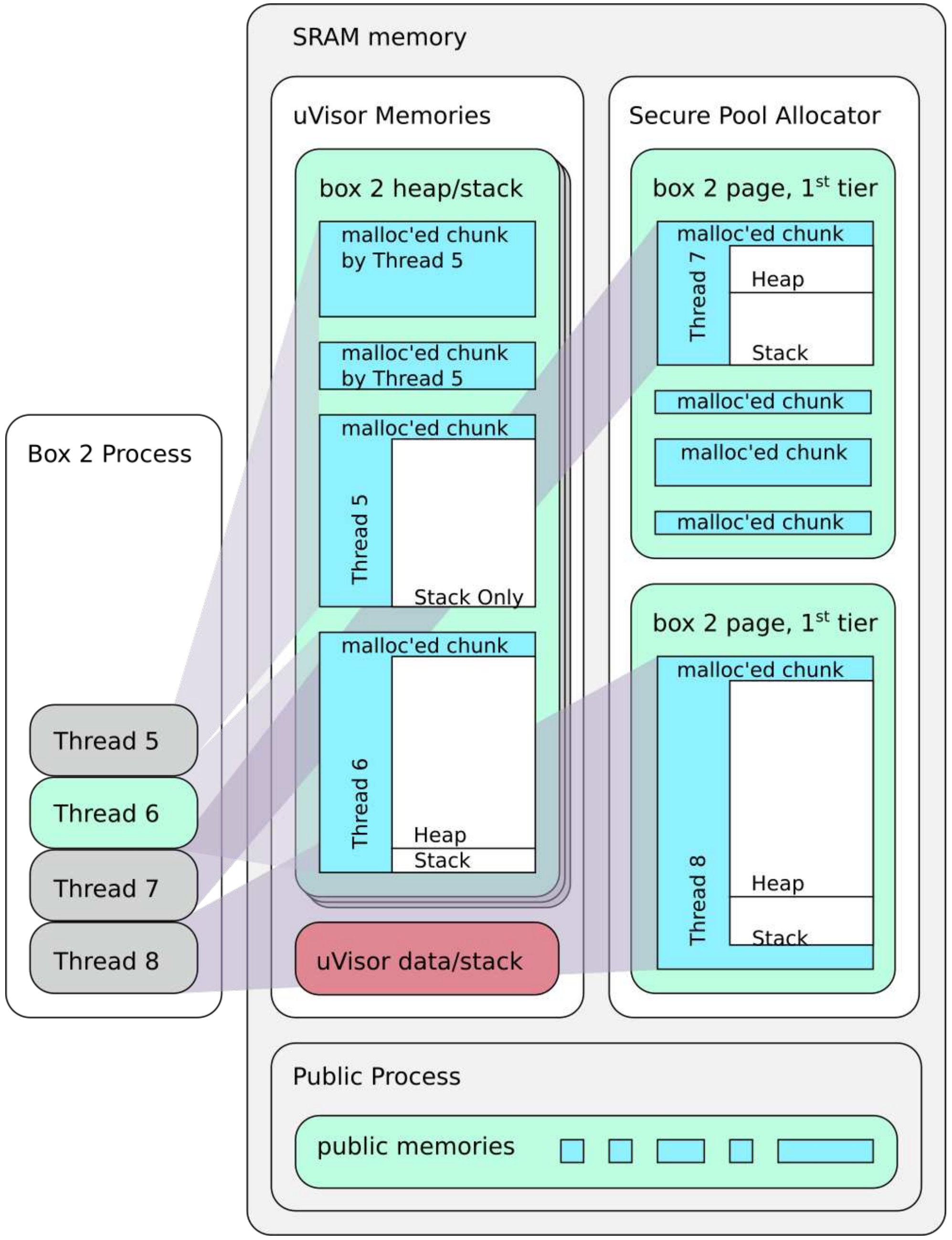
Two-tier memory allocation

- 1st tier page pool is handled by mbed uVisor. Memory security is maintained on a per-page basis
- The guest OS manages the 2nd tier for fine grained allocations.
- 2nd tier allocations are guaranteed to be continuous for backward compatibility as long as they are smaller than the page size.
- mbed uVisor user mode libraries provide a default implementation for the 2nd tier memory allocator.
- The public box allocates pages from the bottom of the page pool to enable continuous allocations.
- Secure boxes allocate from the top of the pool.
- The default page size is SRAM_SIZE divided 16, but can be configured during compile time.



Defeating fragmentation

- Threads allocate stack and heap memories by default in the thread owners box heap.
- Short-lived high-memory threads like crypto libraries get a custom heap for their allocator.
- All 2nd-tier memory allocations are redirected into thread-specific protected heaps:
  - Thread-specific heaps can be allocated on the box heap.
  - Thread-heaps can optionally live in temporary 1st tier pages.
- mbed uVisor switches to the correct thread allocator upon thread switch.
- On thread termination all thread-memories can be released without fragmentation.



“Hello world” Application example

mbed uVisor peripheral sharing across boxes

- main.cpp enables mbed uVisor and configures the public heap size.
- All global variables are accessible to every box for backward compatibility.
- The access control list (ACLs) for the public box is initialized to allow access to non-security-critical system peripherals by the core OS.
- ACLs are white-listed – only listed peripherals are accessible by the public box.

```
#include "uvisor-lib/uvisor-lib.h"
#include "mbed.h"
#include "main-hw.h"

/* Create ACLs for main box. */
MAIN_ACL(g_main_acl);

/* Enable uVisor. */
UVISOR_SET_MODE_ACL(UVISOR_ENABLED, g_main_acl);
UVISOR_SET_PAGE_HEAP(8*1024, 5);

int main(void)
{
    DigitalOut led1(MAIN_LED);
```

https://github.com/ARMmbed/mbed-os-example-uvisor/blob/master/source/main.cpp

ARM

Set up public peripheral ACLs for the main box

- ACLs are declared as arrays of memory start address, peripheral size and permissions.
- Permissions must be either non-overlapping between boxes or marked as shared in all boxes.
- Public peripherals are accessible across all boxes and declared in the public box through ACLs.
- Memories can’t be shared between secure boxes, only from public to secure boxes.

```c
#define MAIN_ACL(acl_list_name) \
    static const UvisorBoxAclItem acl_list_name[] = {     \
        {SIM,    sizeof(*SIM),    UVISOR_TACLDEF_PERIPH}, \
        {OSC,    sizeof(*OSC),    UVISOR_TACLDEF_PERIPH}, \
        {MCG,    sizeof(*MCG),    UVISOR_TACLDEF_PERIPH}, \
        {PORTA,  sizeof(*PORTA),  UVISOR_TACLDEF_PERIPH}, \
        {PORTB,  sizeof(*PORTB),  UVISOR_TACLDEF_PERIPH}, \
        {PORTC,  sizeof(*PORTC),  UVISOR_TACLDEF_PERIPH}, \
        {PORTD,  sizeof(*PORTD),  UVISOR_TACLDEF_PERIPH}, \
        {PORTE,  sizeof(*PORTE),  UVISOR_TACLDEF_PERIPH}, \
        {RTC,    sizeof(*RTC),    UVISOR_TACLDEF_PERIPH}, \
        {LPTMR0, sizeof(*LPTMR0), UVISOR_TACLDEF_PERIPH}, \
        {PIT,    sizeof(*PIT),    UVISOR_TACLDEF_PERIPH}, \
        {SMC,    sizeof(*SMC),    UVISOR_TACLDEF_PERIPH}, \
        {UART0,  sizeof(*UART0),  UVISOR_TACLDEF_PERIPH}, \
        {I2C0,   sizeof(*I2C0),   UVISOR_TACLDEF_PERIPH}, \
        {SPI0,   sizeof(*SPI0),   UVISOR_TACLDEF_PERIPH}, \
    }
```

