Low-Cost Privilege Separation with Compile Time Compartmentalization for Embedded Systems

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/SP46215.2023.10179388

***Abstract*—Embedded systems are pervasive and find various applications all around us. These systems run on low-power microcontrollers with real-time constraints. Developers often sacrifice security to meet these constraints by running the entire software stack with the same privilege. Existing work has utilized compartmentalization to mitigate the situation but suffers from a high overhead due to extensive runtime checking to achieve isolation between different compartments in the system, resulting in a rare adoption. In this paper, we present Compartmentalized Real-Time C (CRT-C), a low-cost compile-time compartmentalization mechanism for embedded systems to achieve privilege separation in a linear address space using specialized programming language dialects. Each programming dialect restricts the programming capabilities of a part of a program, formalizing different compartments within the program. CRT-C uses static analysis to identify various compartments in firmware and realizes the least privilege in the system by enforcing compartment-specific policies. We design and implement a new compiler to compile CRT-C to generate compartmentalized firmware that is ready to run on commodity embedded systems. We evaluate CRT-C with two Real-Time Operating Systems (RTOSs): FreeRTOS and Zephyr. Our evaluation shows that CRT-C can provide compartmentalization to embedded systems to thwart various attacks while incurring an average runtime overhead of 2.63% and memory overhead of 1.75%. CRT-C provides a practical solution to both retrofit legacy and secure new applications for embedded systems.**

I. Introduction

Embedded systems find applications in nearly every aspect of computing. Unlike general-purpose personal computers, embedded systems are resource-constrained and are generally used for a specific task. Often, multiple embedded systems are interfaced with each other to achieve some useful task, e.g., different sensors within a drone or hundreds of ECUs within a vehicle. Even though embedded systems are ubiquitous, the state of embedded systems security paints a rather worrisome picture. Developers often sacrifice security in favor of performance due to the low resource constraint. For instance, embedded systems are often programmed in a flat address space with no privilege separation to avoid the cost of context switching. As a result, any component in the system can freely access any resource in the system, e.g., a user-space task can access any kernel data structure or any device in the system. These poor security practices leave a large attack surface in embedded systems, as demonstrated by several attacks [5]–[8]. For instance, a compromised WiFi System-on-Chip (SoC) can be used to achieve a full device takeover of a mobile phone remotely [19], [20].

Compartmentalization [47], [50], [65] is one of the countermeasures for reducing the attack surface of a computing system. It divides a monolithic system into multiple small compartments, each of which is assigned a particular set of resources. Existing work has tried to achieve compartmentalization in embedded systems using various techniques. Minion [51] creates coarse-grained compartment views using program analysis and clustering algorithms with the help of a memory protection unit (MPU). M2MON [49] implements a Memory Mapped I/O (MMIO) reference monitor for embedded systems using a similar design as Minion. ACES [34] implements developer-guided compartments using compiler instrumentation and MPU.

Unfortunately, these systems suffer from the inevitable runtime overhead caused by instrumentation and monitoring, as shown in Table I. Furthermore, the memory protection hardware employed by embedded systems, such as the MPU, provides a limited number of protected regions and imposes restrictions on the size and alignments of protected memory regions [77], resulting in limited configurations and high memory overhead [34]. Due to the stringent resource constraint in embedded systems, these overheads are not acceptable for most applications, and thus these solutions were never able to catch wide adoption. Moreover, some of the solutions require an extensive amount of effort to port existing systems to their frameworks. For instance, these solutions mandate to move the system software stack from the privileged mode to the unprivileged mode manually, requiring a non-trivial effort.

Similarly, memory-safe programming languages [46], [55], [60], [67], such as embedded Rust, can help reduce the attack surface of embedded systems by proactively enforcing developers to write code in a type-safe fashion. While these solutions could result in lower overhead, adopting these solutions require rewriting the application in a completely new language, breaking compatibility with existing projects. More importantly, these languages do not provide any compartmentalization guarantees, as an unprivileged component can still access privileged resources, making them susceptible to privilege escalation attacks, etc.

In this paper, we present Compartmentalized Real-Time C (CRT-C), a program analysis and programming language-based compartmentalization technique for embedded systems. CRT-C combines the features of compartmentalization and language-based systems to achieve compartmentalization by identifying different compartments in the system at compile-

DOI 10.1109/SP46215.2023.00164

time and enforcing a specialized programming dialect based on the compartment type. Specifically, CRT-C utilizes program analysis to ensure that different compartments are isolated at the variable-level granularity. Each compartment in the system follows different policies to achieve the least privilege in the system. Instead of re-writing the whole application in a new programming language, we provide tools to convert legacy C code to CRT-C code in a semi-automatic fashion.

