EC: Embedded Systems Compartmentalization via Intra-Kernel Isolation

Arslan Khan, Dongyan Xu, Dave (Jing) Tian
{khan253, dxu, daveti}@purdue.edu
Purdue University

2023 IEEE Symposium on Security and Privacy (SP) | 978-1-6654-9336-9/23/$31.00 ©2023 IEEE | DOI: 10.1109/SP46215.2023.10179285

***Abstract*—Embedded systems comprise of low-power microcontrollers and constitute computing systems from IoT nodes to supercomputers. Unfortunately, due to the low power constraint, the security of these systems is often overlooked, leaving a huge attack surface. For instance, an attacker compromising a user task can access any kernel data structure. Existing work has applied compartmentalization to reduce the attack surface, but these systems either incur a high runtime overhead or require major modifications to existing firmware. In this paper, we present Embedded Compartmentalizer (EC), a comprehensive and automatic compartmentalization toolchain for Real-Time Operating Systems (RTOSs) and baremetal firmware. EC provides the Embedded Compartmentalizer Compiler (ECC) to automatically partition firmware into different compartments and enforces memory protection among them using the Embedded Compartmentalizer Kernel (ECK), a formally verified microkernel implementing a novel architecture for compartmentalizing firmware using intra-kernel isolation. Our evaluation shows that EC is 1.2x faster than state-of-the-art systems and can achieve up to 96.2% ROP gadget reduction in firmwares. EC provides a low-cost, practical, and effective compartmentalization solution for embedded systems with memory protection and debug hardware extension.**

I. Introduction

Embedded systems are microcontroller-based systems designed to perform a specific task. They are pervasive and are used extensively in high-criticality systems such as Unmanned Vehicles (UV), health monitoring, and even general-purpose computing systems such as mobile phones and personal computers. As these microcontrollers have to work continuously on a constrained power budget, low computation power becomes an inherent feature of embedded systems. Unfortunately, firmware developers usually overlook the security implications in pursuit of performance. For instance, embedded systems are often programmed in a flat address space with no security mechanisms in place to minimize the runtime overhead. As a result, embedded systems expose large attack surfaces, making them vulnerable to various attacks [1], [2], [23], [31], [33], [34], including Denial-of-Service [58], control hijacking [37], and arbitrary code execution [7], [8]. Existing work [41], [50] has shown that it is possible to overtake the control of an Unmanned Aerial Vehicle (UAV) using a compromised WiFi System on Chip (SoC).

To tackle embedded systems security, existing work has used *Compartmentalization* [48], [51] to reduce the attack surface. Compartmentalization divides the firmware into multiple components with dedicated resources called compartments. Each compartment is only able to access resources within its protection domain to achieve fine-grained isolation at the same time. Minion [52] compartmentalizes the firmware based on different threads in the system. ACES [28] compartmentalizes baremetal systems based on developer-guided policies. M2MON [50] can be used to compartmentalize firmware at a device level. All of these systems require a reference monitor to enforce runtime memory protections in the privileged execution mode. However, the firmware itself runs in the privileged execution mode. Hence to enforce memory protections, users have to port the firmware to the unprivileged execution mode leading to major changes in the firmware. Due to this reason, each compartment switch raises an exception leading to a substantial runtime overhead. Moreover, each solution has limitations in terms of identifying different compartments in the system. For instance, Minion can only compartmentalize based on threads in the system, whereas ACES provides a programmable interface but does not work on RTOSs.

In this paper, we present Embedded Compartmentalizer (EC), an automatic compartmentalization toolchain for RTOSs and baremetal systems. EC provides a custom compiler, Embedded Compartmentalizer Compiler (ECC), to automatically compartmentalize firmware into different compartments, and generate EC-compatible binaries based on programmable compartmentalization policies. EC relies on Embedded Compartmentalizer Kernel (ECK), a formally verified microkernel, to enforce runtime memory isolation among different compartments using intra-kernel isolation.

ECC uses various program analyses to partition a firmware and extends the C type system for programmers to guide the compartment identification. ECC translates the firmware source code into an Intermediate Representation (IR) with information regarding the application and the RTOS, such as threads, memory usage, etc. Next, ECC partitions the firmware based on different user-defined policies. Finally, ECC enables transparent interactions between different compartments by instrumenting the firmware and generates metadata about the identified compartments and their resources for ECK.

ECK is a formally-verified microkernel enforcing memory protection using a novel operating system architecture that splits the hardware privileged mode into a constrained and unconstrained privileged mode. Both constrained and unconstrained privilege modes can carry out all privileged operations, except that: *The constrained privileged execution mode cannot issue memory configuration updates to the hardware*. ECK uses the constrained mode for hosting firmwares with minimal changes, while it runs inside the unconstrained execution mode and enforces the compartments generated by ECC. To realize this

DOI 10.1109/SP46215.2023.00163

intra-kernel isolation, ECK creates the constrained privileged execution mode within the existing hardware privileged mode using hardware memory protection and debug extensions.

We apply EC to baremetal and RTOS firmwares, such as FreeRTOS. Our evaluation shows that EC can achieve up to 96% ROP gadget reduction and 100% removal of functional gadget set expressivity. Furthermore, the ECK enforcement mechanism is 1.2x faster than the state-of-the-art compartmentalization solutions. The main contributions of this paper include:

- We propose ECC, an LLVM-based toolchain to automatically compartmentalize a firmware supporting programmable compartmentalization policies. ECC instruments each compartment to correctly interoperate with other compartments and allows programmers to configure the partitions by providing a programmable interface and generating statistics for estimating the runtime overhead cost of the current compartmentalization scheme.
- We design ECK, a formally verified microkernel that uses a novel architecture for intra-kernel isolation to enforce runtime memory protection without hardware context switching. ECK leverages the memory configuration metadata generated by ECC to enforce compartmentalization at runtime. To the best of our knowledge, ECK is the first ever system to bring intra-kernel isolation to embedded systems.
- We apply ECC and ECK to baremetal and RTOS firmwares to find the efficacy of EC protection. Our evaluation shows that EC can protect commodity baremetal applications and FreeRTOSs, achieving up to 96.2% reduction in ROP gadgets and up to 100% removal of function gadget set expressivity.

To further research on this topic, we have released the source code for EC.

II. Background

ARM Micrcontoller Profile. ARM Microcontroller profile is the specification of ARM Reduced Instruction Set Computer (RISC) that targets low latency and high determinism for embedded systems [89] implemented by Cortex-M processor series. Cortex-M mainly targets low-power systems and is the most prevalent processor for embedded systems supporting hardware privilege separation by providing a privileged and unprivileged mode. It implements the Systick timer for timekeeping and uses the Nested Vectored Interrupt Controller (NVIC) for interrupting the core. Cortex-M implements hardware context switching, i.e., upon a context switch request the context of the running execution state is pushed on the stack by the processor in a well-defined manner.

Memory Protection Unit (MPU). MPU is the hardware memory protection peripheral used by most Cortex-M processors. It enforces access permissions on different memory regions. However, unlike the conventional Memory Management Unit (MMU), MPU does not provide any address translation but enforces memory protection on flat address spaces. MPU can have multiple memory protection ranges, varying from 8 to 16 depending on the processor. MPU also has limitations on the memory region size and alignment. For instance, the memory region offset should be a multiple of its size; the size of the regions must be a power of two and should be greater than 32 bytes. MPU also allows memory regions larger than 256 bytes to be divided into equally sized sub-regions.

Data Watchpoint and Trace Unit (DWT). DWT provides watchpoints, data tracing, and system profiling for ARM processors. DWT can be used for both invasive and non-invasive debugging. DWT consists of multiple comparators and the exact count depends on the processor. Comparators can be configured as hardware watchpoints, tracing triggers, program counter samplers, or data address samplers. While DWT events are mainly used with an external debugger, Cortex-M can also consume DWT events internally using the Debug Monitor exception.

III. Security Model

Trust Model/Assumptions. EC trusts the compiler for correct instrumentation for firmware and compartmentalization metadata generation. EC assumes the boot-time integrity of the firmware and a flat address space. EC also assumes the presence of an MPU or a similar device to enforce hardware-based memory protection. Furthermore, EC assumes the presence of a DWT or a similar peripheral that can monitor memory address ranges for accesses.

Threat Model. EC targets attacks that compromise the embedded system by exploiting vulnerabilities within the firmware. The attacker tries to issue rogue memory and peripheral accesses to carry out malicious activities in the system, such as crashing the system or corrupting system data, leading to attacks such as privilege escalation and arbitrary code execution.

Out-of-Scope. EC does not consider side-channel attacks or physical attacks for this work. Furthermore, in the absence of an Input-Output Memory Management Unit (IOMMU), EC does not consider DMA-based attacks and assumes the correct configuration of the DMA peripheral by the compartment responsible for operating DMA.

IV. Motivation

Compartmentalization retrofits existing firmware to enforce privilege separation by identifying different compartments in the system and allocating exclusive resources to each compartment. Compartmentalization systems are characterized by two features: 1) How are different compartments in the system identified? (*Compartment Identification*), and 2) How does the system enforce the runtime restriction of each compartment?(*Compartment Enforcement*).

Table I shows the comparison of existing compartmentalization solutions for embedded systems. In terms of compartment identification, MINION [52] automatically compartmentalizes different threads in the firmware. ACES [28] implements a programmable interface for identifying compartments in a firmware. M2MON [50] builds upon Minion, to provide a device-based reference monitor for the firmware. These systems are tailored to specific facets of the compartment identification problem. For instance, MINION can work with RTOS but only provide a fixed compartmentalization scheme. On the other

System	Cross Compart. Overhead	TCB/ SLOC	Compartment Identification	Compartment Enforcement	Target	Porting Effort	Comparted Heap	Comparted Stack	Object Sharing	Verified
MINION	24.36%	1K	Thread-Based	MPU + SVC	RTOS	Mode Switch	No	No	None	No
ACES	24.36%	N/A	Programmable	MPU + SVC	Baremetal	Mode Switch	No	No	Dynamic Profiling	No
M2MON	21.12%	3.4K	Device-Based	MPU + SVC + SFI	RTOS	Mode Switch	No	No	None	No
EC	**11.13%**	**160**	**Programmable Tunable**	**Intra-kernel Isolation**	**Baremetal RTOS**	**Minimal***	**Yes**	**Yes**	**Type Extensions**	**Partial+**

TABLE I: Comparison of different compartmentalization solutions based on the *Compartment Identification* and *Compartment Enforcement* scheme. (*Cross compartment interfaces might require annotation to work with EC memory protections.) (+ Only ECK is verified.)

System	*Architecture*	*Enabling Features*
Hypersafe [71]	x64	WP Bit+CFI
Nested Kernel [32]	x64	WP Bit + Binary Analysis
SKEE [20]	ARMv7/8A	TTBCR + Binary Analysis
EC	**ARMv7M**	**MPU+DWT+Binary Analysis**

TABLE II: Comparison of different intra-kernel isolation solutions based on the key enforcement technology and architecture.

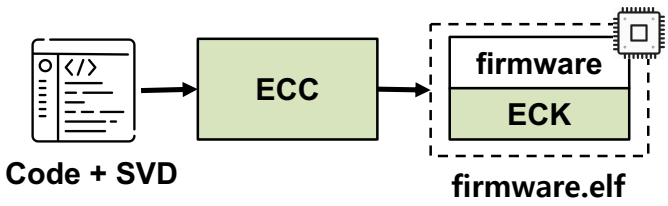

Fig. 1: EC workflow: ECC compartmentalizes on firmware which can be optionally annotated with ECC annotations. ECK enforces the memory protections at runtime.

hand, ACES can implement different compartmentalization policies, but it lacks support for RTOS firmware, dynamic memory, and controlling cross-compartment interactions. A comprehensive compartmentalization solution for baremetal and RTOS systems is still missing. To this end, EC should be able to partition the firmware based on user-defined policies for both baremetal and RTOS firmwares. Furthermore, EC should support dynamic memory allocation and allow flexible mechanisms for resource sharing among different compartments.

