embeddedworld2017
Exhibition&Conference
... it's a smarter world

High-End Security Features for Low-End Microcontrollers

Hardware-security acceleration for multi-domain ARMv8-M systems

Milosch Meriac
Principal Security Engineer,
IoT Group, ARM
Cambridge, UK
milosch.meriac@arm.com

Joseph Yiu
Senior Embedded Technology Manager,
CPU Product Group, ARM
Cambridge, UK
joseph.yiu@arm.com

***Abstract* —ARM® TrustZone® technology for ARM Cortex®-M33 enables systems and their software to be partitioned into multiple security domains. Next generation microcontroller operating systems can benefit from these hardware security features without affecting real time performance.**

This paper presents one of the possible configurations - showing how critical operating system functions can be accelerated with new hardware security features while maintaining the real-time properties of the secure OS. Critical system functions include secure memory allocation, interrupt management, whitelisting of peripheral access, cross-domain-calls and secure boot.

A prototype for such an operating system – the ARM mbed™ OS with uVisor - is currently being developed on GitHub[1] to verify these concepts with practical use cases.

Keywords — ARM TrustZone; Cortex-M33; Security Attribution Unit (SAU); Implementation Defined Attribution Unit (IDAU); Memory Protection Unit (MPU); MPU banking; Bus Level Security; RTOS Security; Real-time Security

I. INTRODUCTION

Although connected microcontroller systems are ubiquitous, the security measures found on these systems commonly lag behind mobile application processors by ten to twenty years. The result of these shortcomings is that potential remote attackers have an easy game to escalate application bugs to system privileges. Once malware becomes resident on a microcontroller, the system turns irrecoverable for a remote system manager – instead engineers must be sent around with laptops and programming dongles for recovering these devices, making the cost of recovery very expensive.

On high-end embedded systems using application processors, security solutions like SeLinux enable fine-grained access control and process separation by implementation of the 'Principle of Least Privilege'.

This countermeasure ensures that the effects of a successful attack are restricted to a small portion of the system and cannot be easily escalated to other parts.

As long as the attacker cannot escalate to the remote recovery context, devices can be remotely recovered by automated processes.

On these high-end systems such measures are often secured by using memory management units (MMU). Although in the case of ARM TrustZone for Cortex-A usually a single security context is used, it is seldom virtualized between security domains. By using the MMU hardware, the operating system can control, on a fine-grained level, which peripherals and memories are accessible by which context. Communication between these isolated contexts can be strictly controlled. Security reviews of such systems therefore can be focused on these inter-context interfaces.

In the absence of an MMU, memory protection units (MPUs) can be used to protect critical microcontroller software - offering isolation whilst maintaining deterministic real-time response and the small footprint required in microcontroller applications.

Embedded operating systems for ARM microcontrollers have recently started to use MPUs for implementing secure spatial and temporal process isolation. Commonly such isolation can come at a very high cost in terms of call-latencies and loss of real-time properties. However, with the introduction of TrustZone for ARMv8-M, hardware security features that were previously only available to application processors are now available for microcontrollers.

We present how granular and lightweight system level security components can be used in combination with the new

[1] uVisor can be found at https://github.com/ARMmbed/uvisor

ARMv8-M architecture-based Cortex-M33 processor. We will demonstrate how an operating system can use ARM TrustZone technology, banked MPUs and AMBA AHB5 interconnect to implement process isolation, without impact on the real-time capability of the system.

We also address acceleration of common security features like dynamic secure memory allocation during runtime, flexible and secure inter-domain communication and peripheral protection on a per-security-domain level on MMU-less microcontrollers.

In addition, we'll present how ARM TrustZone for ARMv8-M technology enables secure boot and allows microcontroller systems to recover, even when part of the system is compromised.

II. TrustZone for ARMv8-M Microcontrollers

ARM Cortex-M23 and Cortex-M33 processors are both based on the ARMv8-M architecture and use the same memory map as the previous generation of Cortex-M processors based on the ARMv7-M/ARMv6-M architecture.

TrustZone security functions are now part of the ARMv8-M architecture. In a TrustZone-enabled system, software is divided into Secure and Non-secure domains:

- *Secure software* executes from the Secure address space and can access both Secure and Non-secure addresses. Secure software typically contains components that are critical to the security of a device such as secure-boot, device provisioning, encryption libraries and firmware update. The processor is in the Secure state when executing Secure software.
- *Non-secure software* executes from Non-secure address spaces and can only access Non-secure memories and peripherals. Traditional application code is likely to be executed in the Non-secure state. The processor is in the Non-secure state when executing Non-secure software. In this paper we describe how the Non-secure state can be split between multiple mutually distrustful domains.