github.com/ARMmbed/mbed-os-example-uvisor/blob/master/source/main-hw.h ARM

Secure multithreading on CMSIS RTOS

Define thread in a dedicated security context

- Set box Namespace to NULL:
  - Later used for RPC communication.
- Thread main heap is set to 8kb.
- Register my_box_main as thread entry point with normal priority.
- Set stack size to default stack size.
- Declare my_box_context type as secure box context.
- Declare name of this security context to be my_box.
- No additional ACL's defined for peripherals in constant access control list (ACL).

```
typedef struct {
    InterruptIn * sw;
    DigitalOut * led;
    RawSerial * pc;
} my_box_context;

static const UvisorBoxAclItem acl[] = {
};

static void my_box_main(const void *);

UVISOR_BOX_NAMESPACE(NULL);
UVISOR_BOX_HEAPSIZE(8192);
UVISOR_BOX_MAIN(my_box_main, osPriorityNormal, UVISOR_BOX_STACK_SIZE);
UVISOR_BOX_CONFIG(my_box, acl, UVISOR_BOX_STACK_SIZE, my_box_context);
```

https://github.com/ARMmbed/mbed-os-example-uvisor/blob/master/source/led.cpp

Secure thread initialization

```
static void my_box_main(const void *)
{
    /* allocate serial port to ensure that code in this secure box
     * won't touch handle in the default security context when printing */
    RawSerial *pc;
    if(!(pc = new RawSerial(USBTX, USBRX)))
        return;
    /* remember serial driver for IRQ routine */
    uvisor_ctx->pc = pc;
```

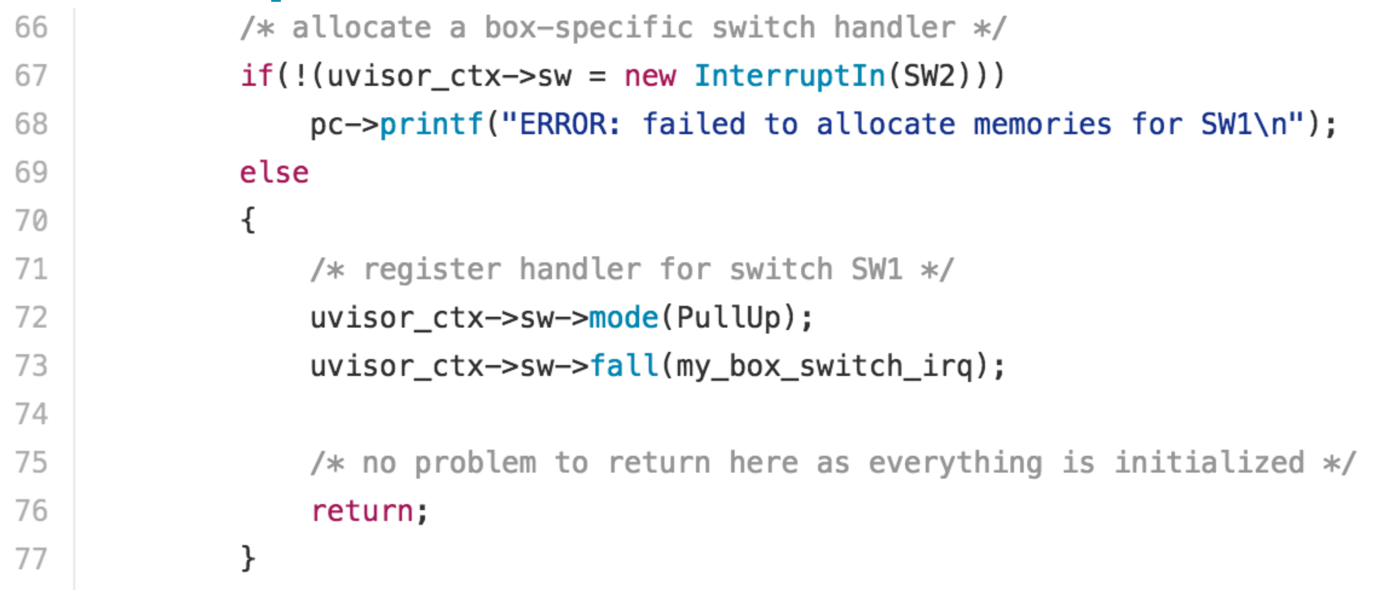
- The my_box_main box entry point will be executed in the boxes security context
- First of al the RawSerial UART class is created in the thread.
- The memory allocation for the class creation is redirected to main thread heap and thus in a memory that is exclusive to my_box_main.
- Each process gets a uvisor_ctx pointer for free at the box-specific type of box_context.
- Data stored in uvisor_ctx is exclusive to the active box.