For compartment enforcement, all of the aforementioned systems use a security monitor to enforce memory protection. The security monitor requires a separate privilege mode for its execution. As the privilege mode is already in use by the firmware, this design results in an extensive porting effort to move legacy firmware to unprivileged mode and an expensive context switch on each call to the security monitor. Ideally, we want both the security monitor and the existing firmware co-existing in the same privileged mode. Existing work for commodity general-purpose OS has shown that it is possible to enforce hardware memory protection without requiring a higher privilege mode, using intra-kernel isolation. Hypersafe [71] uses control flow integrity and the Write-Protection (WP) bit to achieve intra-kernel isolation between the security monitor and the hypervisor. The WP bit is an x64 paging feature that disallows write access to memory regions. Similarly, Nested Kernel [32] uses the WP bit to disable modifications to page tables and uses binary analysis to remove any privileged instructions from the normal kernel. SKEE [20] achieves intra-kernel isolation for ARMv7-A and ARMv8-A by removing any instructions that manipulate the paging base address register (TTBR0/TTBR1) and explicitly making all paging data structures read-only.

In general, intra-kernel isolation systems work by disabling access to paging-related data structures and filtering of memory protection-related instructions that can update the configuration. However, ARMv7-M enforces memory protection using a Memory Protection Unit (MPU) which provides a very limited memory protection feature set[1] and does not prevent access to memory protection related configuration registers. Furthermore, modifications to MPU's configuration are not done using special instructions. Instead, the MPU is memory-mapped and configured using normal memory access. Therefore, filtering out special instructions to protect MPU configuration is also impossible in embedded systems. Hence, applying the same strategy to embedded systems would not work. Table II shows the comparison of existing intra-kernel isolation solutions. To implement the monitor beside the existing firmware within the same privileged mode, EC should tackle these challenges to achieve intra-kernel isolation in embedded system firmware.

V. Overview

EC is an automatic compartmentalization framework for embedded systems. For compartment identification, EC provides a custom toolchain, EC Compiler (ECC), that automatically compartmentalizes a firmware into several self-contained partitions with exclusive resources, called *compartments*, whereas for compartment enforcement EC uses a formally-verified microkernel, EC Kernel (ECK), to enforce the runtime memory protections for each compartment.

ECC accommodates existing compartment identification techniques by providing a programmable interface for developers to identify compartments in RTOS and baremetal firmware. Users can program different *policies* to modify the compartment identification behavior for ECC. Meanwhile, ECC provides a series of compartment policies, including thread-based policy [52], file-based policy [28], etc, covering all existing solutions. ECC conservatively identifies compartments

[1]ARMv7-M memory protection is documented in only 22 pages, whereas ARMv7-A documentation consists of 496 pages.

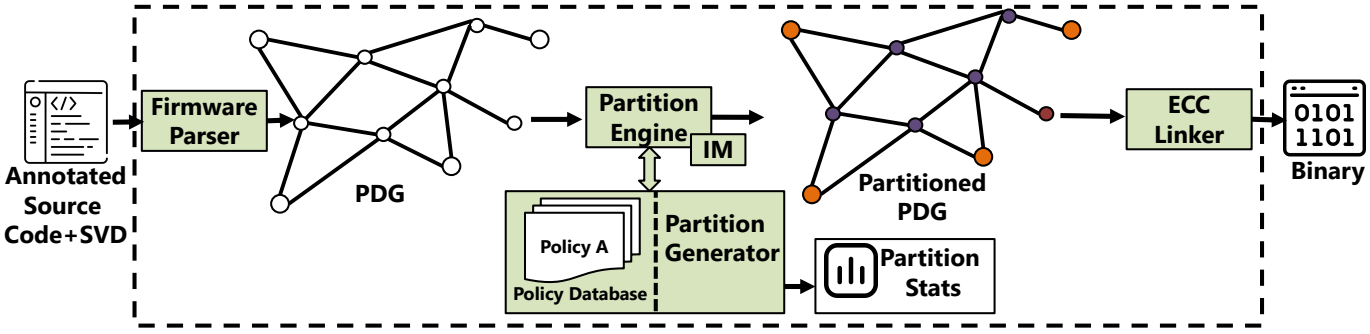

Fig. 2: ECC pipeline: Developers may annotate the source code to guide the ECC partition engine. The firmware parser parses the firmware and sends PDG with metadata to the partition engine. The partition engine invokes the partition generator to partition the PDG and instruments the firmware with appropriate inter-compartment calls using IM. Lastly, the linker generates the binary with associated metadata for the partition.

and allows users to guide the compartment identification analyses and cross-compartment interactions by providing program directives and adding type qualifiers to the C language. Lastly, ECC generates heuristics about the runtime overhead of the selected compartmentalization scheme.

ECK uses a novel operating system architecture for embedded systems to tackle the challenges of existing compartment enforcement techniques. ECK uses intra-kernel isolation to run the firmware besides ECK in the same privilege execution mode, instead of moving the firmware from the privileged mode to the unprivileged mode, minimizing the changes required to existing firmwares. Furthermore, ECK can switch compartments without a context switch resulting in minimal runtime overhead.

Figure 1 shows the workflow of EC. ECC takes in firmware and compartmentalizes the firmware into various compartments based on the selected policy. The compartmentalized firmware is linked with ECK, which enforces the compartments using the metadata provided by ECC.

VI. Design

A. Compartment Identification

In EC, ECC identifies compartments using various program analyses to compartmentalize the firmware. It takes firmware as input and generates various self-contained *compartments*. Each compartment can only access the resources assigned to it by ECC. For interaction between compartments, each compartment uses *Cross-Compartment Calls (xcalls)* to invoke functions from other compartments.

Figure 2 shows the pipeline of ECC. ECC first extends the C type system to enable developers to guide the static analysis process (*Language Extensions*). The *Firmware Parser* uses this information to generate a Program Dependency Graph (PDG) with associated metadata about device usage and OS-specific information for the *Partition Engine*, which partitions the PDG using the *Partition Generator*. The partition generator instruments each compartment for interactions between different compartments and generates the metadata of the compartments for ECK. The ECC *Linker* links the firmware with ECK and lays out each compartment in the memory such that it can be protected by ECK.

Language Extensions: ECC works with C code and supports optional code annotation with complementary type information to guide itself for better compartmentalization. The annotations inform ECC about how to properly pass parameters for cross-compartment calls, capabilities for function, etc. The complete list of type extensions is covered in Section A.

Firmware Parser: The firmware parser compiles C code, understands the ECC type extensions, and extracts the following information from firmware.

Code Annotations: The firmware parser goes through the firmware to generate the PDG. If it finds any annotation for a function, it attributes the PDG node with the information.

Device Usage: Firmware accesses peripherals using Memory Mapped I/O (MMIO). The firmware parser finds MMIO pointers by finding all pointers that are initialized with a hard-coded address. The firmware parser uses the System View Description (SVD) [17] files to associate MMIO pointers to different devices available on the platform and associates devices with the PDG nodes using the device.

RTOS Specific Information: The firmware parser analyses the firmware and generates information related to the RTOS used by the firmware, such as threads, components, configuration, etc. These static analysis passes are specific to each RTOS and they emit the information as metadata appended to the PDG.

Partition Engine: The partition engine works on the PDG with the associated metadata generated by the firmware parser. In general, no single solution fits all problems of firmware partitioning. To this end, we offload the partitioning to the *Partition Generator* that emits a partitioned PDG and *partitionin stats* based on the selected partition policy. The partition engine instruments the partitioned PDG with xcalls using the *Instrumentation Module*.

Partition Generator: The partition generator is a programmable module that can partition an input PDG based on the policy selected by the user. Users can modify the policies or implement new policies to create new partition schemes. After invoking a policy, the partition generator runs the verification module to verify that the generated partitions have exclusive access to resources. By default, we provide a number of policies:

- *Thread [52]:* Each thread in the firmware is assigned a separate compartment.

- *Component:* An RTOS usually comprises of different components. This policy assigns a separate compartment to each component.
- *File [28]:* File policy assigns all the objects in a file in a separate compartment.
- *Device:* Device policy assigns each device to a separate compartment.
- *Color:* Color-based policy tries to evenly create compartments using a greedy clustering algorithm.

During partitioning, ECC also caters to shared variables. Existing work [28] relies on implicit data sharing found by runtime profiling and uses an allow-list to check the access in the runtime exception triggered upon cross-compartment access. Therefore, each cross-compartment access incurs an overhead. Furthermore, this approach has the inherent limitations of dynamic analysis and does not scale well for RTOS, as there is extensive implicit sharing of data between the kernel and threads. For instance, the thread stacks are allocated dynamically by the kernel escaping the static analyses for partitioning the firmware. Therefore, local variable access within a thread stack would generate a cross-compartment access, as the dynamically created stack and the thread will be present in different compartments, resulting in high overhead. On the contrary, ECC does not allow implicit sharing of data by default and mandates explicit sharing of data by extending the C type system and providing runtime services to implement various alternatives.

- *Type Extensions:* Users can ease and secure explicitly shared variables using the `shared` type qualifier. The partition engine merges compartments that share global data. Users can relax this restriction by using the shared type qualifier.
- *Runtime Services:* ECK provides runtime services to enable data sharing in a controlled manner. For instance, compartments can access the data of the last scheduled compartment by using ECK. Section A describe the complete list of runtime services provided by ECK.

Partition Stats: After the firmware is partitioned into different compartments, the partition generator emits partition statistics for the selected policy. These statistics include the number of instructions, number of objects, dynamic memory usage, number of xcalls, number of Return Oriented Programming (ROP) gadgets, etc. These statistics are generated using static analyses and provide insights about the runtime cost and security implications of the current policy. For instance, the number of xcalls can hint to the user about the potential runtime cost of the particular partitioning policy[2], whereas the number of instructions can give information about the TCB reduction in a system.

Instrumentation Module (IM): The instrumentation module imports the partitioned graph from the *Partition Generator* to instrument the firmware to correctly call different compartments. We conduct a *Field and Context Sensitive Points-To Analysis* to minimize the amount of instrumentation in the firmware, as shown in Algorithm 1. For direct calls, IM is aware of the target function and the target compartment ID, therefore the xcall is hardcoded with those parameters. Contrarily, indirect calls may have multiple targets. To this end, IM uses the points-to analysis and the data flow analysis to estimate the targets of the call. If IM finds multiple targets for an indirect call, it instruments the xcall without a target compartment. During execution, ECK finds the compartment ID using a binary search on the compartment metadata based on the target address. Figure 3 shows an example of running the IM on source code with the file-based partitioning policy. The call at line `89` is not instrumented as the call has only one target that lies within the same compartment. The call at line `92` is instrumented with xcall with the compartment ID. Lastly, line `99` is instrumented with xcall without compartment ID. ECK changes the compartment protections based on the call target at runtime.

ECC Linker: The ECC Linker generates binaries compatible with ECK. The linker 1) places each compartment in a separate section such that the memory layout can be protected

[2]Although this will be a crude estimate as a higher number of xcalls does not mean during runtime more xcalls will be invoked necessarily.