The partitioning of the address space is handled by the processor using a programmable unit called the Security Attribution Unit (SAU) in combination with a device specific logic block called the Implementation Defined Attribution Unit (IDAU). The memory space security configuration can only be changed by software running in the Secure state.

Software functions running on the system are executed either in Secure or Non-secure state:

- Secure and Non-secure software can interact with each other using direct function calls
- Exception handling can switch processor states transparently.

The separation between Non-secure and Secure software execution can be made visible to the wider bus system:

III. Bus Level Security

Much like TrustZone on traditional ARMv7-A and ARMv8-A application CPUs, hardware security is now extended to the peripheral bus system of the microcontroller. Each component on the bus can verify and propagate the security level for each bus operation. Each access to the bus is tagged by the CPU with the security attribute of the address and is depending on the processor state (Secure or Non-secure) when the request was originally started.

In case of derived operations like direct memory access (DMA) controllers, Secure software running on the CPU can program a TrustZone aware DMA controller to mark a DMA

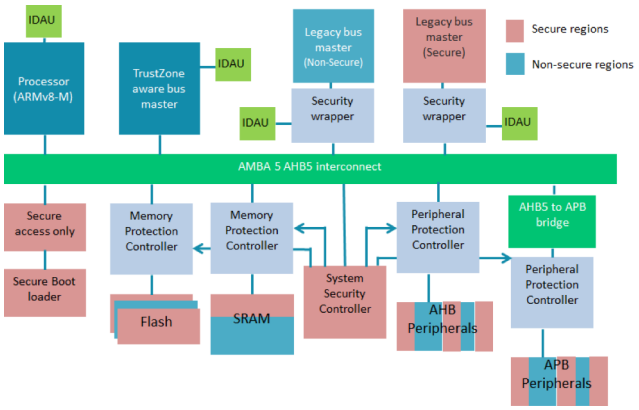

Fig 1. System wide security with AHB5 on chip bus protocol

channel as Secure, which has a Secure transfer attribute propagated by the AHB5 interconnect when accessing Secure address locations. The relevant system components for bus level security are listed below:

A. Block based Memory Protection Controller

For our reference implementation, we recommend using the block based memory controller for protecting internal SRAM memories. The address space of the memory peripheral is partitioned into blocks.

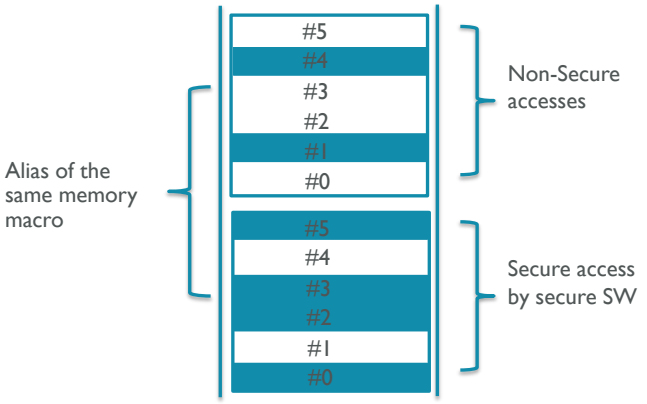

Fig 2. Example behavior of a block based memory protection controller

Each of these blocks is either visible at its Secure alias or at the Non-secure alias of the memory block. Every block

corresponds to one configuration bit in a set of security registers – therefore a register of 32 bits covers 32 pages resulting in 8 kilobytes of memory being protected when assuming the recommended page size of 256 bytes.

For memories larger than 256kB, a block size of 512 bytes is recommended – above 512kB 1kB pages are appropriate. This avoids a significant increase in memory configuration hardware.

The result of this design choice results in Secure and Non-secure memory being distributed across the memory map of the SRAM. Access to holes in the map – either by the CPU or by bus master peripherals (like DMA engines) - must result in bus faults. This way software mistakes - for example accidentally initiating a DMA transfer over a memory hole in SRAM - can be discovered as early as possible to avert attacks.

B. Watermark based Memory Protection Controller

For external S(D)RAM memories or large flash memories, a watermark based approach can be used to partition the memory between the Secure and Non-secure alias.

All memories below the watermark offset are mapped to the Secure alias – all memories above the watermark are mapped into the Non-secure alias of the memory.

Watermark protection is useful for use with flash memories for hiding security configuration storage and encryption keys from application code. Access privileges for each item can be then verified individually depending on the caller context as part of a storage API.

For systems with firmware update functionality (multiple firmware slots) a watermark-protection is not feasible. For all non-trivial non-volatile-memory (NVM) use cases a block based filter with the granularity of the NVM's erase size is required instead.