https://github.com/ARMmbed/mbed-os-example-uvisor/blob/master/source/led.cpp ARM

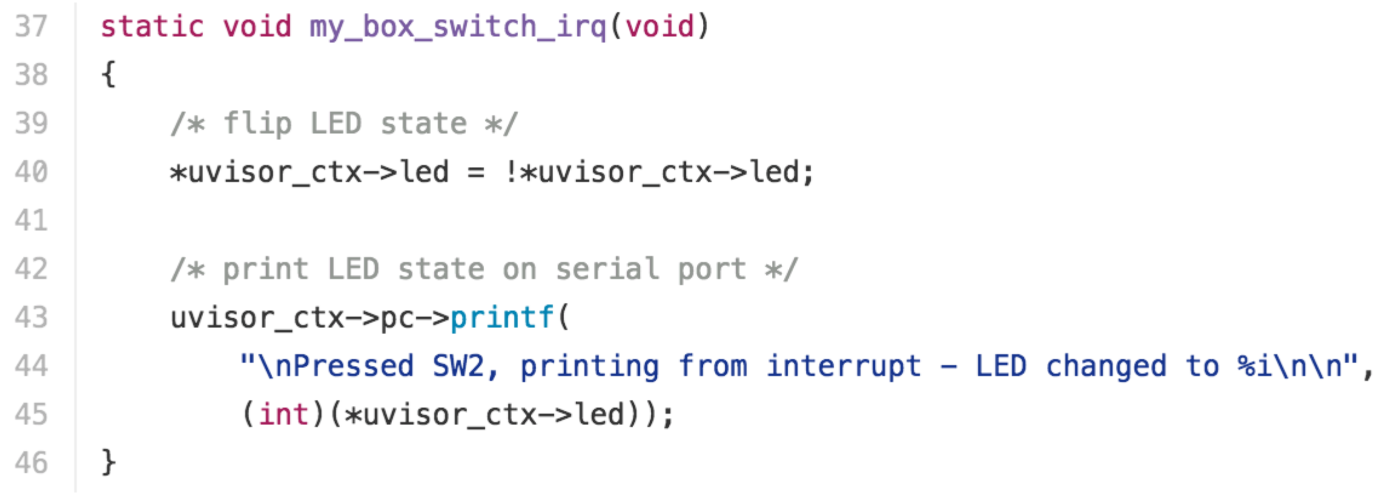
Secure interrupt handling

Secure mbed uVisor interrupt API

- Interrupt ownership is exclusive – multiple boxes cannot register for the same interrupt.
- Registration is based on first-come-first-serve: sharing IRQ's only possible via custom box service APIs .
- mbed uVisor remembers the association between each Interrupt handler and the box security context during registration: other boxes can't use that interrupt until it is released.
- Handlers are executed as unprivileged C functions within the process that registered the IRQ.



```
        /* allocate a box-specific switch handler */
        if(!(uvisor_ctx->sw = new InterruptIn(SW2)))
            pc->printf("ERROR: failed to allocate memories for SW1\n");
        else
        {
            /* register handler for switch SW1 */
            uvisor_ctx->sw->mode(PullUp);
            uvisor_ctx->sw->fall(my_box_switch_irq);

            /* no problem to return here as everything is initialized */
            return;
        }
```



```
static void my_box_switch_irq(void)
{
    /* flip LED state */
    *uvisor_ctx->led = !*uvisor_ctx->led;

    /* print LED state on serial port */
    uvisor_ctx->pc->printf(
        "\nPressed SW2, printing from interrupt - LED changed to %i\n\n",
        (int)(*uvisor_ctx->led));
}
```

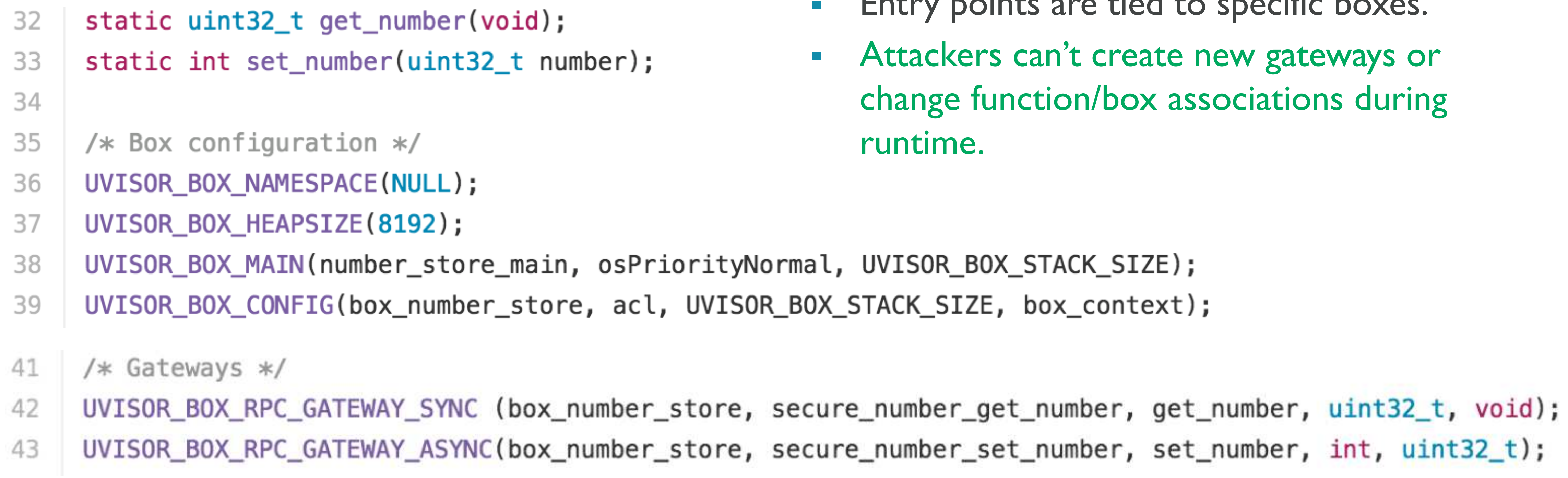
https://github.com/ARMmbed/mbed-os-example-uvisor/blob/master/source/led.cpp

Function calls across security domains

ARM

RPC: secure remote procedure calls

- mbed uVisor forwards calls to boxes
- Both synchronous and asynchronous secure RPC calls are supported.



```c
static uint32_t get_number(void);
static int set_number(uint32_t number);

/* Box configuration */
UVISOR_BOX_NAMESPACE(NULL);
UVISOR_BOX_HEAPSIZE(8192);
UVISOR_BOX_MAIN(number_store_main, osPriorityNormal, UVISOR_BOX_STACK_SIZE);
UVISOR_BOX_CONFIG(box_number_store, acl, UVISOR_BOX_STACK_SIZE, box_context);
```

```c
/* Gateways */
UVISOR_BOX_RPC_GATEWAY_SYNC (box_number_store, secure_number_get_number, get_number, uint32_t, void);
UVISOR_BOX_RPC_GATEWAY_ASYNC(box_number_store, secure_number_set_number, set_number, int, uint32_t);
```

- UVISOR_BOX_RPC_GATEWAY_* macros are used to declare valid API entry points during compile time.
  - Entry points are tied to specific boxes.
  - Attackers can't create new gateways or change function/box associations during runtime.