Algorithm 1 ECC Call Instrumentation

Result: $pFirmware$
$funcMap \leftarrow \varnothing$
$targets \leftarrow \varnothing$
foreach $func \in FUNCTIONS$ **do**
  if $addressTaken(func)$ **then**
    $funcMap[func] \leftarrow func$
end
foreach $call \in INDIRECTCALLS$ **do**
  foreach $func \in funcMap$ **do**
    if $type(call) = type(func) \wedge alias(func, call)$ **then**
      $targets[call] \leftarrow func$
  end
end
foreach $func \in FUNCTIONS$ **do**
  $callerID \leftarrow func.id$
  foreach $call \in func.calls$ **do**
    if $type(call) = Direct \wedge target \neq callerID$ **then**
      $pFirmware[inst] \leftarrow Replace(call, xcall(call.func, call.id))$
    else
      if $\forall target_i,\ target_j \in targets[call] \mid target_i = target_j \wedge target \neq callerID$ **then**
        $pFirmware[inst] \leftarrow Replace(call, xcall(target))$
      else
        $pFirmware[inst] \leftarrow Replace(call, xcall())$
  end
end

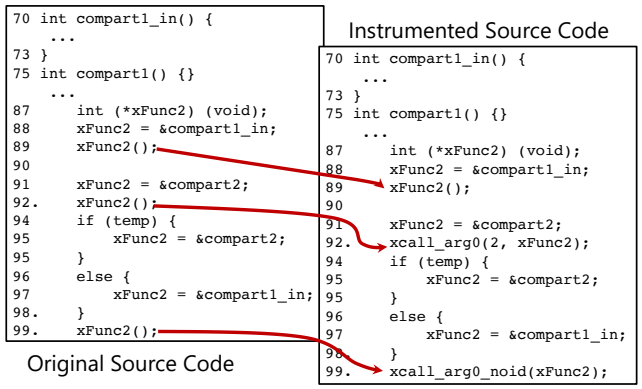

Fig. 3: Running instrumentation module on an example code with *File* policy.

by ECK, and 2) links the metadata generated by ECC to ECK, enabling ECK to enforce the correct memory protection across compartment boundaries at runtime.

B. Compartment Enforcement

EC enforces runtime compartmentalization using ECK. Our key insight behind ECK design is:

A reference monitor for memory protection hardware can enforce memory protections on any firmware.

In other words, if a reference monitor protects the MPU, it can enforce the runtime memory protection for different compartments in the system. To this end, we set up the design goals for our reference monitor and then explore the design spaces to achieve our goals.

1) Design Goals: We derive our initial set of design requirements from the classic reference monitor design principles [46]:

G1: Complete Mediation. The monitor should be able to mediate all accesses to memory configuration within the system, i.e., the monitor should be *non-bypassable* and *always invoked*.

G2: Tamperproofness The monitor should be tamper-proof from any agents outside the TCB, which includes the monitor code and data and the partition metadata to enforce compartmentalization.

G3: Verifiability. The monitor should be verified to ensure correct implementation of **G1:** *Complete Mediation* and **G2:** *Tamperproofness*.

Considering the unique constraints of embedded systems and the limitations of current compartmentalization solutions, we mandate additional requirements for the monitor:

G4: Low Overhead. The monitor should only introduce minimum runtime overhead, without violating the real-time requirements of the system.

G5: Real-Time Design. As the monitor runs beside the RTOS, the monitor itself should be deterministic in its design, i.e., it should update memory configuration in a deterministic manner.

G6: Minimal Changes. The monitor should be able to enforce protection on existing firmwares with minimal changes to ease deployment.

2) Design Exploration: We conduct a thorough literature review of the embedded system design space to design the reference monitor. We evaluate different configurations in terms of the design goals presented in Section VI-B1.

SVC: Existing solutions [28], [50], [52] have employed hardware execution modes to enforce memory protections. Figure 4 shows the SVC-based design for the reference monitor alongside other components. This design requires moving the firmware from the privileged mode to the unprivileged mode, resulting in non-trivial modifications to the embedded firmware, violating **G6 Minimal Changes**. This design also results in non-negligible overhead as shown by existing systems, violating **G4 Low Overhead**.

Software Fault Isolation (SFI): To meet **G6 Minimal Changes**, we can use SFI to ensure that only the monitor can access the MPU without moving the firmware to the unprivileged mode, as shown in Figure 4(b). However, the MPU is implemented as a memory-mapped peripheral. Therefore, protecting the MPU from access from the firmware using SFI requires adding extra checks on each memory access. As shown by existing work [28], [52], this results in a high overhead and does not meet **G4 Low Overhead**.

TEE: Existing work [65] repurposes ARM TrustZone as a hypervisor to host different firmware guests within the same microcontroller. We can utilize this design to declare the memory protection unit (MPU) as a secure device and run the monitor in the secure world. However, upon further research, we found that ARM TrustZone is unable to restrict the MPU configuration registers usage in the normal world. As the rule, R_{LDTN} in the ARM architecture specification [75] states that the System Control Space (SCS), which includes the address range for the MPU, is exempted from security violation checks. In other words, any access from a non-secure world to the SCS cannot be controlled from the secure world. Hence, running the monitor in the secure world does not meet **G1 Complete Mediation**. Furthermore, this design requires ARM TrustZone extensions that are not widely available on existing microcontrollers.

C. ECK Design

While existing designs either move the firmware to the unprivileged mode or use SFI for the monitor, we ask if running the monitor beside the existing firmware as shown in Figure 4(c) is possible. In doing so both the monitor and the RTOS will execute in the privileged mode reducing the changes to the existing firmware and the slowdown of the monitor. The challenge is how to protect the monitor from the RTOS in this design?

We propose ECK, a microkernel as a reference monitor for the MPU by using the hardware watchpoints to watch the MPU configuration, even from the RTOS. More specifically, we utilize the DWT to watch MPU configuration to ensure that there are no rogue accesses to MPU configuration. We further lock the DWT configuration using the MPU. During booting, ECK configures the MPU before passing control to the RTOS and enables the DWT to watch the MPU configuration. If the RTOS tries to access either MPU or DWT to bypass ECK, it would result in either a Debug exception or a Hard Fault exception.

However, similar to security checks of TEE-based design, upon implementation we found that the MPU is unable to protect the DWT. Rule R_{TGQD}[3] states that the MPU uses the default memory map for all memory accesses to the SCS in the privileged mode. Since we run both ECK and the firmware in the same privilege level, the MPU is not able to secure DWT configuration. Instead, we configure the DWT to watch both MPU and DWT configurations to achieve **G1 Complete Mediation**. As shown in Figure 5, if the RTOS tries to modify the MPU or the DWT configuration it results in a Debug Exception served by ECK.

There is still a caveat, the RTOS can mask or override the debug exception to bypass the enforced protections.

[3]The rule is taken from ARMv8-M specification, but the same design applies for ARMv7-M.

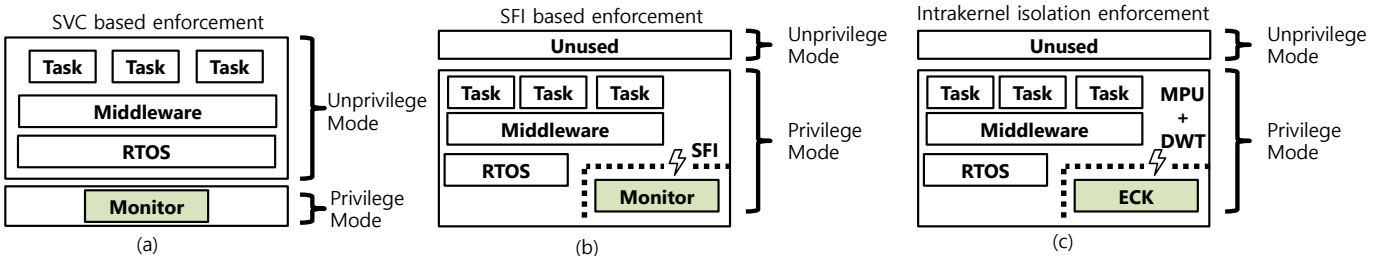

Fig. 4: Different configurations to create a memory protection hardware reference monitor.

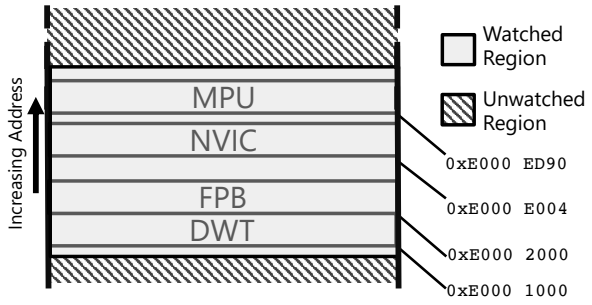

Fig. 5: Intra-kernel isolation by configuring DWT to watch MPU and DWT peripheral I/O regions. The memory map is defined by the ARMv7-M architecture reference specification and should be consistent across all ARMv7-M SoCs.

Listing 1 Code snippet for VeriFast proof assistant. Comments starting with `@` are consumed by the verifier. Some lines are truncated for brevity.

```
11 typedef struct
12 {
13  uint32_t TYPE;
14  uint32_t CTRL;
15  ...
46 } MPU_Type;
47 MPU_Type MPU;
   ...
68 unsigned int switch_view(unsigned int to)
71     //@ requires pointer((void *)&MPU, ...
72     //@ ensures mpuM_CTRL(&mpum, MPU_C ...
74 {
   ...
80    MPU->RNR = 7;
81    //@assert mpuM_RNR(&mpum, 7);
84    if (to < NUM_COMPS) {
85        int temp = (start[to] & MPU_RBAR ...
86        MPU->RBAR = temp;
```

Fortunately, interrupt control is implemented using specialized instructions. These instructions can either 1) disable interrupts globally, or 2) mask interrupts to a specific priority level. We design a static analysis to find these instructions. For 1), we downgrade all global interrupt disabling instructions to interrupt masking instructions. For 2), we configure the debug exception as the highest priority in the system. We ensure that the firmware does not mask interrupts higher than the debug exception's priority level using compile-time instrumentation.

With the watching infrastructure in place, we can also mediate access to other sensitive peripherals. For instance, the firmware can try to override the debug monitor exception by writing a malicious value to the Vector Table Offset Register (VTOR). Similarly, the firmware can try to patch ECK code using the Flash Patch and Breakpoint (FPB) Unit. To this end, ECK also includes the VTOR and the FPB in the watched region and simply denies any modifications to these registers. This restriction does not impact the functionality of normal firmware, as normally firmware does not modify either VTOR or FPB after system bootup. With these restrictions in place, ECK achieves **G1 Complete Mediation**. Furthermore, as ECK is in control of the MPU, it configures the memory protections such that ECK code and data are immutable to ensure **G2 Tamperproofness**.

Since ECK is only responsible for memory configuration, the code size for ECK is small rendering it possible for formal verification. We formally verify ECK to ensure:

- *Memory Safety*: ECK is free of any buffer overflows.
- *Correct Compartment Switching*: After the switching call, ECK always switches to the correct compartment.
- *Thread Safety*: ECK is free of concurrency errors such as data races.

We use VeriFast [45] to prove the guarantees discussed in ECK. For memory and thread safety, VeriFast reports zero errors on ECK, implying that the program is free of illegal memory accesses and data races. For the correctness of compartment switching, we add VeriFast assertions in the code to ensure that ECK switches to the correct configuration supplied by ECC, as shown in Listing 1. We model the MMIO peripherals, i.e., the MPU, using C structures, as shown on Line 47 in Listing 1, and assume the hardware works according to the architecture specification. The small code size and the formal proof of ECK help us achieve **G3 Verifiability**.

Unlike existing solutions, our design does not require a mode switch or an exception for a memory configuration update, and we do not require any extensive code instrumentation for the compartmentalized code. Thanks to this design, ECK imposes a minimal overhead for enforcing memory protection, achieving **G4 Low overhead**, as shown in Section VIII experimentally.

To achieve **G5 Real-Time Design**, each compartment switch is done with the interrupts masked. ECC generates the system configuration statically during compile time and ECK does not use dynamic memory. Because of these considerations, ECK operates in a highly deterministic manner. Lastly, since ECK runs the firmware in the original execution mode, we can run the firmware with minimal changes, achieving **G6 Minimal Changes**.

