Retrofitting Zephyr Memory Protection

Wayne Ren, Sensor Software Engineer, Synopsys
Acknowledgements

These slides are based on Andrew Boie (Intel)’s “Retrofitting Zephyr Memory Protection”
Thank you, Andrew!
01 Background
02 Design and Implementation in Zephyr
03 ARC Specific Implementation
04 Future Work
Background
What is Zephyr

- **Zephyr**: a modular RTOS and a complete solution stack
  - RTOS for use on connected resource-constrained and embedded devices
  - Focused on safety, security, connections with Bluetooth support and a full native networking stack
  - Apache 2.0 license, hosted at Linux Foundation
  - Support diverse use cases and architectures: ARC, ARM, RISC-V, X86...
  - Web site: https://www.zephyrproject.org/

More Zephyr related events:

- “Introduction to the Zephyr Project” - Ryan Qian, NXP & Kate Stewart, The Linux Foundation, Tuesday, June 26 • 11:20 - 12:00
- “License Information Management” – Kate Stewart, The Linux Foundation, Tuesday, June 25 • 15:50 - 16:30
- IoT Meetup, Tuesday, June 26 • 18:00 - 21:00, 北京海淀区科学院南路2号, 融科资讯中心C座南楼1层 南极洲会议室
Why Required

- **Security concern of IoT devices**
  - More and more “things” are connected, traditionally offline->online
    - Secure communication
    - Trusted execution (secure boot)
    - Data protection
    - ....

- **Zephyr uses systematic approaches for security**
  - Static: high quality design, code review, tests, certification
  - Dynamic: secure communication, cryptography, memory protection

- **Memory protection**
  - One important approach for more secure, reliable and safe system
  - 1st step to implement security
## Memory Protection Hardware

### Memory Protection Unit (MPU)

- Popular in low-end device, ARM Cortex M4, ARC EM
- Fixed number of configurable regions, each with their own access policy
- No virtualization, physical memory addresses
- Typically have constraints on region specification, e.g. region sizes must be power of two, aligned to their size

### Memory Management Unit (MMU)

- Popular in Application processors, x86, ARM Cortex A series, ARC HS series
- Address space divided into equal sized pages (typically 4K).
- Configuration for caching and access, policy for each page set in page tables
- MPU-like behavior with identity page table, but Optional support for virtual memory
Memory Protection in Zephyr

• Zephyr had no means of preventing unwanted memory access before

• Joint effort with most contributions from Linaro (ARM), Synopsys (ARC), and Intel (x86), initial efforts targeting MPU-based systems

• Milestones
  – 1.9 release (7/2017): MMU/MPU enabled, stack overflow protection on ARM/x86
  – 1.10 release (11/2017): user mode support on x86 MMU
  – 1.11 release (3/2018): user mode support on ARC/ARM MPU
  – 1.12 release (6/2018): more tests, refinement

• Future work
  – Additional CPU architecture support
  – Flesh out APIs and iterative refinement
  – Support of TEE (Trusted Execution Environment), e.g., secure and non-secure world (1.13 or later)
Use Cases

• Protect against unintentional programming errors
  – Stack overflows
  – Writing to bad memory
  – Data corruption

• Sandbox complex data parsers and interpreters
  – Network stacks/protocols
  – File systems
  – Reduce likelihood of third-party data compromising the system

• Support the notion of multiple logical isolated applications
Comparison with Other RTOSes

- **FreeRTOS-MPU**
  - not default configuration of FreeRTOS
  - Unprivileged "User" threads with configurable memory access, system calls for privileged operations
  - Not well maintained, often doesn't compile

- **NuttX Protected Build**
  - Supports ARM MPU and MMU (with identity page table)
  - Unprivileged threads similar to FreeRTOS-MPU
  - Separately loaded applications
  - Many features proposed but still WIP

- **ThreadX Modules**
  - MPU or MMU Virtualized address spaces for separately loaded modules with thread-level memory protection features
  - Support for lots of different CPUs
  - Not free. Royalty-free license with significant upfront cost, modules feature costs extra

- **Zephyr**
  - Thread-level protection
  - Support for lots of different CPUs
  - Same kernel & driver APIs for kernel and user mode threads
  - Free, Apache 2.0 License
  - More features in the future
Design and Implementation in Zephyr
Threat Model