https://github.com/ARMmbed/mbed-os-example-uvisor-number-store

RPC: secure remote procedure calls

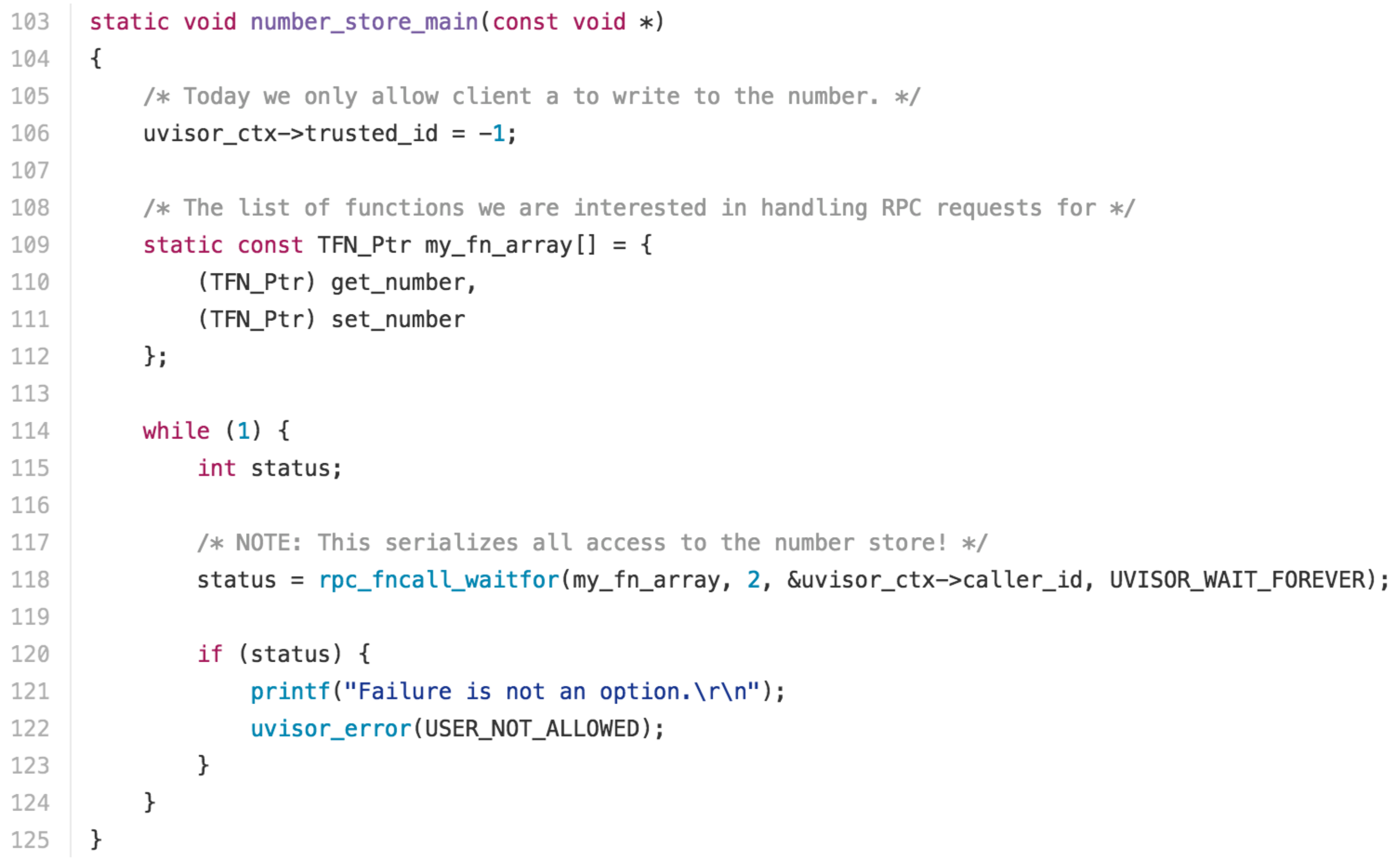
- Callee verifies the caller identity by resolving the box namespace.
- Verification result is cached.
- Caller box-name can be only defined during compile time.
- Callee can impose arbitrary restrictions based on the caller ID and the ownership and origin of an object.

```c
static int set_number(uint32_t number)
{
    const int id = get_caller_id();

    /* Cache the name verification result. This allows future checks to replace
     * a relatively more expensive string compare with a cheaper integer
     * comparison. */
    if (uvisor_ctx->trusted_id == -1) {
        char name[UVISOR_MAX_BOX_NAMESPACE_LENGTH];
        memset(name, 0, sizeof(name));
        uvisor_box_namespace(id, name, sizeof(name));
        /* We only trust client a. */
        static const char * trusted_namespace = "client_a";
        if (memcmp(name, trusted_namespace, sizeof(*trusted_namespace)) == 0) {
            uvisor_ctx->trusted_id = id;
            printf("Trusted client a has box id %u\n", id);
        } else {
            return 1;
        }
    }
    if (uvisor_ctx->trusted_id != id) {
        /* This box is not allowed to write to the secret number. */
        return 1;
    }
```

https://github.com/ARMmbed/mbed-os-example-uvisor-number-store

RPC: secure remote procedure calls

- Callee waits in one or more threads on remote function calls.
- Only one call executed at a time per thread to protect thread stack.
- Remote call executed thread-safe in waiting thread.
- Similar to Posix poll function – callee waits on array of function pointers.



```c
static void number_store_main(const void *)
{
    /* Today we only allow client a to write to the number. */
    uvisor_ctx->trusted_id = -1;

    /* The list of functions we are interested in handling RPC requests for */
    static const TFN_Ptr my_fn_array[] = {
        (TFN_Ptr) get_number,
        (TFN_Ptr) set_number
    };

    while (1) {
        int status;

        /* NOTE: This serializes all access to the number store! */
        status = rpc_fncall_waitfor(my_fn_array, 2, &uvisor_ctx->caller_id, UVISOR_WAIT_FOREVER);

        if (status) {
            printf("Failure is not an option.\r\n");
            uvisor_error(USER_NOT_ALLOWED);
        }
    }
}
```