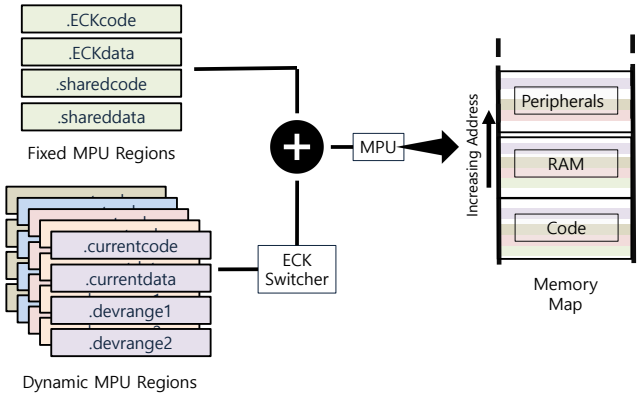

Fig. 6: Switching between different compartments using the MPU. ECK configures the last four regions to the current compartment memory protection based on the configuration generated by ECC.

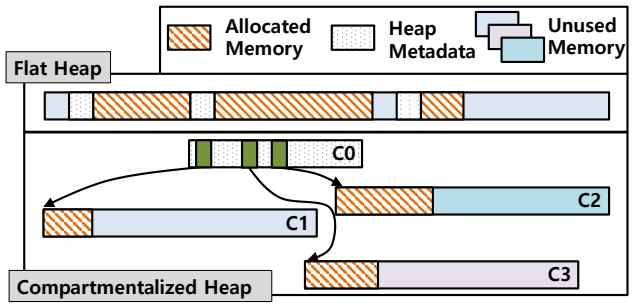

Fig. 7: Layout of the normal flat heap compared to the compartmentalized heap based on the *Heap Analysis*.

In summary, ECK relies on the compile-time compartment configuration generated by ECC for the correct operation between different compartments. As mentioned in Section II, the MPU has alignment and size requirements for the protected regions. Furthermore, the MPU can support up to 16 protection regions, although, in practice, most SoCs provide eight regions. Figure 6 shows the assignment for the MPU regions. ECK uses the lower four regions to enforce the protection for ECK code/-data and shared code/data. For the current running compartment, ECK switches the higher four regions for the current compartment code, data, and two contiguous device memory ranges.

D. Runtime Memory Protection

As shown in Table I, existing solutions ignore dynamic memory allocation as it is generally a hard problem to estimate the dynamic memory required by firmware at compile time. To tackle this problem, existing solutions either allow compartments to access memory across compartments leading to weaker security guarantees, or require profiling the firmware dynamically, thus suffering from the incomplete profiling problem. To overcome the limitations of existing systems, we propose a compartmentalized dynamic memory allocation model.

Dynamic Memory Allocation: Dynamic memory allocation allows firmware to create memory objects at runtime. Memory allocators usually use one single contiguous chunk of memory as the heap and allocate dynamic objects from the heap to

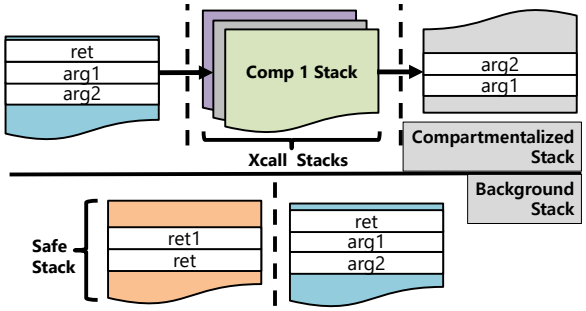

Fig. 8: EC stack protection techniques used by ECK. Compartmentalized stacks copy the local variables to a dedicated stack, whereas Background stack uses a common stack, with a safe stack to maintain control flow integrity.

callers. They use metadata to keep track of the allocated and free memory, which is placed in-band within the same memory chunk.

Due to their runtime nature, dynamically allocated objects escape ECC static analyses while partitioning. While this does not violate the reference monitor guarantees of ECK, the RTOS could crash if it accesses these dynamically allocated objects. Naively, users could place the dynamic memory in a shared section. However, doing so would allow compartments to have free access to dynamic objects of other compartments. To ensure compartmentalization over dynamic objects, we design a compartmentalized memory allocator.

We split the heap buffer into different chunks based on the information provided ECC. Figure 7 shows the changes made from a flat heap to a compartmentalized heap. Each chunk is placed in the respective compartment. The metadata for accounting heap memory is placed into the allocator compartment. The allocator queries ECK about the calling compartment on each allocation request and returns a free chunk from the respective compartment as an opaque pointer. Note that, the allocator is designed as a library on top of ECK and is outside the TCB of ECK.

To minimize the memory overhead of the compartmentalized heap, we design a *Heap Analysis* on the firmware to find compartments in the system that use dynamic memory. It also looks for memory allocation requests to the RTOS, such as the user thread stack and heap. For completeness, we consider all indirect calls as potential dynamic allocation calls. ECC generates the conservative memory consumption estimation of each compartment based on the heap analysis. We remove heap from the compartments that do not utilize dynamic memory.

Stack Protection: Similar to dynamic objects on the heap, dynamically allocated objects on the stack can also escape ECC analyses. Due to this reason, the callee compartment may not have access to the stack of the caller compartment during an xcall. We could also place the stack in the shared section. However, a compromised compartment can not only modify the local objects of other compartments but also hijack the control flow of different compartments. To tackle these security concerns, ECK provides two configuration options for stack hardening. *Compartmentalized Stack:* This configuration allocates a

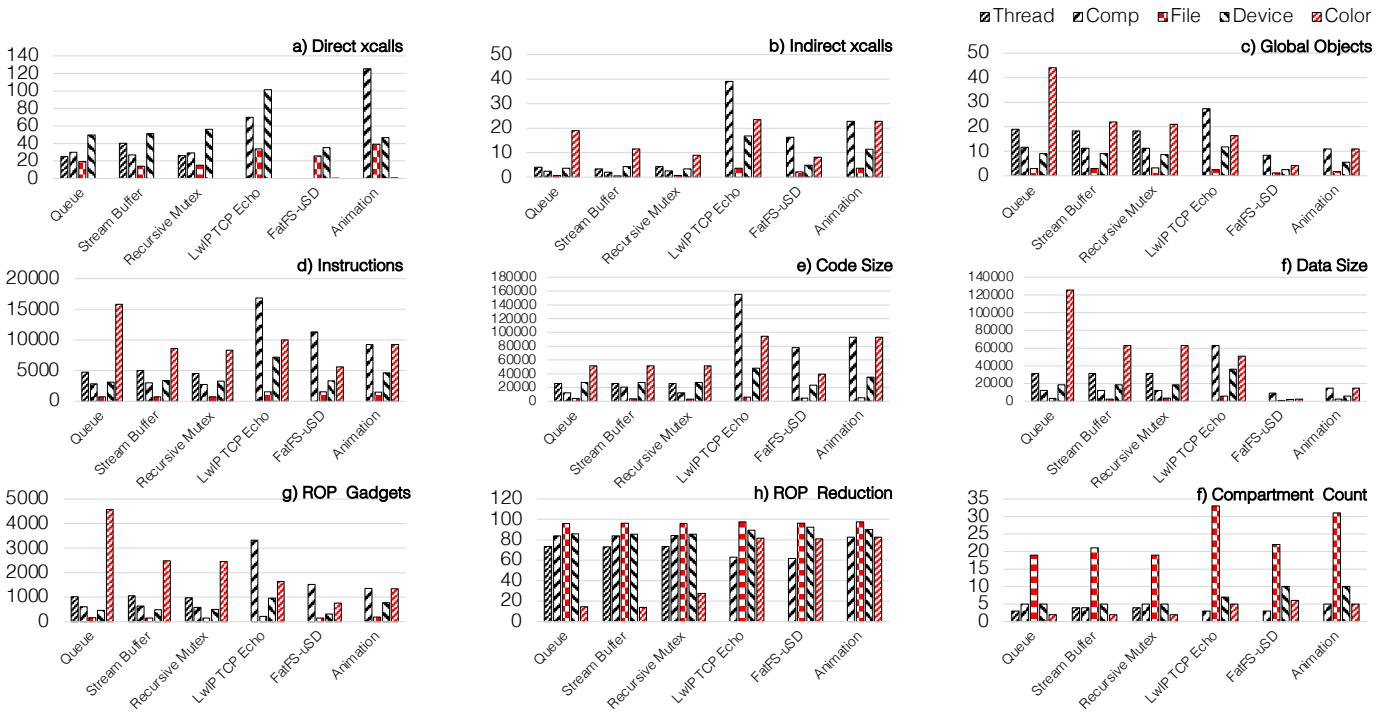

Fig. 9: ECC partitioning stats against different firmware. To summarize the partitions, the average number of xcalls (direct/indirect), global variables, instructions, code size and data size, number of ROP gadgets, and the reduction of ROP gadgets are listed.

bank of xcall stacks within the accessible memory range of respective compartments. During an xcall, the bridge code, operated by ECK, copies the frame of the calling stack to the dedicated xcall stack from the stack bank and switches the stack. Similarly, on return, the return value is copied from the xcall stack to the caller stack before switching the stack. Compartmentalized stacks provide strong isolation between different compartments as the callee is not able to access the stack of the caller compartment.

Background Stack: This design allocates stacks from the shared memory region. Since all stacks are allocated from the shared memory region, the callee can use the stack of the caller compartment. To ensure Control Flow Integrity (CFI) across different xcalls, the xcall bridge code copies the return address of the xcall in the shadow stack, which is not accessible to any compartment in the system, therefore the callee compartment is not able to modify the control flow and always returns to the caller compartment[4].

Figure 8 shows the difference between the two stack protection configurations. The compartmentalized stack provides better inter-compartment isolation, as it guarantees isolation among local variables of different compartments. However, this protection requires an extra copy of the call frame to the compartmentalized stack, resulting in a higher runtime and memory overhead. Background stack mitigates these overheads by allocating stacks in the shared memory regions. Compartments are not able to modify the back edges of an xcall in either configuration.

[4]Forward edges are not protected in this scheme.

VII. Implementation

ECC: The partitioner is built upon Clang-12 and LLVM-12. We use SVF [69] to generate the points-to analysis. Different static analyses mentioned in Section VI are implemented as LLVM passes. We also enhance LLVM to parse SVD file for the running platform. The SVD parser and the *Partition Generator* are implemented as Python scripts, based on CMSIS-SVD [6]. To decouple the design from LLVM, we implemented an intermediate representation (IR), allowing *Partition Generator* plugin writers to work without any prior knowledge about LLVM or Clang. New policies can be implemented as Python classes extending the *Partition Generator*.

The EC linker is implemented as a Python wrapper around the LLVM linker (LDD). We use GNU Binutils and LDD to generate the partitioning layout required for proper protection using MPU. To generate an EC-compatible binary, we use multi-stage linking: 1) EC-linker generates the binary with stubs for EC data and partitioning metadata, 2) EC-linker lays out the memory according to the requirements for MPU restrictions, 3) EC-linker links the binary with the required layout, the initialized ECK, and partitioning metadata into an executable binary.

ECK: ECK is implemented as a self-contained C library to ensure that it is readily linkable to any embedded systems firmware. We use CMSIS [16] device headers to extract the device addresses and offsets required to implement the MPU and DWT drivers.

FreeRTOS: The modifications in FreeRTOS for effective interoperability with ECK are contained within the architecture-specific code of FreeRTOS, whereas the interface annotations

are implemented in the kernel interface headers as described in Section A.

VIII. Evaluation

Our evaluation dataset consists of six applications, including three RTOS and three baremetal applications as described in Section B. First, we evaluate the compartment identification efficacy and partitioning statistics of ECC on the evaluation firmwares, followed by an evaluation of ECK with different firmwares. Lastly, we discuss the security evaluation of EC with different attack scenarios and demonstrate how EC protects against different CVEs.

A. ECC

We evaluate ECC by compartmentalizing the evaluation dataset with different compartmentalization policies and emitting the statistics for each policy. We do not evaluate the thread policy for baremetal applications, as they do not use threads in the firmware. To evaluate the distribution of code and data across each compartment, we use the average number of instructions and global variables across all compartments. We list the number of compartments and the average number of xcalls (both direct and indirect) for all compartments to get an insight into the runtime overhead incurred by the policy. Intuitively, the number of xcalls should be directly proportional to the overhead. However, this is not always true, as there could be compartments with a lower number of xcalls but with higher usage of the xcalls (To this end, we conduct the runtime evaluation Section VIII-B). Similarly, for the security impact of the selected policy, we calculate the average number of ROP gadgets found in all compartments and the average ROP gadget reduction for all compartments. Similarly, we also calculate the functional gadget set expressivity [25] for each policy. The code/data size and the instruction count also hint to the user about the security impact of the selected policy, as a larger instruction count and code/data size leave a larger attack surface.