C. Peripheral Protection Controller

Like the block based memory protection controller mentioned above, each peripheral is individually marked as Secure, Non-secure, and additionally marked as privileged or non-privileged using simple bit masks. Depending on the security status of that bit, each peripheral block is mapped either into the Secure or Non-secure peripheral range.

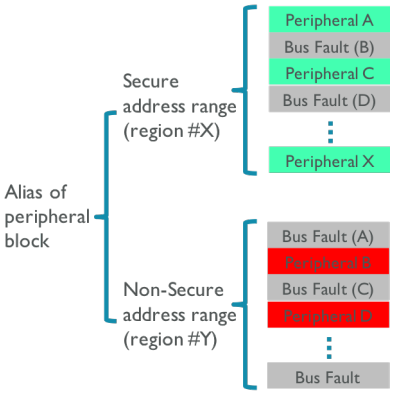

Fig 3. Example behavior of a peripheral protection controller

A bus-fault will occur if the CPU or bus master peripherals (like a DMA engine) try to access the holes in between mapped peripheral blocks. In the same way as the block based memory protection controller, the bus-fault is detectable, and hence can be used to prevent a security violation.

D. Memory Protection Unit (MPU)

ARMv8-M offers an improved MPU design compared to ARMv7-M designs. The new MPU design provides a very flexible configuration scheme of start and end addresses for protected memory regions with a granularity down to 32 bytes.

As in the previous design, the MPU is used to separately specify access for privileged and unprivileged code. Unprivileged code can be locked out from privilege memories and peripherals. Unprivileged code is also barred from accessing the MPU configuration and therefore cannot circumvent the MPU protection.

Independent of the currently active privilege level of the CPU, the MPU also protects SRAM regions against accidental writes, and prevents code execution from the stack.

Both the Secure and the Non-secure sides own individual MPU instances. Both these MPUs are transparently banked by the architecture according to the currently active CPU state. Although the MPU itself is an optional configuration option, as a minimum we recommend enabling the Secure MPU option to partition privileged and unprivileged code execution.

E. Security Attribution Unit (SAU)

The MPU handles horizontal access between privileged and unprivileged mode. The SAU on the other hand is used for defining access between the Secure and the Non-secure execution state.

SAU settings are therefore orthogonal to MPU settings. Thus, we get four quadrants in our security model – allowing fine-grained configuration of the system security depending on the precise CPU state.

Silicon vendors can define default hardware-specific security settings for peripherals and memories, and they do this in the IDAU. The IDAU is therefore responsible for initial static partitioning of the system according to the needs of a particular device. These settings can then be refined by a Secure software developer using the configurable SAU unit, to meet the needs of their security applications.

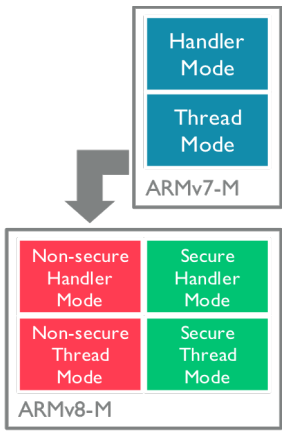

Fig 4. Four quadrants of processor states in ARMv8-M architecture

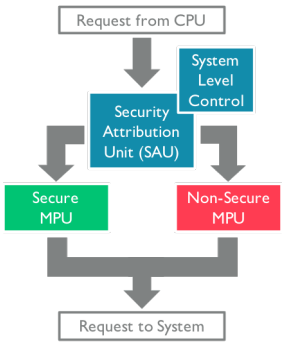

Fig 5. SAU and banked MPU are used together for permission control

The SAU is optional – silicon vendors can choose to enable security configuration dynamism by solely using the IDAU. This can reduce silicon area. We recommend against such a decision as it breaks portability across different platforms and makes it hard for developer and penetration-testers to compare security models across devices from different vendors.

F. Hardware Accelerated Portability – the TT instruction

The "Test Target" (TT) instruction allows discovery of the security state and memory layout of an ARMv8M microcontroller system in a portable way.

For every given address, the access permission and ownership of a memory region can be queried. The TT instruction allows an access query for the current processor state – as well as for all privilege states lower or equal the current state and all security states lower or equal to the current state:

- **TT** – Use current security state and current privilege level
- **TTT** – Privileged software can determine unprivileged access permission of current security state
- **TTA** – Secure software can see the alternate domain's access permission
- **TTAT** – Secure software can see the alternate domain's unprivileged access permission

Every distinct block (i.e. a block with a consistent security setting for the memory) is assigned a unique ID – the IDAU region number. Each peripheral block in the peripheral range must have a distinct number. The main benefit of using the TT instruction is to discover security-discontinuities for a given memory range.