https://github.com/ARMmbed/mbed-os-example-uvisor-number-store

Securely shared peripheral registers

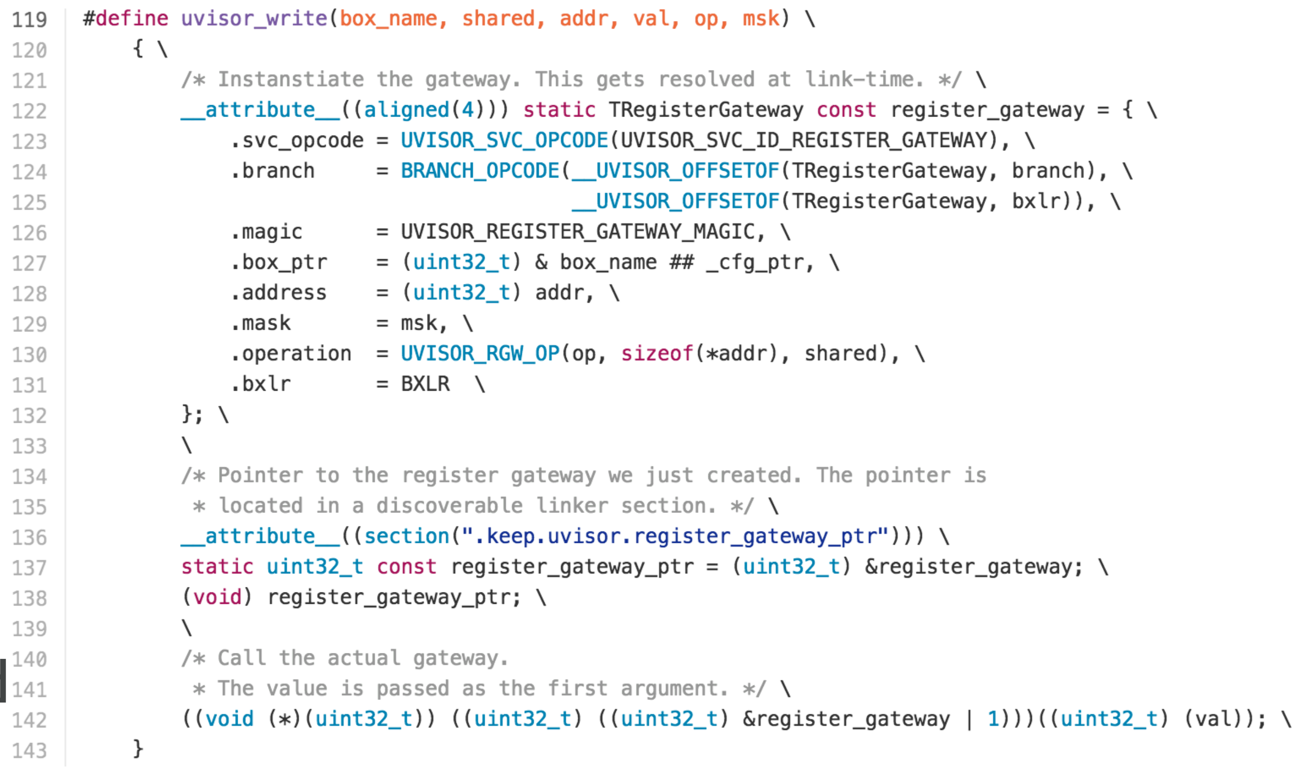
Security-agnostic register level access functions

- Fine grained peripheral access down to register/bit level:
  - Alternative method to peripheral ACL's – mbed uVisor executes access on behalf of the box.
  - Multiple boxes can have access to selected bits in one register.
  - Perfectly suitable for shared GPIO and clock registers to stop interference and DoS.
- Convenient portable wrapper functions:
  - SECURE_WRITE(address, value)
  - SECURE_READ(address)
  - SECURE_BITS_GET(address, mask)
  - SECURE_BITS_CHECK(address, mask)
  - SECURE_BITS_SET(address, mask)
  - SECURE_BITS_CLEAR(address, mask)
  - SECURE_BITS_SET_VALUE(address, mask, value)
  - SECURE_BITS_TOGGLE(address, mask)

github.com/mbedmicro/mbed/blob/master/hal/targets/cmsis/core_cmSecureAccess.h

Register level access security

- Limits access to specific register bits to the specific caller boxes.
- Call gateways only accepted by mbed uVisor from flash memory:
  - Attacker has no write access to flash controller, can't make up new gateways.
- Gateways verified during firmware update or before firmware signing.
- Metadata of register gateways at a fixed offset from mbed uVisor gateway context switch – a SVC supervisor call.
  - Contains pointer to register and access mask, guaranteed verification latency for cross-box calls.



```c
#define uvisor_write(box_name, shared, addr, val, op, msk) \
    { \
        /* Instanstiate the gateway. This gets resolved at link-time. */ \
        __attribute__((aligned(4))) static TRegisterGateway const register_gateway = { \
            .svc_opcode = UVISOR_SVC_OPCODE(UVISOR_SVC_ID_REGISTER_GATEWAY), \
            .branch     = BRANCH_OPCODE(__UVISOR_OFFSETOF(TRegisterGateway, branch), \
                                        __UVISOR_OFFSETOF(TRegisterGateway, bxlr)), \
            .magic      = UVISOR_REGISTER_GATEWAY_MAGIC, \
            .box_ptr    = (uint32_t) & box_name ## _cfg_ptr, \
            .address    = (uint32_t) addr, \
            .mask       = msk, \
            .operation  = UVISOR_RGW_OP(op, sizeof(*addr), shared), \
            .bxlr       = BXLR  \
        }; \
        \
        /* Pointer to the register gateway we just created. The pointer is
         * located in a discoverable linker section. */ \
        __attribute__((section(".keep.uvisor.register_gateway_ptr"))) \
        static uint32_t const register_gateway_ptr = (uint32_t) &register_gateway; \
        (void) register_gateway_ptr; \
        \
        /* Call the actual gateway.
         * The value is passed as the first argument. */ \
        ((void (*)(uint32_t)) ((uint32_t) ((uint32_t) &register_gateway | 1)))((uint32_t) (val)); \
    }
```

github.com/mbedmicro/mbed/blob/master/features/FEATURE_UVISOR/includes/uvisor/api/inc/register_gateway.h