Figure 9 shows the results of the compartmentalization of the evaluation dataset. Our evaluation shows that stricter policies, such as file policy, result in a higher number of compartments. Consequently, these policies result in more xcalls and better security guarantees. In general, stricter policies result in a higher ROP gadget reduction, whereas each configuration results in a full reduction in the functional gadget set expressivity. Similarly, stricter policies result in a more balanced distribution of memory and code resources. Generally, the file policy results in the most fine-grained compartments, whereas the color policy results in the most coarse-grained compartments.

B. ECK

Micro-benchmark. As shown in Figure 9, different compartmentalization policies lead to diverse partition results. To conduct a policy independent evaluation, we measure the runtime overhead of a single xcall, which implements a wait loop to emulate some workload and does not take any input arguments. We compare the overhead incurred by ECK xcall with existing compartment solutions including SVC, SFI with

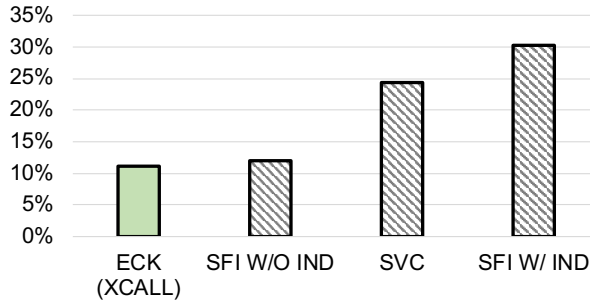

Fig. 10: Comparison of EC with existing compartmentalization compartment enforcement techniques employed by existing systems.

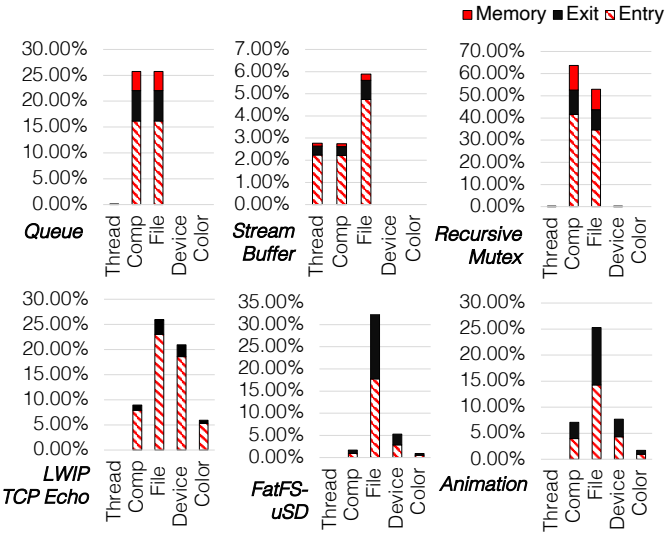

Fig. 11: Execution time of different FreeRTOS applications. The Y-axis shows the percentage overhead incurred.

(W/) and without (W/O) indirect access instrumentation. SVC-based compartment enforcement is used by Minion, ACES, and M2MON, whereas SFI-based compartment enforcement is used by M2MON for interrupt handling routines.

Figure 10 shows the results of the micro-benchmark evaluation. Compared to native execution, ECK xcalls incurs an overhead of 11.13%. SVC incurs 24.36% overhead; SFI incurs 12.00% overhead without indirect access instrumentation and 30.25% overhead with indirect access instrumentations. Based on the micro-benchmark, ECK xcall is around 1.2 times faster than SVC and 1.8 times faster than SFI-based compartment enforcement (on average). ECK outperforms the existing solutions, as it does not require a runtime exception nor does it require extensive code instrumentation to check memory accesses at runtime.

Macro-benchmark. We benchmark the execution time of the firmware using the ARM Systick timer [13]. We use the compartmentalized heap and the background stack configuration for our evaluation. The purpose of the evaluation is two-fold. First, we see the runtime overhead of EC on firmwares at the application level. Secondly, we examine the correlation of ECC compartmentalization statistics with actual runtime overhead.

Figure 11 shows the results of the macro-benchmark evalu-

ation. The `stream buffer` app has the lowest overhead of 2.8% for the thread policy and the highest overhead of 5.9% for the file policy. Similarly, the `queue` app has the lowest overhead of 0.1% for the thread policy and the highest overhead of 25.7% for the file policy. For the `recursive mutex` app, we see the highest overhead for the component policy instead of the file policy. Upon further investigation, we found that the `recursive mutex` component is implemented on top of the queue component. When the component policy is used, ECC identifies them as different compartments. As a result, a single call to a recursive mutex API could result in several xcalls, thus the highest overhead of 63.7% for compartment policy.

For the `LwIP TCP Echo` application, the file policy shows the highest overhead of 25.89%, while the component policy results in 0.01% overhead. For the `FatFS-uSD` application, the file policy results in a 32.65% overhead and the component policy results in the lowest overhead of 1.6%. Lastly, for the `animation` application, file policy results in a 25.2% overhead, whereas the component policy results in the lowest overhead of 7.07%.

The results show a high dependency on the application and the selected policy. In general, the results correlate with ECC heuristics in Figure 9. The color policy incurs the least runtime overhead, whereas the file policy incurs the highest overhead. The color policy usually results in very coarse-grained compartments, resulting in poor security guarantees, whereas the file policy results in very fine-grained compartments, resulting in a large number of xcalls during execution. For the RTOS applications, the thread policy results in the second lowest overhead, because this policy aims to include the resources used by each application within its compartment. As shown in Figure 9, the thread policy introduces no more than four compartments, whereas the file policy generates up to 33 compartments.

Memory overhead. We evaluate EC memory overhead by calculating the size of different sections of compartments and ECK. Figure 12 shows the memory overhead for the evaluation dataset with five different policies. Overall, the memory overhead depends on both the application and the selected policy, except ECK, whose overhead stays almost constant for all of the applications due to 128 bytes large buffer for the shadow stacks placed inside the ECK data section. The microkernel, ECK incurs an average of 2.57% overhead.

ECC places the resource of each compartment, including the stack and heap, in the memory so that ECK can enforce exclusive memory protections for each region. However, this special layout results in fragments across code and data regions of adjacent compartments, as the MPU imposes special requirements on the address alignments of the protected memory regions (See Section II). The fragmentation incurs 2.5x more code memory and 2.3x more data memory for all applications. Overall, EC incurs an average overhead of 2.4x. In general, we see less code fragmentation with finer-grained policies, such as the file policy, although the data fragmentation does not follow a fixed pattern. Note that the fragmentation overhead does not result from EC, but is incurred due to the limitation of the MPU design.

C. Security Evaluation:

EC drastically reduces the attack surface of the firmware. In this section, we evaluate the security guarantees of EC. More specifically, we assume different vulnerabilities in a compromised compartment and evaluate 1) if the compromised compartment is able to break the reference monitor guarantees of ECK, and 2) if the compromised compartment can modify the contents of other compartments.

Buffer Overflow: Buffer overflow arises when memory buffers are indexed beyond the bounds of the buffer without proper checking. A malicious user can index the memory buffer to manipulate program memory.

Stack-Based Overflow: In this experiment, we allow a compartment to overflow buffers on the stack. As explained in Section VI, ECK provides two modes of stack protection. For the compartmentalized stack design, the compromised compartment is unable to access other stacks, because of the ECK compartmentalization enforcement. For the background stack design, the compromised compartment is able to access the local variables for other compartments. However, the compromised compartment was unable to alter the memory protections of the current compartment. As the return address is always saved in the safe stack region inside ECK, which is not accessible to any compartment in the system.

Heap-Based Overflow: In this experiment, we assume a heap overflow in one compartment, which could allow attackers to overwrite: 1) the heap metadata and 2) the memory owned by other compartments, including the stack. In EC, for 1) the heap metadata is kept in a separate compartment, thus a malicious compartment should not be able to access the heap metadata. For 2), the compartmentalized heap ensures that dynamic memory is exclusively owned by each compartment. An attacker can only corrupt the data belonging to its compartment and should not be able to propagate its corruption to other compartments.

CVE-2018-16528 [79], CVE-2018-16525 [77], and CVE-2018-16526 [78] are examples of buffer overflows in FreeRTOS. Using EC, we experimentally show that these CVEs can be confined within their affected compartments, instead of compromising the whole system.

Privilege Escalation: Privilege escalation attacks exploit vulnerabilities in the system to manipulate privileged resources that are typically owned by the RTOS. CVE-2022-22733 [76] is an example of privilege escalation in QNX RTOS. Since EC provides fine-grained compartments, a generalization of this attack is to access resources belonging to other compartments. For instance, an attacker can 1) directly access the memory belonging to another compartment, or 2) modify the memory protection range of its compartment. For 1), accessing the memory (code/data) belonging to another compartment results in an illegal memory access exception as each compartment has its exclusive memory range. For 2), the memory configuration metadata is loaded from the Read-Only Memory (ROM) and once it is loaded to RAM, ECK ensures the configuration is read-only, disallowing a compartment to access memory outside its compartments. Therefore, an attacker cannot escalate the privilege beyond its own compartment.

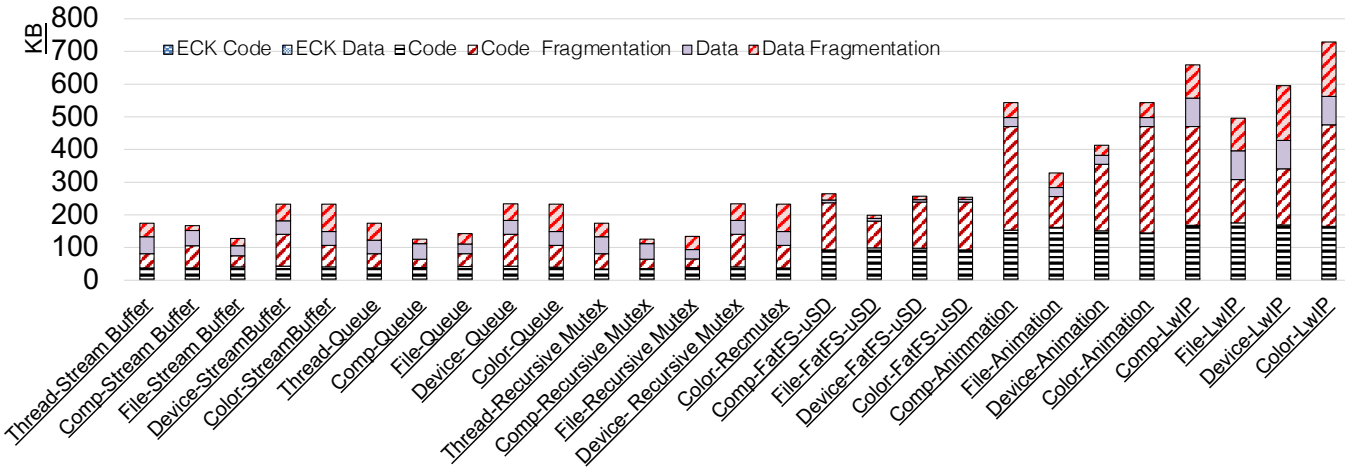

Fig. 12: Memory overhead of using EC on different firmwares.

One interesting implication of such attacks is the *Confused Deputy Attack* [39], where instead of accessing privileged resources directly, a compromised compartment can use available interfaces to maliciously manipulate resources beyond its privilege. EC does not enforce any interface design on the RTOS for compatibility reasons. If the RTOS is vulnerable to these attacks, an attacker might use xcalls to carry out the attack, which could be defended by existing preventive techniques [24], [26], [67], [68], [92].