By checking the start and end address of a memory range with two TT-instructions, the operating system can verify if the security-configurations are identical for both addresses:

- if the IDAU-ID's are the same, that tells TT for example that an intended DMA access will not run across two peripherals. This enables detection in the planned transaction – for example an intended DMA request spanning two peripherals.
- the SAU-region ID, in case an SAU region is configured for the tested address. SAU regions are not allowed to overlap and therefore cannot be nested. The same SAU region ID returned for the start and end implies that start and end of the transaction is within the same region.
- the MPU-region ID in case a MPU region is configured for the address to be tested. Like SAU regions, MPU regions cannot overlap and therefore cannot be nested – again a detection of discontinuities is possible here.

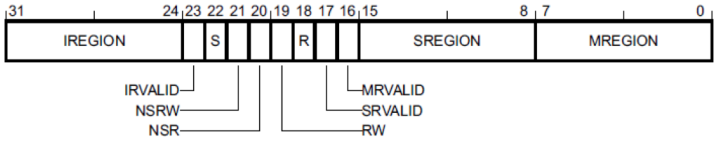

Fig 6. 32-bit result from a TT instruction contains various information

The checks itself are very straight-forward: Perform two TT instructions on both the start and end address of a transaction and compare the results to be identical. The main use case for applying the TT instruction is the security verification of DMA transactions inside one domain or between Secure and Non-secure security domains.

Field	Descriptions
IREGION	IDAU region number
IRVALID	IDAU region valid
S	Secure
NSRW	Non-secure RW
NSR	Non-secure readable
RW	Read and writeable
R	Readable
SRVALID	SAU region valid
MRVALID	MPU region valid
SREGION	SAU region number
MREGION	MPU region number

Both the source and destination are independently verified to ensure that the DMA transaction is within the permission-set of the currently active security domain.

G. Crossing Security Domains – adding SG/BXNS/BLXNS instructions

The secure gateway instruction (SG) marks allowed code entry points on the Secure side for Non-secure callers. This feature is vitally important for two reasons:

- the SG instruction prevents callers to jump past a security check on the Non-secure side
- the SG instruction also switches the security state from Non-secure to Secure. This enables a Non-secure caller to access Secure resources, but enables application-specific restrictions for each individual call.

The SG-instruction is designed to be transparent both for the caller and the function return. From the software developer's perspective, the called security function simply returns from the call on completion. The CPU transparently ensures that privileges are switched back to the Non-secure caller when returning. The return address must be protected by the SAU or IDAU on the Secure stack – the OS therefore needs to be sure to configure its memories appropriately.

Independent of the SG-instruction, Secure code can jump or call into Non-secure state – with the reverse process: Execution can be handed back from the Non-secure callback function to the Secure caller if necessary (using the BLXNS instruction).

H. Crossing Security Domains – Secure Interrupts

Each interrupt can be assigned to either the Secure or the Non-secure side using a configuration register accessible only to the Secure side. This implies that an interrupt must be able to cross a security domain depending on its interrupt priority and domain ownership.
Thanks to the banking of the System Control Block (SCB) –

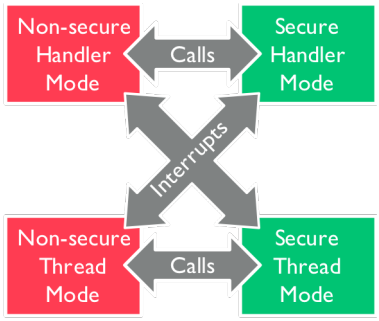

Fig 7. Security state transitions can be triggered by exceptions and function calls

one SCB for each the Secure and the Non-secure domain – there are two copies of the interrupt vector table offset register (SCB_VTOR).

Important system exceptions like the Systick, PendSV or the SVC are banked and can be therefore used independently on both sides. This enables an existing OS to be easily ported to the ARMv8-M architecture.

Depending on the requirements of the target application, the Secure side can configure the device to share the interrupt priority levels between the Secure and Non-secure domains. For safety critical applications Secure interrupt priorities can be shifted to obtain higher priority levels exclusive to the Secure side. The Secure side can then decide to trigger itself regularly for detecting corruption of the system or malware traces to allow controlled recovery of the system. The shifting of priority levels ensures that the Secure side cannot be overridden by malicious code on the Non-secure side.

To ensure CPU register data is not leaked across to the Non-secure side, CPU registers are cleared by the CPU after storing them to the Secure stack.

Thus whenever processing a Non-secure exception after switching from Secure state during a crypto-operation (or other confidential operations), this register clearing is performed automatically. Non-secure code therefore can't get access to Secure register contents.