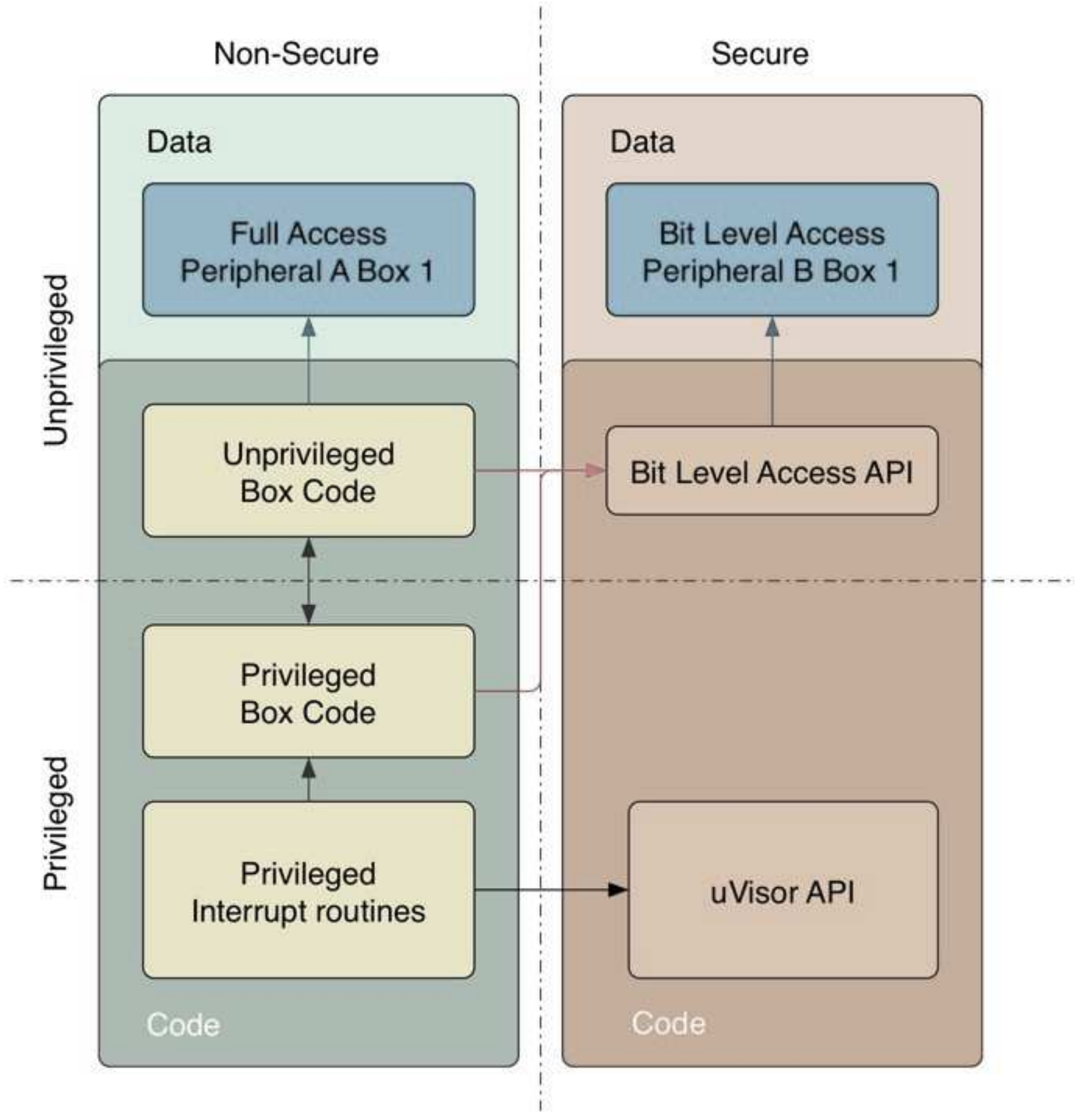
mbed uVisor: Future outlook

mbed uVisor on TrustZone-M

- ARM mbed uVisor application security model of TrustZone for ARMv8M is source-compatible with the ARMv7-M security model.
- Additionally TrustZone for ARMv8M enables bus level protection in hardware:
  - ARMv7-M requires software API filters for DMA access and other security critical operations.
  - ARMv8-M can filter for DMA access for requests initiated by unprivileged code on bus level.
- TrustZone for ARMv8M MPU banking reduces complexity of secure target OS:
  - Secure OS partition owns a private MPU with full control.
  - OS keeps the privileged mode for fast IRQs.
  - Fast interrupt routing and register clearing in hardware.
  - Fast cross-box calls on TrustZone for ARMv8M – optimized call gateways.

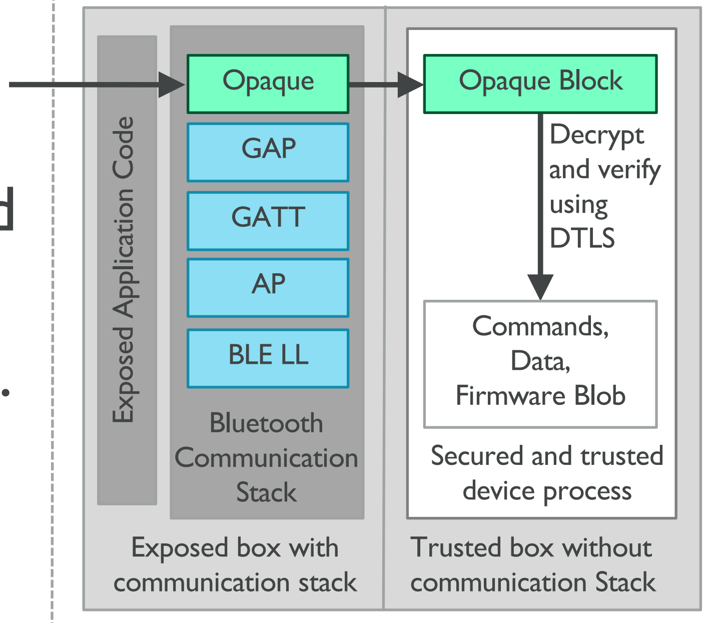
Secure device services on TrustZone-M: SDS

- Use hardware-accelerated security context switching for low latency system services.
  - Secure Interrupt management
  - Secure GPIO access (pin-wise access)
  - Register Level / Bit Level access gateway
  - IPC
  - DMA-APIs
  - Shared Crypto Accelerators / Crypto API
  - Random Entropy Pool Drivers
  - Key Provisioning / Storage
  - Configuration Storage APIs
  - … and many more



Case study: Secure server communication

- Trusted messages contain commands, data or firmware parts.
- Box security not affected by communication stack exploits or infections outside of trusted box.
- Payload delivery is agnostic of protocol stack.
- Resilient box communication over the available channels:
  - Ethernet, CAN-Bus, USB, Serial
  - Bluetooth, Wi-Fi, ZigBee, 6LoWPAN

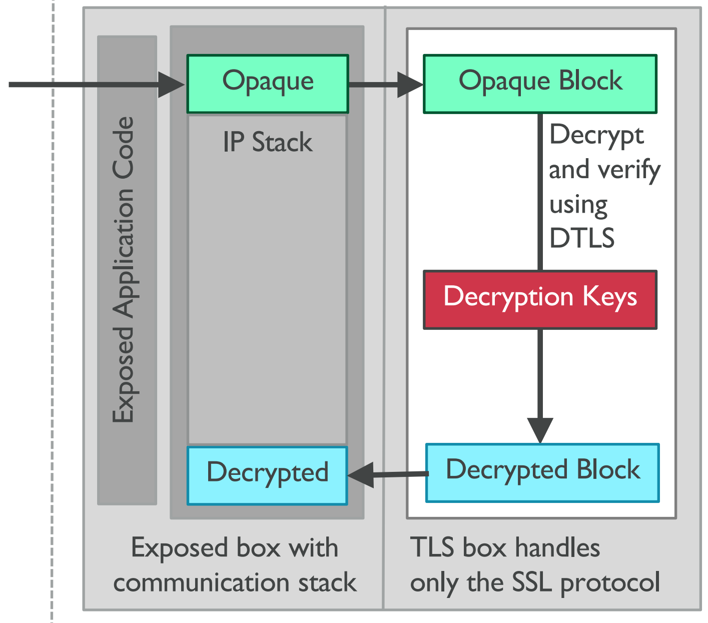


Exposed Application Code
Opaque
GAP
GATT
AP
BLE LL
Bluetooth Communication Stack
Opaque Block
Decrypt and verify using DTLS
Commands, Data, Firmware Blob
Secured and trusted device process
Exposed box with communication stack
Trusted box without communication Stack