Format String: Format string attack uses a malicious format specifier string to access arbitrary memory locations in variadic functions[5]. These attacks exploit the interpretation of the format string specifiers to write to arbitrary memory locations. CVE-2021-43041 [86], CVE-2021-35331 [90], and CVE-2021-30145 [84] are examples of format string vulnerabilities. Since an xcall could be variadic and the bridge has to interpret the format string, a fabricated format string can allow access to the memory of the callee compartment. Since EC cannot enforce the proper checking required by format string, EC cannot eliminate such vulnerabilities. However, EC ensures that the corruption is confined within its own compartment. Furthermore, EC does not allow xcalls with variable arguments by default. Users can either modify the ECC policy or the variadic interface to use fixed argument interfaces to overcome this limitation. Note that, users can still use variadic functions within a compartment.

Illegal Pointer Value: In memory-unsafe languages, a pointer could be uninitialized, null, or pointing to an object that is not owned by the pointer, i.e., dangling pointers. CVE-2021-3322 [85], CVE-2020-1939 [83], and CVE-2020-10066 [82] are examples of malformed pointer accesses in an RTOS. Using EC, if a compartment dereferences an illegal pointer pointing to memory in some other compartment, it will result in an illegal memory access exception.

Code Injection Attack: Code injection attacks write malicious code to writable memory and transfer control to the malicious code. CVE-2022-23603 [5], CVE-2022-23120 [4], and CVE-2022-0895 [3] are examples of code injection vulnerabilities. To this end, EC ensures Data Execution Prevention (DEP) by enforcing $W \oplus X$ protection for compartment memory, i.e., if a memory region is writable, it cannot executable at the same time.

[5]Functions with a variable number of arguments.

IX. Discussion

DMA Attacks: Currently, we do not consider DMA attacks in our threat model. However, users can add EC-awareness to the DMA device driver for some extent of DMA level isolation. More specifically, the DMA Controller can be compartmentalized into a separate compartment and can serve DMA requests based on the configuration data generated by ECC. The DMA controller can ensure that compartments are not issuing transfers to memory or peripherals beyond their compartment. However, this design still suffers from several limitations: Firstly, without the presence of an IOMMU, malicious peripherals may trigger vulnerabilities in the RTOS using DMA [9]. Examples of such peripherals include CAN bus controller, USB controller, WiFi controller, etc. Furthermore, even if we assume the availability of an IOMMU, existing work [14], [60], [61] has shown that the peripherals can exploit the DMA interface within the constraints imposed by an IOMMU. Lastly, if we assume all peripherals are secure, we might still face the confused deputy problem if the original firmware is not designed properly. Existing work [62] has shown that legacy firmware can be modified to adapt Extensible Access Rights (EAR) to mitigate the confused deputy problem. We intend to tackle these challenges in future work.

Scaling EC to Multicore: ARM-based multicore systems work by instantiating the single core multiple times and providing an SoC-specific mechanism for interprocessor communication. Each core runs a separate firmware in these systems and exceptions (such as the DebugMonitor exception) are kept private to each core. Scaling EC to these multicore systems would require running ECK on each core. Furthermore, EC should ensure that it is the first agent executing on each core to ensure EC can set up the memory protections before transferring control to the RTOS. Furthermore, EC has to ensure that it is still the owner of all of the required exceptions, such as the DebugMonitor exception, in the multicore setup.

However, EC cannot protect against race condition vulnerabilities, such as Time-Of-Check-To-Time-Of-Use (TOCTTOU), on multicore systems. CVE-2019-17102 [81] and CVE-2019-11482 [80] are examples of race condition vulnerabilities. Developers still need to rely on existing synchronization primitives. However, accessing variables outside their compartment should result in access fault. We leave support for multicore systems as future work.

Compartmentalization Policies: ECC implements various compartmentalization policies for partitioning an existing firmware. Meanwhile, EC supports modifying existing policies or creating new ones. For instance, we can compartmentalize the firmware based on data flows to minimize the amount of information flow across different compartments. We plan to investigate new partitioning schemes in the future.

Trusted Computing: ARM TrustZone provides a trusted hardware execution environment that runs code securely with isolation from the non-secure world. As mentioned in Section VI, EC does not use TrustZone as it fails to meet the reference monitor guarantees. However, it is possible to treat TrustZone as a secure partition and enhance ECC to move code into the secure world. We plan to pursue this approach as a future research direction.

Partitioning Runtime Heuristics: ECC generates partitioning stats that help estimate the runtime overhead. Currently, the stats are purely generated based on static analysis. Due to the static nature of our estimates, the current estimates could present an inaccurate representation of runtime overhead as we do not know how frequently particular xcalls are executed on runtime. Usually, the number of xcalls is directly proportional to the overhead incurred, however, this is might not be always true. Augmenting the runtime stats with either 1) developer-provided hints about the frequency or 2) dynamic runtime profiles should help ECC to generate better estimations for runtime costs. Furthermore, finding more statistics to estimate the runtime overhead remains an open problem. We plan to explore this direction further in future work.

X. Related Work

Compartmentalization Enforcement: There has been extensive work on enforcing runtime compartments. Secure Virtual Architecture [30] introduces a new architecture to enforce memory safety and control-flow integrity. Nested Kernel [32] uses a kernel that is in charge of memory operations with x86_64 WP bit to ensure that only the nested kernel can do memory operations. Lightweight Virtualized Domains [63] uses VM functions (VMFUNC) to create different compartments in the system. xMP [66] uses the alt2pm feature of Xen [21] hypervisor to enforce partitioning at runtime. Mondrix [74] Memory Isolation for Linux uses Mondriaan Memory Protection (MMP) [73] to create different protection domains. Shreds [27] uses the domains feature of the ARM memory protection system to create lightweight threads (called shreds) using a specialized compiler, kernel module, and userspace library. PUMP [35] tackles memory-related challenges such as isolation, corruption, and spatial and temporal issues by complementing software with metadata processes using hardware extensions. Hodor [40] uses Intel MPK [64] to protect libraries from the main applications by analyzing the binary and watching instruction sequences that can modify the PKRU register. ERIM [70] achieves the same goal by using binary patching to forbid PKRU updates. Program-mandering [59] compartmentalizes software using a weighted PDG. It splits a program into low-integrity and high-secrecy domains. Hsu et al. [42] provide APIs to secure multithreaded applications so that each thread can have a separate memory view. TZ-RKP [19] uses binary analysis and TrustZone to provide lifetime integrity to the kernel. Similarly, Hypersafe [71] uses intel's virtualization extension and the WP bit to provide lifetime integrity to the kernel. However, all of these systems rely on features unavailable to embedded systems. EC provides the same level of compartmentalization enforcement within the constraints of embedded systems.

Microkernels: L3 [36] implements a microkernel with a lightweight IPC mechanism. L4 [18] implements the next generation of L3 by further reducing IPC overhead. seL4 [54] implements a formally verified version of the L4 kernel. L4Linux [87] is a Linux kernel port for the L4RE (Fiasco.OC) [88] microkernel. TinyOS [57] implements a component-based OS build using nesC [38]. PikeOS (previously P4) [47] modifies the scheduling, partitioning, and mapping infrastructure of the L4 kernel for embedded systems. CAmkES [56] implements a component-based architecture that builds on Iguana IPC and L4 for embedded systems. In general, microkernels are notorious for their runtime overhead, as they divide the monolithic kernel into small components, which are isolated from each other. Each cross-component interaction incurs a runtime overhead. EC intrakernel isolation provides a fast mechanism to switch between components.

Embedded Systems Security Frameworks: There has been a plethora of existing work on securing embedded systems using various techniques [43], [44], [53], [93], [94]. Epoxy [29] implements security mechanisms such as CFI for firmwares. μRAI [15] prevents control-flow hijacking attacks targeting backward edges by enforcing the Return Address Integrity. Multiple tools [28], [50], [52] have tried to compartmentalize embedded systems to reduce the attack surface. Similarly, existing work [49], [55], [72] has used hardware security extensions to provide trusted computing in embedded systems. However, existing work either requires non-trivial modifications to the existing firmware or incurs a high overhead to achieve its security goals. In contrast, EC introduces a low overhead and allows to run firmware with minimal changes.

XI. Conclusion

In this paper, we present EC, an automatic compartmentalization framework that uses program analysis to identify compartments in bare-metal and RTOS firmware. EC partitions firmware using a custom toolchain, ECC, and enforces runtime protection using ECK, a formally-verified microkernel implementing a novel OS architecture. We evaluate EC on real-world firmwares and show that EC is 1.2x faster than state-of-the-art compartmentalization techniques, and can achieve up to 96.2% ROP gadget reduction.

Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by ONR under Grant N00014-1-21-2328. This work is also based on research sponsored by NSF under Grant 1801601. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR or NSF. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government.

REFERENCES

[1] . Cve - cve-2021-31571. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31571. (Accessed on 10/12/2021).

[2] . Cve - cve-2021-31572. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31572. (Accessed on 10/12/2021).

[3] . Nvd - cve-2022-0895. https://nvd.nist.gov/vuln/detail/CVE-2022-0895. (Accessed on 03/14/2022).

[4] . Nvd - cve-2022-23120. https://nvd.nist.gov/vuln/detail/CVE-2022-23120. (Accessed on 03/14/2022).

[5] . Nvd - cve-2022-23603. https://nvd.nist.gov/vuln/detail/CVE-2022-23603. (Accessed on 03/14/2022).

[6] posborne/cmsis-svd: Aggegration of arm cortex-m (and other) cmsis svds and related tools. https://github.com/posborne/cmsis-svd. (Accessed on 10/31/2022).

[7] . Project zero: Over the air - vol. 2, pt. 1: Exploiting the wi-fi stack on apple devices. https://googleprojectzero.blogspot.com/2017/09/over-air-vol-2-pt-1-exploiting-wi-fi.html. (Accessed on 09/30/2021).

[8] . Project zero: Over the air: Exploiting broadcom's wi-fi stack (part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html. (Accessed on 09/30/2021).

[9] Project zero: Over the air: Exploiting broadcom's wi-fi stack (part 1). https://googleprojectzero.blogspot.com/2017/04/over-air-exploiting-broadcoms-wi-fi_4.html. (Accessed on 10/11/2022).

[10] Stm32f769i-eval - evaluation board with stm32f769ni mcu - stmicroelectronics. https://www.st.com/en/evaluation-tools/stm32f769i-eval.html. (Accessed on 10/19/2022).

[11] Stmicroelectronics/stm32cubef4: Stm32cube mcu full package for the stm32f4 series - (hal + ll drivers, cmsis core, cmsis device, mw libraries plus a set of projects running on all boards provided by st (nucleo, evaluation and discovery kits)). https://github.com/STMicroelectronics/STM32CubeF4. (Accessed on 10/18/2022).

[12] Stmicroelectronics/stm32cubef7: Stm32cube mcu full package for the stm32f7 series - (hal + ll drivers, cmsis core, cmsis device, mw libraries plus a set of projects running on all boards provided by st (nucleo, evaluation and discovery kits)). https://github.com/STMicroelectronics/STM32CubeF7. (Accessed on 10/18/2022).

[13] Systick timer (systick). https://www.keil.com/pack/doc/CMSIS/Core/html/group__SysTick__gr.html. (Accessed on 10/30/2022).

[14] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. Characterizing, exploiting, and detecting dma code injection vulnerabilities in the presence of an iommu. In *Proceedings of the Sixteenth European Conference on Computer Systems*, pages 395–409, 2021.

[15] Naif Almakhdhub, Abraham Anthony Clements, Saurabh Bagchi, and Mathias Payer. urai: Return address integrity for embedded systems. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2020.

[16] ARM. Common microcontroller software interface standard (cmsis) – arm developer. https://developer.arm.com/tools-and-software/embedded/cmsis. (Accessed on 03/26/2022).

[17] ARM. System view description. https://www.keil.com/pack/doc/cmsis/SVD/html/index.html. (Accessed on 03/25/2022).