Other side channels like cache attacks must be mitigated whenever needed by clearing caches on context switches. To avoid unnecessary cache-clearing, code in a security context can mark itself temporarily as side-channel-critical. The uVisor (a hypervisor-like software component) can observe such a hint for triggering cache-clearing and skip the clearing in all other cases.

A similar approach is possible for Non-secure access to FPU registers in interrupts: by enabling FPU security, all FPU registers are saved to the Secure memory space before switching to the Non-secure domain. The FPU registers are then cleared before handing over control to Non-secure interrupt code.

The Secure exception prioritization features outlined above are also highly applicable to safety critical applications (like motor controllers or safety-related trip wires for shutting down machine operation) where safety-critical code and data can be shielded from the rest of the system for preventing corruption either by accident or by an active attacker on the system.

I. Instant stack overflow detection: The stack limit register set

Traditional RTOS's verify each thread's stack pointer when switching out of a thread. Traditional checks involve overflow or underflow checking of stack pointer register (R12, SP) and verification of stack canaries. (Stack canaries are special numbers stored by the OS around the stack memory, and a sub-class of stack canaries can be used for stack overflow detection.) The OS then checks on every context switch, whether these numbers are still intact or have been (accidentally) overwritten.

The methods employed may not be sufficient however:

- Unfortunately stack overflows during execution within an active thread slice stay undetected for the remainder of the slice.
- Even advanced protection measures, for example using MPU-bands for creating "MPU-region-death-traps" around the currently active stack might fall short as the stack pointer might grow beyond the invalid region and can grow undetected into a valid, but critical memory (heap etc.).

The ARMv8-M architecture therefore introduced a new feature: The stack pointer register itself is continuously monitored in hardware – changes to the stack pointer register below a pre-determined address threshold are caught right at the spot where the stack pointer changes are calculated and stored back to the stack pointer register. Therefore, one doesn't have to rely on access to invalid stack memories any more for late detection of overflows.

The remaining responsibility of the RTOS regarding stack protection is to ensure that the stack limit register is always updated before switching into the next thread slice.

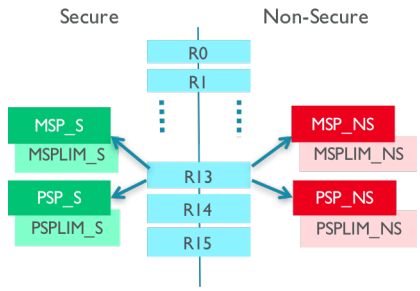

Fig 8. Each stack pointer in ARMv8-M Mainline processor has a corresponding stack limit register

For practical reasons the stack limit register is banked for all privilege states in ARMv8-M Mainline:

- **MSPLIM_S**: Secure privileged stack pointer limit
- **PSPLIM_S**: Secure unprivileged stack pointer limit
- **MSPLIM_NS**: Non-secure privileged stack pointer limit
- **PSPLIM_NS**: Non-secure unprivileged stack pointer limit

The ARM ARMv8-M Baseline profile only supports stack limit registers in Secure state. The Non-secure side needs to use the MPU to protect its stacks like the ARMv7-M version of the architecture.

IV. Pulling things together: Hardware accelerated security for realtime crticical systems

For enabling secure and safe real time critical operations, it is beneficial to use static MPU configurations on the Secure side.

During the secure-boot process a lightweight hypervisor-like security software (called ARM mbed uVisor) will initialize the SAU for partitioning the system into Secure and Non-secure domains. The Non-secure side can be partitioned into multiple domains – for multi-domain security systems with three or more security domains or at least two mutually distrustful security domains.

This design choice allows the OS to set up a simple and static MPU configuration for the Secure side:

- The core lightweight hypervisor functionality running in the Secure privileged domain which is protected by the Secure MPU instance against the non-privileged Secure domain. All required hypervisor memories are protected by the SAU and memory protection controllers against Non-secure access.
- OS-specific support functions run in the non-privileged Secure domain. These new support functions are called “Secure Device Services” (SDS). Thanks to hardware-accelerated secure gateway calls (SG calls) across Secure and the Non-secure domains are very efficient. This results in very low and predictable latency when using SDS APIs.

A. Secure Device Services

Secure device services (SDS) are designed to prevent execution blocking and use as little stack as needed to maintain at least one stack per Non-secure box domain. From the caller’s perspective, the Non-secure domain-instance can either block concurrent access from more than one thread in the same box at the same time or have one SDS stack per Non-secure thread allocated.

If more complex operations need to be triggered from an SDS, requests can be queued and sequentially processed in a shared thread on the Secure side. The amount of stack usage then becomes independent from the number of pending requests or domains. An SDS call must return immediately with a result code indicating a deferred operation in this case.