IoT Device owned by user.
Initial identity provisioned by System Integrator
Messages delivered agnostic of communication stack

Case study: Secure server communication

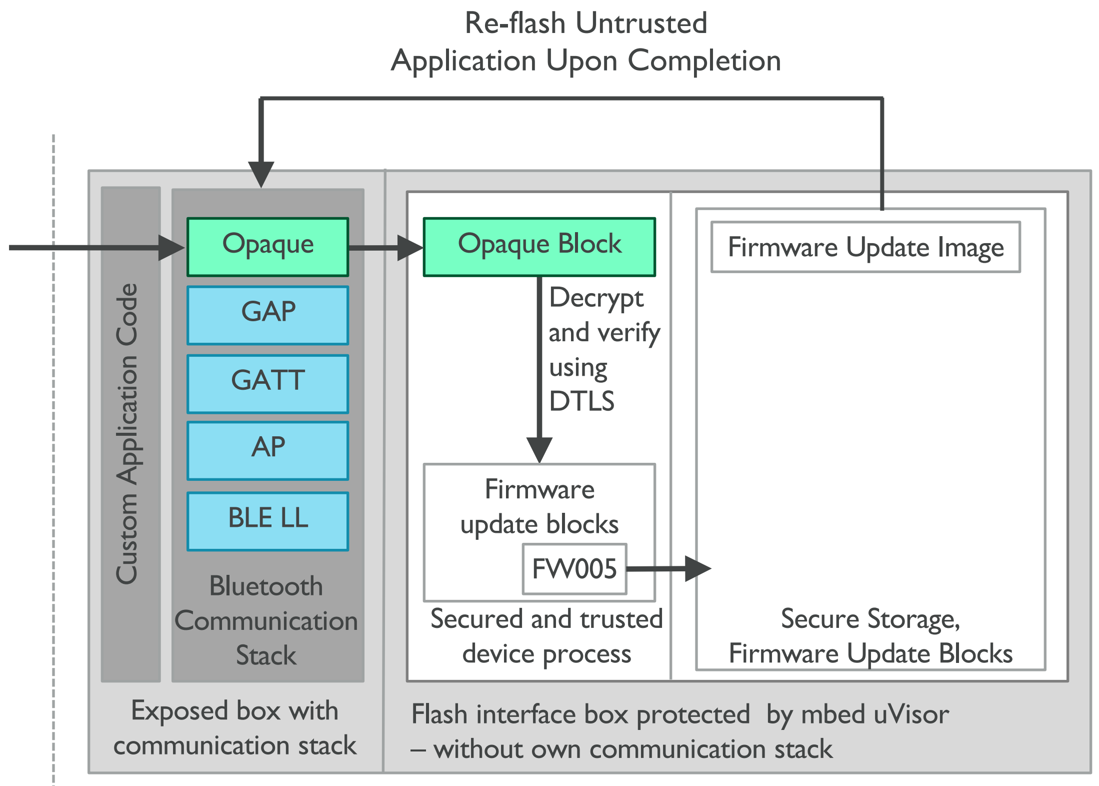
- Communication protected by mbed TLS.
- Raw message payloads decrypted and verified directly by protected code:
  - mbed TLS box not exposed to communication protocol stack bugs.
  - No interference by other boxes.
  - Low attack surface.
- Authentication and encryption keys are protected against malware.
- Malware can't interfere without knowing the encryption or signing keys.


Exposed Application Code
Opaque
IP Stack
Decrypted
Opaque Block
Decrypt and verify using DTLS
Decryption Keys
Decrypted Block
Exposed box with communication stack
TLS box handles only the SSL protocol

Initial keys provisioned by System Integrator. Messages decoded independent of stacks using mbed TLS in separate security context

Case study: Secure remote firmware update

- Delivery of firmware update must be decoupled from a protocol-independent firmware image verification.
- Bugs in communication stacks or cloud infrastructure must not compromise the firmware update:
  - End-to-end security for firmware updates between the firmware developer and the one secure box on the device with exclusive flash-write-access ACLs.
  - Box with flash controller ACLs only needs the public update key to verify validity of firmware.
  - Local malware can't forge a valid firmware signature to the firmware update box, the required private firmware signature key is not in the device.


Re-flash Untrusted Application Upon Completion
Custom Application Code
Opaque
GAP
GATT
AP
BLE LL
Bluetooth Communication Stack
Opaque Block
Decrypt and verify using DTLS
Firmware update blocks
FW005
Secured and trusted device process
Firmware Update Image
Secure Storage, Firmware Update Blocks
Exposed box with communication stack
Flash interface box protected by mbed uVisor – without own communication stack

IoT device owned by user,
Initial identity provisioned by System Integrator,
Messages delivered independent of stacks

Case study: Controlled malware recovery

- Secure box can remotely recover from malware:
  - Enforces communication through the exposed side to the server.
  - Receives latest security rules and virus behaviour fingerprints for detection.
  - Shares detected pattern fingerprint matches with control server
  - Distributed detection of viruses and live infrastructure attacks.
  - Thanks to flash controller ACL restrictions, malware can't modify monitor code or install itself into non-volatile memories.
- When communication with the server breaks for a minimum time:
  - Parts of the device stack are reset to a known-good state.
  - Reset prevents malware from staying on the device.
  - Device switches to a safe mode to rule out network problems or to remotely update the firmware via reboot if needed.

https://commons.wikimedia.org/wiki/File:Biohazard.svg

“The mantra of any good security engineer is: **“Security is not a product, but a process”** It's more than designing strong cryptography into a system; it's designing the entire system such that all security measures, including cryptography, work together.”

– Bruce Schneier

Thank you – Questions?

Fork me on GitHub

ARM

Get the latest information on mbed security…
https://www.mbed.com/en/technologies/security/

... follow live mbed uVisor development on github.com:
https://github.com/ARMmbed/uvisor/
More presentations and in-depth OS-level integration and porting documents can be found there as well.