[18] Alan Au and Gernot Heiser. *L4 user manual*. Citeseer, 1998.

[19] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 90–102, 2014.

[20] Ahmed M Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *NDSS*, volume 16, pages 21–24, 2016.

[21] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. *ACM SIGOPS operating systems review*, 37(5):164–177, 2003.

[22] Richard Barry. Freertos - market leading rtos (real time operating system) for embedded systems with internet of things extensions. https://www.freertos.org/. (Accessed on 03/26/2022).

[23] Carlos Augusto Tovar Bonilla, Octavio José Salcedo Parra, and Jhon Hernán Díaz Forero. Common security attacks on drones. *International Journal of Applied Engineering Research*, 13(7):4982–4988, 2018.

[24] Scott Brookes and Stephen Taylor. Containing a confused deputy on x86: A survey of privilege escalation mitigation techniques. *International Journal of Advanced Computer Science and Applications*, 2016.

[25] Michael D Brown and Santosh Pande. Is less really more? why reducing code reuse gadget counts via software debloating doesn't necessarily indicate improved security. *arXiv preprint arXiv:1902.10880*, 2019.

[26] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19. Citeseer, 2012.

[27] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. Shreds: Fine-grained execution units with private memory. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 56–71. IEEE, 2016.

[28] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. {ACES}: Automatic compartments for embedded systems. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 65–82, 2018.

[29] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 289–303. IEEE, 2017.

[30] John Criswell, Andrew Lenharth, Dinakar Dhurjati, and Vikram Adve. Secure virtual architecture: A safe execution environment for commodity operating systems. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 351–366, 2007.

[31] Ang Cui, Michael Costello, and Salvatore Stolfo. When firmware modifications attack: A case study of embedded exploitation. *NDSS*, 2013.

[32] Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell, and Vikram Adve. Nested kernel: An operating system architecture for intra-kernel privilege separation. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 191–206, 2015.

[33] Drew Davidson, Hao Wu, Rob Jellinek, Vikas Singh, and Thomas Ristenpart. Controlling uavs with sensor input spoofing attacks. In *10th {USENIX} Workshop on Offensive Technologies ({WOOT} 16)*, 2016.

[34] Jyoti Deogirikar and Amarsinh Vidhate. Security attacks in iot: A survey. In *2017 International Conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud)(I-SMAC)*, pages 32–37. IEEE, 2017.

[35] Udit Dhawan, Catalin Hritcu, Raphael Rubin, Nikos Vasilakis, Silviu Chiricescu, Jonathan M Smith, Thomas F Knight Jr, Benjamin C Pierce, and Andre DeHon. Architectural support for software-defined metadata processing. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 487–502, 2015.

[36] Kevin Elphinstone and Gernot Heiser. From l3 to sel4 what have we learnt in 20 years of l4 microkernels? In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 133–150, 2013.

[37] Zhiwei Feng, Nan Guan, Mingsong Lv, Weichen Liu, Qingxu Deng, Xue Liu, and Wang Yi. Efficient drone hijacking detection using onboard motion sensors. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1414–1419. IEEE, 2017.

[38] David Gay, Philip Levis, Robert Von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. *Acm Sigplan Notices*, 38(5):1–11, 2003.

[39] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *ACM SIGOPS Operating Systems Review*, 22(4):36–38, 1988.

[40] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, pages 489–504, 2019.

[41] Michael Hooper, Yifan Tian, Runxuan Zhou, Bin Cao, Adrian P Lauf, Lanier Watkins, William H Robinson, and Wlajimir Alexis. Securing commercial wifi-based uavs from common security attacks. In *MILCOM 2016-2016 IEEE Military Communications Conference*, pages 1213–1218. IEEE, 2016.

[42] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. Enforcing least privilege memory views for multithreaded applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 393–405, 2016.

[43] Muhammad Ibrahim, Andrea Continella, and Antonio Bianchi. Aot - attack on things: A security analysis of iot firmware updates. In *2023 IEEE 8th European Symposium on Security and Privacy (EuroS&P)*, 2023.

[44] Muhammad Ibrahim, Abdullah Imran, and Antonio Bianchi. Safetynot: on the usage of the safetynet attestation api in android. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 150–162, 2021.

[45] Bart Jacobs, Jan Smans, Pieter Philippaerts, Frédéric Vogels, Willem Penninckx, and Frank Piessens. Verifast: A powerful, sound, predictable, fast verifier for c and java. In *NASA formal methods symposium*, pages 41–55. Springer, 2011.

[46] Trent Jaeger. Reference monitor., 2011.

[47] Robert Kaiser and Stephan Wagner. Evolution of the pikeos microkernel. In *First International Workshop on Microkernels for Embedded Systems*, volume 50, 2007.

[48] Paul A Karger. Limiting the damage potential of discretionary trojan horses. In *1987 IEEE Symposium on Security and Privacy*, pages 32–32. IEEE, 1987.

[49] Arslan Khan, Joseph I Choi, Dave Tian, Tyler Ward, Kevin RB Butler, Patrick Traynor, John M Shea, and Tan F Wong. Enclave-based privacy-preserving localization. In *In 2021 IEEE 12th Annual Ubiquitous Computing, Electronics & Mobile Communication Conference (UEMCON)*, 2021.

[50] Arslan Khan, Hyungsub Kim, Byoungyoung Lee, Dongyan Xu, Antonio Bianchi, and Dave Jing Tian. M2mon: Building an mmio-based security reference monitor for unmanned vehicles. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[51] Douglas Kilpatrick. Privman: A library for partitioning applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 273–284, 2003.

[52] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *NDSS*, 2018.

[53] Hyungsub Kim, Muslum Ozgur Ozmen, Z Berkay Celik, Antonio Bianchi, and Dongyan Xu. Pgpatch: Policy-guided logic bug patching for robotic vehicles. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1826–1844. IEEE, 2022.

[54] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, et al. sel4: Formal verification of an os kernel. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 207–220, 2009.

[55] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the Ninth European Conference on Computer Systems*, pages 1–14, 2014.

[56] Ihor Kuz, Yan Liu, Ian Gorton, and Gernot Heiser. Camkes: A component model for secure microkernel-based embedded systems. *Journal of Systems and Software*, 80(5):687–699, 2007.

[57] Philip Levis, Samuel Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, et al. Tinyos: An operating system for sensor networks. In *Ambient intelligence*, pages 115–148. Springer, 2005.

[58] Lulu Liang, Kai Zheng, Qiankun Sheng, and Xin Huang. A denial of service attack method for an iot system. In *2016 8th international conference on Information Technology in Medicine and Education (ITME)*, pages 360–364. IEEE, 2016.

[59] Shen Liu, Dongrui Zeng, Yongzhe Huang, Frank Capobianco, Stephen McCamant, Trent Jaeger, and Gang Tan. Program-mandering: Quantitative privilege separation. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 1023–1040, 2019.

[60] Theo Markettos, Colin Rothwell, Brett F Gutstein, Allison Pearce, Peter G Neumann, Simon Moore, and Robert Watson. Thunderclap: Exploring vulnerabilities in operating system iommu protection via dma from untrustworthy peripherals. 2019.

[61] Alex Markuze, Adam Morrison, and Dan Tsafrir. True iommu protection from dma attacks: When copy is faster than zero copy. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 249–262, 2016.

[62] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. D-Box: DMA-enabled Compartmentalization for Embedded Applications. In *Proceedings 2022 Network and Distributed System Security Symposium*, 2022.

[63] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 157–171, 2020.

[64] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel {MPK}). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, 2019.

[65] Sanndro Pinto, Hugo Araujo, Daniel Oliveira, Jose Martins, and Adriano Tavares. Virtualization on trustzone-enabled microcontrollers? voilà! In *2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 293–304. IEEE, 2019.

[66] Sergej Proskurin, Marius Momeu, Seyedhamed Ghavamnia, Vasileios P Kemerlis, and Michalis Polychronakis. xmp: selective memory protection for kernel and user space. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 563–577. IEEE, 2020.

[67] D. Sbîrlea, M. G. Burke, S. Guarnieri, M. Pistoia, and V. Sarkar. Automatic detection of inter-application permission leaks in android applications. *IBM Journal of Research and Development*, 57(6):10:1–10:12, 2013.

[68] Darius Suciu, Stephen McLaughlin, Hayawardh Vijayakumar, Lee Harrison, Michael Grace, and Amir Rahmati. Poster: Automatic detection of confused-deputy attacks on arm trustzone environments. *IEEE SecDev*, 2018.

[69] Yulei Sui and Jingling Xue. Svf: interprocedural static value-flow analysis in llvm. In *Proceedings of the 25th international conference on compiler construction*, pages 265–266, 2016.

[70] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. {ERIM}: Secure, efficient in-process isolation with protection keys ({MPK}). In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1221–1238, 2019.

[71] Zhi Wang and Xuxian Jiang. Hypersafe: A lightweight approach to provide lifetime hypervisor control-flow integrity. In *2010 IEEE Symposium on Security and Privacy*, pages 380–395, 2010.

[72] Thomas Winkler and Bernhard Rinner. Securing embedded smart cameras with trusted computing. *EURASIP Journal on Wireless Communications and Networking*, 2011:1–20, 2011.

[73] Emmett Witchel, Josh Cates, and Krste Asanović. Mondrian memory protection. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 304–316, 2002.

[74] Emmett Witchel, Junghwan Rhee, and Krste Asanović. Mondrix: Memory isolation for linux using mondriaan memory protection. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 31–44, 2005.

[75] Armv8-m architecture reference manual. https://developer.arm.com/documentation/ddi0553/latest. (Accessed on 01/28/2022).

[76] Cve - cve-2008-3024. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-3024. (Accessed on 01/25/2022).

[77] Cve - cve-2018-16525. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16525. (Accessed on 01/25/2022).

[78] Cve - cve-2018-16526. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16526. (Accessed on 01/25/2022).

[79] Cve - cve-2018-16528. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16528. (Accessed on 01/26/2022).

[80] Cve - cve-2019-11482. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-11482. (Accessed on 01/25/2022).

[81] Cve - cve-2019-17102. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-17102. (Accessed on 01/25/2022).

[82] Cve - cve-2020-10066. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-10066. (Accessed on 01/25/2022).

[83] Cve - cve-2020-1939. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-1939. (Accessed on 01/25/2022).

[84] Cve - cve-2021-30145. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-30145. (Accessed on 01/25/2022).
[85] Cve - cve-2021-3322. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3322. (Accessed on 01/25/2022).
[86] Cve - cve-2021-43041. https://cve.mitre.org/cgi-bin/cvename.cgi?name=2021-43041. (Accessed on 01/25/2022).
[87] L4linux. https://l4linux.org/. (Accessed on 01/09/2022).
[88] L4re – the l4 runtime environment. https://l4re.org/. (Accessed on 01/09/2022).
[89] M-profile architectures – arm developer. https://developer.arm.com/architectures/cpu-architecture/m-profile. (Accessed on 01/06/2022).
[90] Nvd - cve-2021-35331. https://nvd.nist.gov/vuln/detail/CVE-2021-35331. (Accessed on 01/25/2022).
[91] Stm32f4discovery - discovery kit with stm32f407vg mcu * new order code stm32f407g-disc1 (replaces stm32f4discovery) - stmicroelectronics. https://www.st.com/en/evaluation-tools/stm32f4discovery.html. (Accessed on 01/09/2022).
[92] Jianliang Wu, Tingting Cui, Tao Ban, Shanqing Guo, and Lizhen Cui. Paddyfrog: systematically detecting confused deputy vulnerability in android applications. *Security and Communication Networks*, 8(13):2338–2349, 2015.
[93] Jianliang Wu, Ruoyu Wu, Daniele Antonioli, Mathias Payer, Nils Ole Tippenhauer, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Lightblue: Automatic profile-aware debloating of bluetooth stacks. In *Proceedings of the USENIX Security Symposium (USENIX Security)*, 2021.
[94] Ruoyu Wu, Taegyu Kim, Dave Jing Tian, Antonio Bianchi, and Dongyan Xu. {DnD}: A {Cross-Architecture} deep neural network decompiler. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2135–2152, 2022.