• User thread
  – Untrusted
  – Isolated from the kernel and each other

• Kernel thread and kernel
  – Trusted, privilege to access all

• A flawed or malicious user thread cannot:
  – Leak or modify private data of another thread unless specifically granted permission
  – Interfere with or control another thread except through designed thread communication APIs (pipes, semaphores, etc.)
User Mode

• Control access to kernel objects and device drivers
  – Per-object and per-thread basis

• Maintain compatibility with existing Zephyr APIs

• Implement system calls for privilege elevation

• Arch-specific code to enter user mode

• Validate system call parameters including kernel object pointers

• Do not require changes to individual drivers

• Manage user mode access to memory
High-Level Policy

• User threads are by default granted only
  – Read/write access to their own stack memory and application memory
  – Read-only/execute access to program text and ROM
  – Memory domain APIs to configure access to additional regions with child thread inheritance

• User threads cannot use device drivers or kernel objects without being granted permission
  – Permission granted by other threads with sufficient permission or inherited

• System call API parameters are rigorously checked

• User mode stack overflows are safely caught
Permission Model

• Each kernel object has a bitfield indicating what user threads have access to it.

• Kernel threads can grant object access to any user thread.

• User threads may grant object access to another user thread if the calling thread has permissions on both the object and the target user thread.

• Newly created user threads may optionally inherit object permissions of the parent thread.
Kernel Object

• Three main types of kernel-private data structures
  – Kernel API data structures - `k_thread, k_sem, k_mutex, k_pipe`, etc.
  – All device driver instances
  – All thread stacks, instead of individual structs, these are arrays of a special typedef to character data

• To preserve Zephyr API compatibility, all are referenced by memory address
  – Act as a handle for user threads, object memory not accessible
  – Need a system for validating object pointers passed to system calls
Kernel Object Permissions

• Kernel threads can access all objects
  – Permissions still tracked, because
    • Thread drops to user mode
    • Creates child user threads with object permission inheritance enabled
  – May designate some objects as "public" and usable by all threads

• User threads
  – If created with permission inheritance, gain access to all parent thread's permissions except parent thread object
  – \texttt{k_object_access_grant()} calls must have permission on both the target thread and the object being granted permission to
Kernel Object: New Type

• Creating new kernel object types is easy!
  – Add the name of the associated data structure to the build
    • Struct name itself for new kernel APIs
    • API struct name for new device driver subsystem types
  – Small modifications to some lists in two C files
    • Could eventually be automated

• Recognizing instances of kernel objects and providing a validation function for them is all handled automatically at build time
Kernel Object: Constraints

• **Must be declared as a top-level global**
  – Needs to appear in the kernel's ELF symbol table
  – OK to declare with static scope
  – May be embedded as members of larger data structures

• **Memory for an object must be exclusive to that object**
  – Can't be part of a union data type

• **Must be in the kernel data section**

• **Objects that do not meet these constraints will not be accessible from user mode**

• **Future work: support runtime allocation use-cases from slabs/kernel heap**
Kernel Object: Definition

- **perms**: permission bitfield indicating thread permissions for that object

- **type**: object type information
  enum, `K_OBJ_THREAD`, `K_OBJ_UART_DRIVER`, ...

- **flags**: initialization state, public/private, others as needed

- **data**: extra data in some cases
  - Stack object size
  - Build-time assigned thread ID

```c
struct _k_object {
    char *name;
    u8_t perms[CONFIG_MAX_THREAD_BYTES];
    u8_t type;
    u8_t flags;
    u32_t data;
}

extern struct _k_object * _k_object_find(void *obj);
```
How to Get Kernel Object Info?