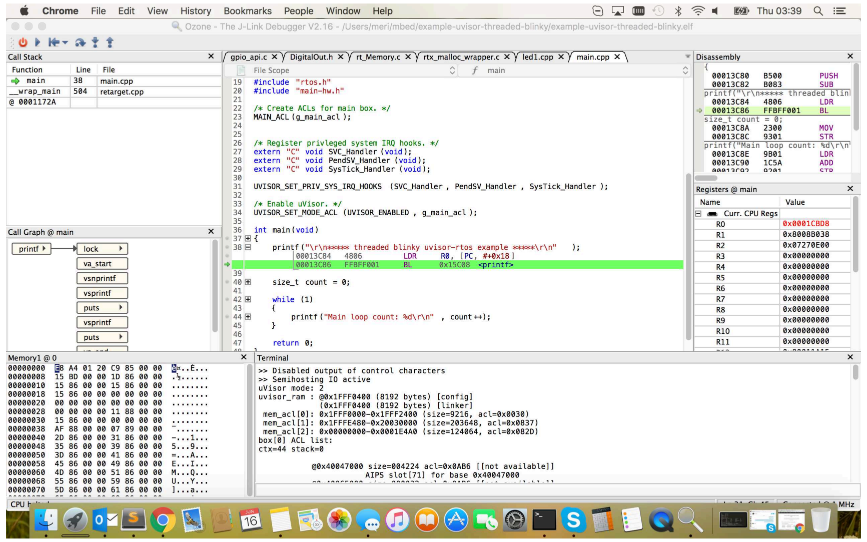
Contact: Milosch Meriac <milosch.meriac@arm.com>

Support slide:
Getting started

ARM

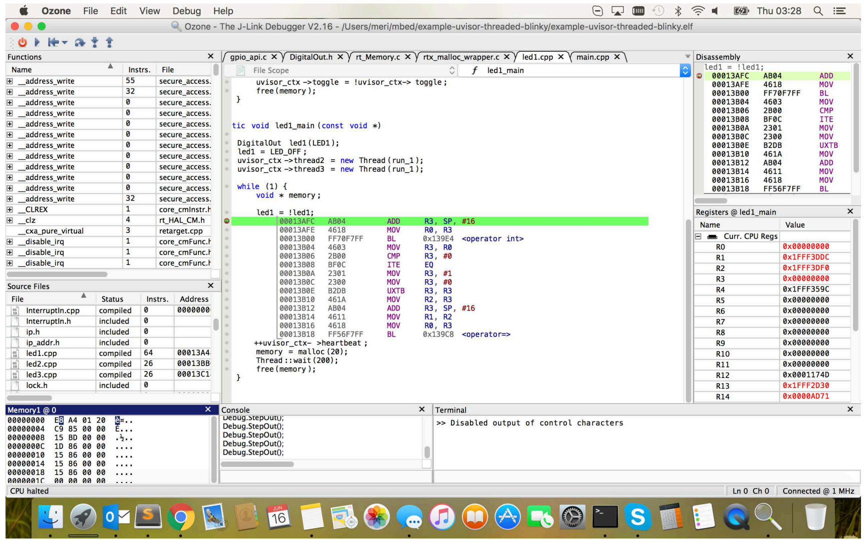
uVisor: development basics

- uVisor debugging is platform independent via Semi-hosting. Please ensure that your debugger of choice supports semihosting and that its turned on.
- Please use the secure threaded blinky example for starting uVisor development.
- For release builds, please enter:
  - `mbed compile -m K64F_SECURE -t GCC_ARM -c`
- When connecting a debugger, output can be seen on the debug console with a debug build:
  - `mbed compile -m K64F_SECURE -t GCC_ARM -c -o debug-info`
- Debug Alternatives for NXP FRDM-K64F:
  - OpenSDA Users Guide: Follow instructions for updating the boot loader. The boot loader firmware image can be downloaded here.
  - Ozone – J-Link compatible debugger, but CodeWarrior, Eclipse, Keil, IAR, KDS should be fine, too.
  - Serial console printf serial enabled at 9600 baud over CDC USB serial interface of the K64F
- For porting uVisor to new platforms, please refer to our porting guide
- You can download a docker-based pre-configured uVisor development here
- **Your device will freeze if your run a debug version of your app as it will wait for a debugger to connect!**



```
#include "rtos.h"
#include "main-hw.h"

/* Create ACLs for main box. */
MAIN_ACL (g_main_acl );


/* Register privleged system IRQ hooks. */
extern "C" void SVC_Handler (void);
extern "C" void PendSV_Handler (void);
extern "C" void SysTick_Handler (void);

UVISOR_SET_PRIV_SYS_IRQ_HOOKS  (SVC_Handler , PendSV_Handler , SysTick_Handler );

/* Enable uVisor. */
UVISOR_SET_MODE_ACL (UVISOR_ENABLED , g_main_acl );

int main(void)
{
    printf("\r\n***** threaded blinky uvisor-rtos example *****\r\n"   );
          00013C84   4806       LDR     R0, [PC, #+0x18]
          00013C86   FFBFF001   BL      0x15C08  <printf>

    size_t count = 0;

    while (1)
    {
        printf("Main loop count: %d\r\n" , count++);
    }

    return 0;
```

```
>> Disabled output of control characters
>> Semihosting IO active
uVisor mode: 2
uvisor_ram : @0x1FFF0400 (8192 bytes) [config]
             (0x1FFF0400 (8192 bytes) [linker]
 mem_acl[0]: 0x1FFF0000-0x1FFF2400 (size=9216, acl=0x0030)
 mem_acl[1]: 0x1FFFE480-0x20030000 (size=203648, acl=0x0837)
 mem_acl[2]: 0x00000000-0x0001E4A0 (size=124064, acl=0x082D)
box[0] ACL list:
ctx=44 stack=0

           @0x40047000 size=004224 acl=0x0AB6 [[not available]]
                 AIPS slot[71] for base 0x40047000
```



Ozone - The J-Link Debugger V2.16 - /Users/meri/mbed/example-uvisor-threaded-blinky/example-uvisor-threaded-blinky.elf
Functions
Source Files
Disassembly
Registers @ led1_main
Memory1 @ 0
Console
Terminal
CPU halted

```
    uvisor_ctx ->toggle = !uvisor_ctx-> toggle;
    free(memory);
}


tic void led1_main (const void *)

 DigitalOut  led1(LED1);
 led1 = LED_OFF;
 uvisor_ctx ->thread2  = new Thread(run_1);
 uvisor_ctx ->thread3  = new Thread(run_1);

 while (1) {
     void * memory;

     led1 = !led1;
          00013AFC   AB04       ADD    R3, SP, #16
          00013AFE   4618       MOV    R0, R3
          00013B00   FF70F7FF   BL     0x139E4  <operator int>
          00013B04   4603       MOV    R3, R0
          00013B06   2B00       CMP    R3, #0
          00013B08   BF0C       ITE    EQ
          00013B0A   2301       MOV    R3, #1
          00013B0C   2300       MOV    R3, #0
          00013B0E   B2DB       UXTB   R3, R3
          00013B10   461A       MOV    R2, R3
          00013B12   AB04       ADD    R3, SP, #16
          00013B14   4611       MOV    R1, R2
          00013B16   4618       MOV    R0, R3
          00013B18   FF56F7FF   BL     0x139C8  <operator=>
     ++uvisor_ctx- >heartbeat;
     memory = malloc(20);
     Thread::wait(200);
     free(memory);
 }
```

```
Debug.StepOut();
Debug.StepOut();
Debug.StepOut();
Debug.StepOut();
Debug.StepOut();
```

```
>> Disabled output of control characters
```