A good example for such APIs are long running crypto operations – the corresponding SDS API returns immediately and the Non-secure box receives a Non-secure callback once the operation terminates. Using SDS-functions, the result can then be examined across the security boundary and the result buffer can be processed on the Non-secure side.

Examples for other useful Secure Device Services are:

- Real-time critical interrupt handling – where just the time-critical part runs on the Secure side – everything else runs in Non-secure domains. Verification can be therefore focused on the secure-stub.
- Secure GPIO access (pin-wise access depending on the caller security domain, see register level access gateway notes below).
- Handling system-wide secure pool allocators: The Non-secure side can allocate coarse memory pages from the Secure side that are only accessible from their individual memory context.
- Register level / bit-level access gateways – enabling the access to individual bits of hardware registers depending on caller privileges.
- Inter-process communication APIs for enabling distrustful Non-secure boxes communicating with each other.
- DMA-APIs for enforcing caller-specific permissions across distrustful Non-secure boxes.
- Shared Crypto Accelerators / Crypto APIs – virtualize access from multiple domains to share accelerators while protecting against leakage of suspended operations across distrustful security domains.
- Random Entropy Pool Drivers – ensure that the whitening keys are invisible to caller domains and prevent starvation of entropy.
- Key provisioning and configuration storage to keep key material in shielded places in flash to ensure that each individual Non-secure box can decide whether it allows access to each individual storage items for other Non-secure boxes.
- Partitioning access to the flash memory for complex Non-secure clients like the firmware update or the mentioned configuration storage.

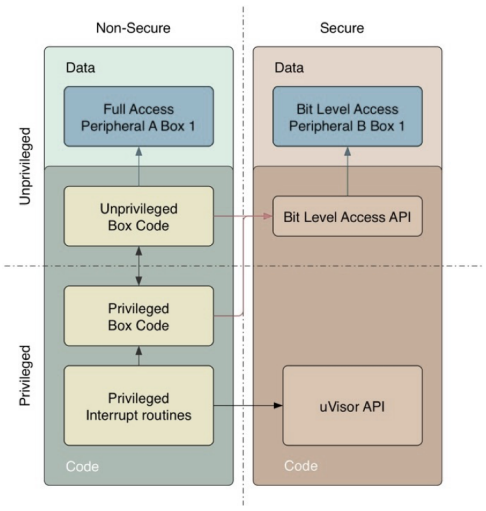

Fig 9. Security arrangement with uVisor and SDS

The Secure side always has access to all Non-secure memories (unless prevented by the privileged MPU configuration on the Secure side) – it can therefore efficiently copy data around and execute Non-secure caller requests on their behalf in case their context has the privilege to do so.

B. Paging for MMU-less microcontrollers

In constrained microcontroller systems one can expect a large temporary spike of memory consumption for seldom used processes like firmware update, device configuration or the provisioning of new encryption keys.

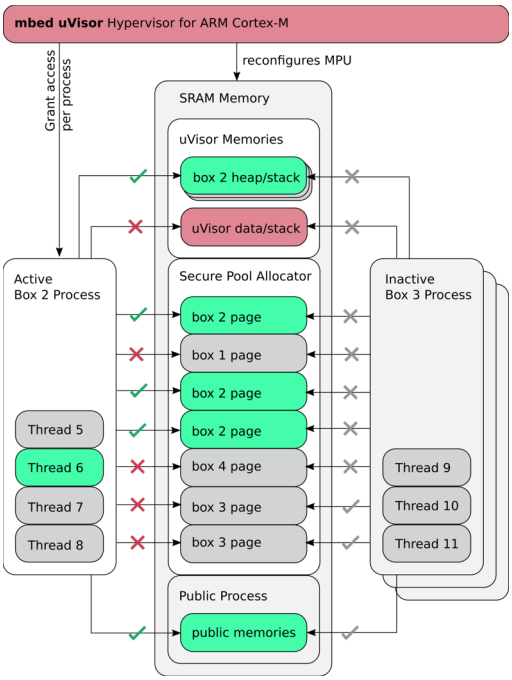

Fig 10. Memory access management in uVisor

To ensure that these memories can be used in other parts of the system during daily operations, the OS must dynamically move pages of memories around between Non-secure domains during runtime – the first tier of a two-tier memory allocator.