APPENDIX

APPENDIX A
INTEROPERABILITY WITH RTOSS

While RTOS should be able to run without any modifications with ECK, for better security guarantees (such as least privilege) and an efficient cooperation with RTOS, we design different interoperability services.

Bootstrapping. To reduce the TCB, we do not include any boot logic inside ECK. To this end, we need to patch the boot code of the RTOS to invoke ECK. We do not lose any security guarantees in doing so, as the RTOS has to invoke the ECK for the correct setup. Otherwise, as soon as the firmware encounters the first xcall, it will leave the memory configuration in an inconsistent state leading to a crash.

Interface Design. ECC instruments the firmware with xcalls without any assumptions of these calls. As a consequence, all xcalls are instrumented conservatively to ensure correct operation. For instance, if an xcall passes a pointer as an argument, the xcall bridge code ensures to copy the memory pointed in the calling compartment to a shared buffer and passes the shared buffer as an argument to the callee compartment. To guide the compartmentalization, we introduce new type qualifiers to C types guiding the automatic compartmentalization by ECC. The added extensions are as follows:

Opaque Pointers: RTOSs often use opaque pointers to provide user tasks handles to kernel objects. These objects can have a large size and if they are used as arguments to xcalls, they incur significant overhead for copying between compartments. Since ECC cannot statically determine opaque pointers, we introduce the `opaque` type qualifier that informs ECC and ECK to pass the argument as an opaque pointer. For `opaque` pointers, ECC does not copy the pointed memory to the shared buffer and passes the pointer without any modifications to the callee compartment. Developers can use opaque pointers to securely share handles to objects with secure isolation from user threads. Due to the memory protections enforced by ECK, if a user thread accesses an opaque pointer, it results in an illegal memory access exception.

Custom Bridging: Some functions require context-specific information to correctly process arguments. For instance, a queue creation function could take the size of each data item in the queue and save it in the queue handle. Since the queue utility already knows the size of the items enqueued, the queue append function could have a polymorphic pointer to the data to be enqueued without size information. To correctly copy the complete memory pointed by the polymorphic pointer, we need a custom method that can copy arguments based on the item size in the queue handle. ECC provides the `custom_bridge` annotation to facilitate such scenarios. If a user-supplied function with `custom_bridge` annotation is called, ECC expects a custom bridge function and terminates with a compilation error if it is not found.

Helper/Utility Functions: Some functions purely work on data supplied by other compartments. For instance, a list utility can provide a helper function for creating and manipulating lists. To reduce xcalls for such functions, we can either 1) provide each compartment with a local version of the utility library, or 2) allow the utility compartment to gain access to the calling compartment.

ECK User Routines: Some functions such as context switching routines need access to various compartments. As these functions need cross-compartment accesses, we include them inside ECK via the `"eck_user"` annotation. We use SFI on user routines to ensure that they cannot modify the ECK data. We keep the number of these routines to a minimum to minimize the potential impact on the established least privilege in the system and runtime performance.

Context Switching. RTOSs usually use the thread stack to save the context of a running thread, which requires a thread compartment to have a complete access to almost all stacks in the system. Instead, we design a *Split Context Stack* architecture to restrict the cross-compartment accesses. In this design, once the initial stack is populated using ECK *User Routines*, the threading compartment saves the context of the running thread on a separate stack, called the *Context Stack*, which is allocated inside the threading component and is exclusively for saving the context of the running thread. The threading compartment has exclusive access to the context stacks.

Interrupt Handling. The interrupt handling routines could be contained within a single compartment or multiple compartments. During normal execution, the compartment of the running compartment is enforced and is often different from the interrupt handlers. To this end, EC instruments interrupt handler memory protection using a trampoline mechanism, which sets the memory view of the interrupt handler, serves the interrupt request, and switches back to the previous memory view.

Table III shows the complete set of type extensions added by EC. Users can use these qualifiers to tune the behavior of ECC partitions.

Qualifier	Target	Input	Usage
OPAQUE	Function argument	None	If the base type is a pointer type ECK does not copy the pointed memory to the target compartment.
STRING	Function argument	None	If the base type is a pointer type, ECC uses strlen to infer the size of the pointed memory.
LEN(AR)	Function argument	Integer	If the base type is a pointer type, ECK uses the value of ARth argument as size of pointed memory.
UTILITY	Function prototype	None	Utility functions can access the memory belonging to the calling compartment.
SHARED	Global Object	None	The qualified variable will be accessible to all compartments in the system.
USER	Function prototype	None	The qualified function has global access to memory, beside ECK memory.
SIZE(SI)	Function argument	Integer	If the base type is a pointer type, ECK copies SI bytes to the target compartment.
CUSTOM	Function argument	None	The qualified function uses a custom bridge with the same identifier suffixed by "custom".

TABLE III: Different type qualifier offered by EC to guide the partitioning behavior.

APPENDIX B
EVALUATION DATASET AND SETUP.

We include six applications from three different projects in our evaluation dataset. We describe the evaluated projects and provide a brief description of the evaluated applications.

FreeRTOS [22]: We use different applications shipped with FreeRTOS for our evaluation. We pick these applications to cover different features of FreeRTOS. We run the RTOS applications on STM32F407VG [91], including `stream buffer`, `queue` and `recursive mutex`. `stream buffer` uses a single-reader single-writer queue to communicate between two tasks. `queue` uses general purpose queues to communicate between two tasks. `recursive mutex` uses a recursive mutex to synchronize between different tasks.

STM32F4Cube [11]: STM32F4Cube is a firmware package repository for STM32F4 family of microcontrollers. We use STM32F469NI [91] to run these applications. `FatFS-uSD` and `Animation` applications are from this package. `Fatfs-usd` creates a File Allocation Table (FAT) file system on an external Micro SD(uSD) card. `Animation` displays an animation from an external uSD card.

STM32F7Cube [12]: STM32F7Cube is a firmware package repositry for STM32F7 family of microcontroller. We run these applications on STM32F769NI-EVAL [10]. We use the Lightweight IP (LwIP) TCP `echo` application from this package. The `echo` runs a TCP Echo client on the microcontroller.

APPENDIX C
ANNOTATION CASE STUDIES:

In this section, we evaluate the impact of different user annotations.

FatFS-uSD (Shared Variables): EC requires explicit sharing of data across different compartments. If two compartments are sharing a variable, EC merges them into a single compartment. During the partitioning process, ECC logs the merger of different compartments due to shared variables. EC user can easily go through the logs to find the offending variables. They can either create local copies of the variables or mark the variable as shared variables to avoid the compartment being merged.

During our evaluation, we see this problem for nearly every firmware. For instance, for the FatFS-uSD application, initially, the device policy resulted in only four compartments. However, EC was able to find 15 distinct devices. Following up on the diagnostic message we found that each device driver used the clock configuration variable to initialize the device. We overcame this problem by explicitly marking the clock configuration variables as a shared variable, resulting in ten compartments.

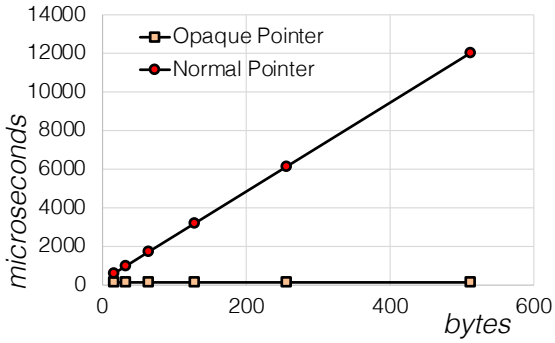

Fig. 13: Execution time of bridge functions for different sizes of buffers.

The annotation of the shared attribute could be automated, but we leave the final decision to the user. ECC emits all the shared variables with usage information. Users can apply the qualifier based on the policy requirements.

FreeRTOS (Opaque Pointers): During transitioning between different compartments, opaque pointers are passed without copying the pointed memory. Opaque pointers have both performance and security implications. Opaque pointers drastically decrease the overhead of an xcall. Figure 13 shows the time taken by an xcall using a normal pointer in comparison to an opaque pointer.

On the other hand, opaque pointers also reduce the exposure of data among compartments. RTOSs usually use handle objects to manipulate different kernel resources. For instance, FreeRTOS's task APIs use a task handle (`xTaskHandle`). The task creation APIs return the task handle to the caller, which is used to manipulate the task in the future. If the caller can modify the object associated with the object handle, a malicious caller could compromise the kernel threading operations. To this end, EC users can mark such object handles as opaque pointers, ensuring that only the RTOS can modify these objects. With an opaque task handle, a malicious caller trying to modify the task object would result in a memory protection fault.

ECK prints potential opaque pointers, such as void/char pointers, MMIO pointers (i.e. pointers pointing to IO memory), and pointers to large objects, in the diagnostic messages. Using these diagnostic messages, EC users can easily attribute the required pointers based on the policy requirements.

List library (Utility Functions): FreeRTOS implements a

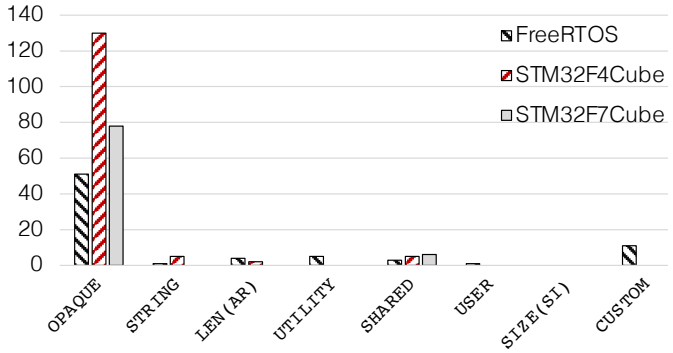

Fig. 14: EC annotation usage in different firmware packages.

list library (vList family of functions) that can be used for list management. This library is used extensively by different components in the system. We use the utility attribute to disassociate the library from other components in the system.

Note that the usage of utility attributes is not needed for most use cases. Opaque and shared attributes can usually result in a similar effect. However, it provides an elegant solution by minimizing the sharing between different compartments without code duplication. The usage for this attribute is usually determined per case and is triggered due to fault investigation during runtime.

Inter-Task Communication (Custom Bridges): ECC automatically instruments input firmware to cater for xcalls. However, during this instrumentation, ECC only copies data based on the type of the object. If the interface uses a char or a void pointer, ECC would only copy one byte of information. To this end, users can use annotations to point to the size information, which could be another input argument or in the case of strings the length of the string. Users can also fully implement the bridge function using the custom bridge annotation.

The Queue application shows a scenario where the size of the buffer cannot be inferred from the type or the function interface. More specifically, FreeRTOS's queue creation APIs takes the size of the queued item during queue creation as an input. The item size information is stored in the queue metadata. Due to this design, ECC cannot correctly copy the queued item, as the Send/Receive API does not contain the size information. To this end, we had to create custom bridges for the queue send and receive APIs. In the custom bridge, we copy the required amount of memory based on the size metadata field to correctly implement the queue operations.

Similar to utility functions, the usage for these attributes is determined per case and is triggered due to fault investigation during runtime.

Conclusion. As shown in the above case studies, to correctly compartmentalize firmware, EC provides several ways to control the compartments and their interactions. The usage of these annotations includes ensuring the correct operation of the firmware, application of security policies, and performance tuning of the firmware. The annotation-based guidance is one of the ECC key features and helps EC outperform where existing solutions struggle. Figure 14 shows the usage of annotations in the different projects used in our evaluation. We see the minimum number of annotation usage in FreeRTOS, as we specifcally design ECC extensions with RTOS design patterns under consideration. FreeRTOS only required 76 annotations, whereas STM32CubeF4 and STM32CubeF7 required 142 and 82 annotations respectively. The majority of the annotations are associated with data sharing and interface description, such as opaque pointers.