• Problem: need to find all the kernel objects
  – Map object memory addresses to instantiations of struct _k_object containing metadata
  – Validate kernel object pointers passed in from user thread

• Solution:
  – gen_kobject_list.py
    • Use pyelftools to unpack ELF binary and fetches all the DWARF debug information, and does object identification
  – gperf
    • a GNU tool for creating perfect hash tables
    • Generate the hash table of kernel objects for efficiency
Kernel Object: Flow

zephyr pre-built.elf (1st build) → gen_kobject_list.py → kobject_hash.gperf → gperf tool → kobject_hash_prep_process.c

zephyr.elf → preprocess tool → kobject_hash.c

2nd build
System Call

• Typical OS mechanism for allowing user threads to perform operations they can’t do

• On all arches, API ID and parameters are marshaled into registers and a software interrupt/exception is triggered
  – Up to six registers used; additional args passed in via struct and stack

• Common landing site for system calls on kernel side
  – Validate API ID, execute the handler function
  – Clean general purpose registers on exit to prevent private data leakage

• Use build-time logic to make adding new system calls as painless as possible
System Call: Components

• Very easy for developers to define

• Created by developer for each system call:
  – System call header prototype `__syscall void k_sleep(s32_t duration)`
  – Handler function for argument validation `Z_SYSCALL_HANDLER(k_sleep, duration)`
    • Verify caller permissions on provided memory buffers or data passed via pointer
    • Copy any parameter data passed in via pointer to local memory
    • Verify object pointers, permission, initialization state
    • Verify parameter values which are otherwise left to assertions or simply un-checked
  – Implementation function `void __impl_k_sleep(s32_t duration)`
    • Kernel object API code under `kernel/
    • Driver subsystem API functions defined at the subsystem level

• Auto-generated for each system call:
  – System call ID enumerated type
  – Handler function prototypes
  – `__k_syscall_table` entry mapping ID to handler function
  – `__weak` handler function for system calls excluded from kernel config
  – System call invocation function
System Call: Flow

API Call

User Mode?

Y: Marshal args, Trigger SW IRQ

N: k_oops()

Valid call ID?

Y: Lookup handler in dispatch table

N: k_oops()

Handler Checks

Implementation Function

Y: Marshal Return Value, exit IRQ

N: Return to Caller
System Call: Build-Time Magic

• Limited parsing of kernel header files, looking for function prototypes prefixed with "__syscall".

• Parsing limited to determining return value and argument types to generate additional functions.
  – Some minor limitations in parsing with array/function pointer argument types which can be easily worked around.

• Generated headers contains implementation of API as an inline function - invokes system call trap or direct call to implementation as appropriate.

• Some generated C code for default handler and dispatch table entry.
Memory Domain

- User threads by default can't look at any RAM except their own stacks
- Need a flexible way to designate additional memory areas that a thread has access to
- Limited number of total MPU regions needs to be taken into consideration
- Grant access to top-level data or BSS section globals defined and used by the thread, or application data that needs to be shared between threads
- Memory Domain APIs exist to handle re-programming the MPU for the incoming thread's memory access policy on context switch
Memory Domain: Implementation

- Memory domain APIs are kernel-access only, no system calls
- Implemented as an object `struct k_mem_domain`
  - Contains some number of memory partitions (`struct k_mem_partition`)
    - Up to the maximum number of regions supported by MPU hardware, no limit for MMU
    - Each partition is a starting address, size, and access policy
    - Hardware dictates alignment and size constraints
  - APIs to add/remove partitions to an initialized memory domain object
- Any thread may be added/removed to a particular memory domain to implement an access policy for that thread
- MPU region registers or MMU page tables updated upon context switch to activate policy for incoming thread
- Special Case: Application Memory
  - Shared to all threads, CONFIG_APPLICATION_MEMORY in Kconfig
  - All top level globals in non-kernel object files (libs, application code) placed in user read/writable section by linker and access policy configured in MMU/MPU at boot
- Facilities for grouping data by the linker (WIP)
ARC Specific Implementation
Introduction of ARC

**EM Family**
- Optimized for **ultra low power** IoT
- 3-stage pipeline w/ high efficiency DSP
- Power as low as 3uW/ MHz
- Area as small as 0.01mm² in 28HPM
- Well supported in Zephyr

**HS Family**
- **Highest performance** ARC cores to date
- High speed 10- stage pipeline
- SMP Linux support
- Single, dual, quad core configurations
- Support in Zephyr: in progress