Each domain is then responsible for running a fine-grained second tier memory allocator in the allocated domain-specific 1st tier pages:

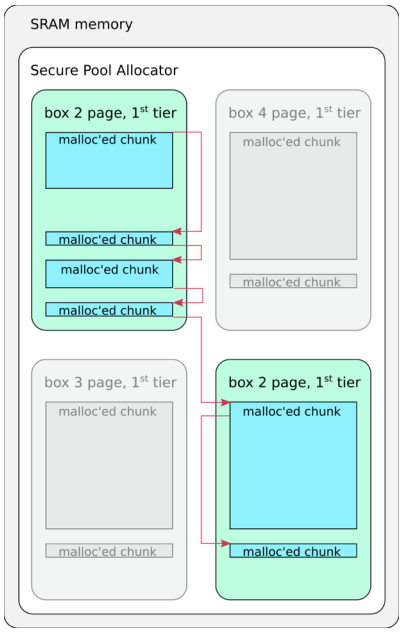

Fig 11. Two tier memory allocation

To ensure that Non-secure domains cannot perform denial-of-service attacks against each other, the maximum memory consumption must be capped for Non-secure boxes or large memory allocation limited to short periods – for example for performing memory-intense crypto operations.

To use the 1st tier pages effectively, application developers must ensure that the chosen 1st tier page size is larger than the largest possible 2nd tier memory allocation. For backward compatibility 2nd tier allocation are guaranteed to be continuous.

A good rule of thumb is to split the available program memory into 8 to 16 equal sized chunks to determine a suitable page size for the average IoT system. The memory allocation can be either handled by the SDS APIs or by the privileged hypervisor.

This approach is ideally suited for ARMv8-M block based memory protection controllers. By using the Non-secure/Secure mapping bitmask for protecting the SRAM, the Secure side can selectively map in just the pages accessible to the currently active Non-secure domain. All the pages owned by the other, currently inactive, Non-secure domains are inaccessible to the Non-secure side.

On every box switch the whole SRAM access bitmap is replaced by the one relevant for the newly activated box and depends on the pages owned by that security domain in the 1st tier allocator.

C. Register and Bit-level Access Protection

Using SDS services the Secure side can efficiently perform operations on behalf of Non-secure domains. Like access control to peripherals, each security domain needs to commit at compile time the registers (or bits inside a register) the security domain requires access to during runtime. This is particularly important as both the MPU and SAU have a minimum granularity of 32 bytes and are therefore not suitable for granting access to individual registers.

In case of uVisor, this commitment is transparently done by the pre-compiler through issuing of permissions into read-only memories during compile time. An application developer has convenient API functions to request and execute access to registers or bits by using convenient and 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)

Behind the scenes these calls are turned into a combination of the call and the corresponding permission plus the security domain of the call owner. In the case that code is compiled for a system that has no concept of these security functions, the functions are turned into direct register access or the corresponding bit-band access - resulting in no overhead.

During runtime the Secure domain can use the saved Non-secure program counter to determine the Non-secure source of that register level API call and seek back to a fixed relative position in code for inline-verification of permission-metadata for checking [2]whether the calling box is permitted to access a given register bit. If yes, it will perform the chosen operation on the Secure side on behalf of the Non-secure caller (if the caller's domain matches the quoted domain in the permission).

Potential malware can therefore neither re-use other domain's register level API calls, nor can it create new register calls during runtime. The addresses and access mask are part of the hardcoded and in-lined proof of permission. These proofs are pure metadata and designed to be discoverable inside a device firmware image without decompilation.

A firmware signing process is therefore able to check the requested permissions and ACLs for each security domain and can refuse signing the firmware image in the case it detects unwanted access requests. The target device can then trust the judgment of the server to install the image after verifying the firmware signature and blindly trust all quoted permissions.

One of the main applications of the register/bit-level access feature are clock-enable registers – it ensures that Non-secure domains are limited just to the few bits relevant to them so they won't be able to shut down clocks or reconfiguring clock dividers for the other security domains.

D. Whitelisting peripheral access in real-time

Using the peripheral protection controller, the Secure hypervisor can re-configure the Peripheral Protection Controller Bitmap on every box switch. For an instance a system with 128 peripheral blocks only needs to update four 32bit registers for completely reconfiguring access when switching between boxes:

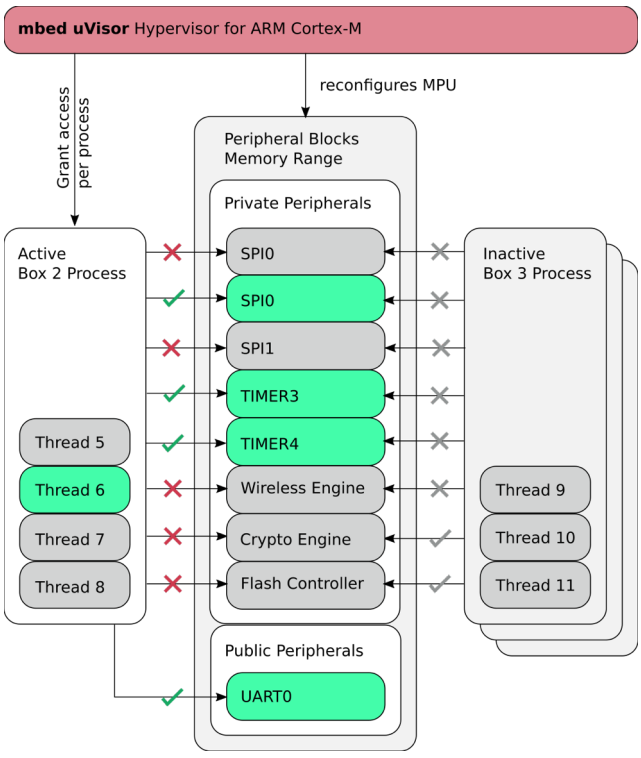

Fig 12. Peripheral access management with uVisor

Each domain requests access only for the peripherals it needs for its operation. In case of the ARM mbed uVisor, security domains are forced to commit to their resource requirements during compile time and the ACLs are only processed once

[2] See also the uVisor talk on Secure Register Level Access Gateways: https://goo.gl/Wlx6fj

during boot. During runtime Non-secure domains need to live with their ACLs and cannot expand on them dynamically.

This approach ensures that access-timings are predictable. As long as switching latency between boxes is predictable, we can guarantee hard real-time even in the presence of a local attacker running in other Non-secure boxes.

In the case of DMA access requests, the SDS API verifies if the source and target of an operation is consistent with the box permission before moving data from peripherals or memory pages to the Secure state. The requested source and target addresses are then updated to the secure mappings and the DMA request can be started. This ensures that DMA can continue while a security domain is inactive and while the next box becomes active.

While running the DMA, requests to the 1st page allocator need to block pages with a running DMA request from being returned to the pool to avoid cross-domain attacks.

Another option is to move the low-level DMA operations of a device to SDS and run the higher level functions on the Non-secure side – allowing unsecure domains to get temporary access to them (virtualizing/sharing).

I. Summary and Conclusion

By using newly introduced hardware features of the ARM TrustZone technology for Cortex-M33, modern microcontroller operarating systems can provide strong compartmentalization of mutually distrustful processes.

Particularly block level protection features like the block-based memory protection controller and the peripheral protection controller allow swapping out security settings in constant time while switching between processes with distinct access privileges – independent of their complexity.

Block level memory protection enables secure dynamic memory re-allocation between Secure processes during runtime to make the best use of the constrained resource SRAM with predictable realtime performance. Memories can change ownership during runtime between different security domains without affecting the security of the system.

The additional security state of the CPU - the Secure state - is not just ideally suited for providing low latency security services to processes. Among obvious use cases like inter-process-communication (IPC) and crypto acceleration, the SDS can also be used to run Secure interrupts at low latency and high safety requirements. Using the two hardware security domains – Secure and Non-secure - code can be partitioned not just to fulfil the security promise, but also for achieving functional safety for realtime-critical drivers or system functions.

All application code is executed on the Non-secure side, which is split into as many security domains as required in software: enabling mutual distrustful operation between these domains on the Non-secure side. The reach of a software bug is limited to its own security domain and the domains API – simplifying system design and code review.

The new ARMv8M hardware security features are designed for seamless integration with secure operating systems - creating high device resilience by enabling malware detection and remote recovery while maintaining device safety and timing integrity even under active attack[3].

The ARMv8M security model for Cortex-M33 micro-controllers nicely maps on models developers exercised in the past on high-end Cortex-A systems - reducing the attack surface for safety- and security-critical microcontroller applications dramatically.

I. Further Reading

[1] Milosch Meriac, "Practical real-time operating system security for the masses", ARM TechCon 2016 (https://goo.gl/Wlx6fj)

[2] ARM mbed uVisor project source code and documentation on github: https://github.com/ARMmbed/uvisor

[3] Joseph Yiu, ARM, "Software Development in ARMv8-M Architecture", Embedded World 2017

[4] Joseph Yiu, ARM, "The Next Steps in the Evolution of Embedded Processors for the Smart Connected Era", Embedded World 2016

[5] Joseph Yiu "ARM, "Whitepaper – ARMv8-M Architecture Technical Overview", ARM TechCon 2015.

[6] ARM, "ARMv8-M Architecture Reference Manual"

[7] ARM, "Memory Protection Unit for ARMv8-M based platforms"

[8] ARM, "RTOS design considerations for ARMv8-M based platforms"

[9] ARM, "Secure software guidelines for ARMv8-M based platforms"

[10] ARM, "ARMv8-M Security Extension: Requirements on Development Tools"

[3] See the ARM mbed uVisor talk on remote recovery from malware by securing flash memory access and a Secure watchdog: https://goo.gl/Wlx6fj