ARC EM Starter Kit
ARC HS Development Kit
ARC Support in Zephyr

- **ARC in Zephyr**
  - `<zephyr root>/arch/arc`
  - Board:
    - ARC EM Starter Kit
    - Arduino 101 sensor subsystem
    - Quark_se based board

- **Processor:**
  - User/kernel mode
  - Stack overflow check
  - DSP, fast IRQ
  - SecureShield

- **MPU:**
  - `em_starterkit_em7d_v22` (emsk 2.2 firmware)
    - MPUv2, power of 2, >2048 bytes
  - `em_starterkit_em7d` (emsk 2.3 firmware)
    - MPUv3, 32 bytes aligned, no overlapping

- **FPGA-based board**
- **128 MB DDR3 RAM + PMOD interfaces**
- **Fmax 20-25 MHz**
- **Supports all ARC EM Processors:**
  - em7d, em9d, em11d
- **Usage:** Early prototyping
Layered Approach

L4: Virtual Memory
- Zephyr "processes" in their own VM
- Demanding Pages
- Implementation in progress

L3: User thread
- User threads running in un-privileged mode
- System calls
- Stack & memory isolation
- Thread-level kernel object/driver permissions and memory policy

L2: Stack Overflow detection
- In kernel mode
- Detection of Stack overflow errors

L1: Boot-time
- Config MMU/MPU
- No-execute for non-text
- NULL pointer dereferences
- Exceptions for nonsense address
Memory Map

- Correct memory map is the foundation of memory protection

- Static MPU entries
  - Boot time memory configuration
  - ROM: 1 MPU entry, RO+EXE
  - RAM(kernel): 1 MPU entry, kernel RW
  - RAM(application memory): 1 mpu entry, User RW
  - Peripheral area: 1 mpu entry, kernel RW

- Dynamic MPU entries
  - Thread stack
  - Memory domain
Thread Stack

- **Kernel thread**
  - Merge the privilege stack into thread stack for more stack space

- **User thread**
  - 1 MPU entry for user stack

- **Stack overflow protection**
  - STACK_CHECKING(optional)
    - both for user stack and privilege stack
  - MPU based: stack guard page
Future work
Runtime Kernel Object Allocation

• Not always possible to define all kernel objects used at build time
  – Build-time constraints prevent allocation of kernel objects in separately loaded application code at all

• Two approaches, both under implementation
  – Build-time defined slab pools of kernel objects
    • Pools are build-defined arrays of various objects and validated as normal
  – Kernel-side heap allocation of kernel objects
    • Supplemental runtime hash table for tracking validity of new objects
    • User mode no direct access to this heap!
Kernel API Improvements

- Not all kernel APIs exposed as system calls
  - Many combine user and private kernel data in ways which could be attacked
  - `k_mem_pool`, `k_poll`, `k_queue`

- Need some better heap features
  - `k_mem_pool` APIs were designed to be ISR-safe and not usable from user mode
  - newlib heap is just a singleton for entire address space since no VM

- Need a `k_mem_pool` equivalent that runs entirely in user mode, using memory domains to control access

- User-mode work queues
  - `k_work_q` threads currently run in kernel mode using `k_queue` for data buffering
Memory Organization Features

• "Application Memory" feature was useful for getting test cases up but does not work well for real-world uses

• Need a solution which handles both setting up 1..N memory areas for applications
  – Configure memory domains
  – Tie into linker scripts to ensure the data gets where it needs to be
  – Handle alignment constraints

• No design for this yet, under discussion
TEE & Secure Mode Support

- **Zephyr**
  - High-level design on discussion
  - Arch specific work starts

- **ARM**
  - Hardware: Trustzone-M, Cortex M23/33
  - Software: PR #6766, #6748, #4985 ...

- **ARC**
  - Hardware: SecureShield, em7d of emsk 2.3
  - Software: WIP
Call To Action

- Want to learn more? Have some ideas? Get started here:
  - https://www.zephyrproject.org/

- Check out codebase on GitHub:
  - https://github.com/zephyrproject-rtos/zephyr

- Join our mailing list or hang out in our IRC channel (WeChat, etc)

- Join weekly on-line meetings, TSC meeting, secure, network, ....
LINUX CON

CONTAINER CON

CLOUD OPEN

CHINA 中国

THINK OPEN

开放性思维