We design and implement the CRT-C compiler using LLVM and apply CRT-C to two RTOSs, FreeRTOS [14] and Zephyr [25], demonstrating how CRT-C can help retrofit existing systems. Our evaluation shows that CRT-C can effectively avoid memory corruption and privilege escalation issues in commodity systems. Furthermore, our evaluation shows that CRT-C can eliminate known CVEs in these RTOSs while incurring an average of 2.63% runtime overhead and 1.75% memory overhead. In summary, this paper makes the following contributions:

- We propose CRT-C, a novel compile-time compartmentalization technique for embedded systems using specialized programming dialects for separate compartments in the system. CRT-C uses static program analysis and an extended C type system to achieve this goal.
- We design and implement the CRT-C compiler that can identify different compartments in the system and restrict developers to the dedicated dialect of the corresponding compartment for memory safety and privilege separation. We provide tools to convert legacy C code to CRT-C code in a semi-automatic fashion.
- Using CRT-C, we compartmentalize two RTOSs to evaluate the efficacy and efficiency of CRT-C. Our evaluation shows that CRT-C can defend against several vulnerabilities and thwart known CVEs with compartmentalization while incurring an average of 2.63% runtime overhead, varying from 0.4% to 11.2%, and 1.75% memory overhead, varying from 0.9% to 5.8%.

To further research on this topic, we have released the source code for CRT-C.

II. Background

A. Real-Time Operating Systems (RTOS)

Real-Time Operating Systems are operating systems that schedule tasks in a deterministic manner. In an RTOS, given the set of runnable tasks in the system a user can pre-determine the schedule of tasks in the system. While RTOS schedules tasks deterministically, there can be some variability in the system called jitters, caused by various factors such as network congestion, interrupt handling, etc. To minimize jitters, RTOSs usually run in flat address space in embedded systems. In such a configuration, the tasks or threads in the system are functions that are directly linked with the RTOS code, both running inside the same privileged mode. Popular examples include FreeRTOS [14], Zephyr [25], NuttX [17], etc.

B. Compartmentalization

Compartmentalization is a design principle to divide a monolithic software stack into various compartments. Each compartment has its own set of resources, which can only be accessed via the compartment. Modern processors provide features such as memory management units and different execution modes to achieve compartmentalization in a system. We call compartmentalization using these hardware extensions, *Hardware-Based Compartmentalization*. Existing systems have employed various techniques to achieve compartmentalization in embedded systems as shown in Table I. However, these systems have not seen the same level of adoption because of the low-resource constraints and high overhead incurred due to hardware-based compartmentalization.

C. CheckedC

CheckedC [38] is a spatially safe dialect of C. Compared to existing safe dialects, CheckedC provides memory safety by adding extensions to C. Because of this design, every C program is also a CheckedC program. CheckedC extends C pointer types to provide three new safe pointer types that can be used to access memory safely. This includes `ptr`, `array_ptr` and `array_ptr_nt`. `ptr` type is used for pointers that do not allow any pointer arithmetic operations, whereas `array_ptr` and `array_ptr_nt` allow pointer arithmetic. Each pointer type contains sufficient checks to ensure that there are no buffer overruns or null dereferences at runtime. Additionally, CheckedC introduces the concept of checked and unchecked scopes. CheckedC restricts programmers to utilize only safe pointer types inside the checked scopes, whereas inside unchecked scopes the weak C type system can be utilized. By enforcing these restrictions on checked scopes, CheckedC guarantees that *"Code in checked scopes cannot be blamed for spatial safety violations"* [38]. For interoperability between checked scopes and unchecked scopes, CheckedC introduces the concept of bound-safe interfaces, with which programmers can do sufficient checking at the boundary of checked and unchecked scopes.

III. Security Model

Threat Model. We consider the classical threat model for operating systems, i.e., code running in user space, such as user threads, and code from 3rd-party vendors, such as device drivers, are untrusted and exposed to attacks directly. Attackers aim to compromise the whole system by exploiting vulnerabilities in the untrusted code, e.g., via memory corruption to achieve arbitrary code execution or privilege escalation within the system. As a result, attackers could gain access to systems resources that are not supposed to be accessed by default, such as interrupt handlers, schedulers, MPU configurations, secret keys, etc., due to the flat memory address space and running all the code within the same privileged mode. For instance, a sensor spoofing attack might exploit an integer overflow within the device driver, which further triggers a buffer overflow and leads to arbitrary code execution within the privilege mode. Ideally, even if the device driver was compromised, the exploitation

Project	Runtime Overhead	Memory Overhead	Mechanism	Granularity	TCB	Porting Effort
Minion [51]	6.13%	N/A (Fragmentation)	MPU	Thread Level	Small	Mode Switch
M2MON [49]	8.85%	5.02%	MPU	Device Level	Small	Mode Switch
ACES [34]	13%	61% (Fragmentation)	MPU +SFI	Programmable^{+}	Small	Mode Switch
CRT-C	**2.63%**	**1.75%**	**Language Based**	**Variable Level**	**Medium**	**Minimal***

TABLE I: Existing compartmentalization solutions: Mechanism describes the isolation technique used by the framework and Granularity shows the smallest compartment achieved by the framework. Each project is evaluated on a different variant of the STM32F4 microcontroller. ($^{+}$Granularity is limited by the MPU functionality) (* Firmware needs a compatibility layer (See Section V) to work with an RTOS.).

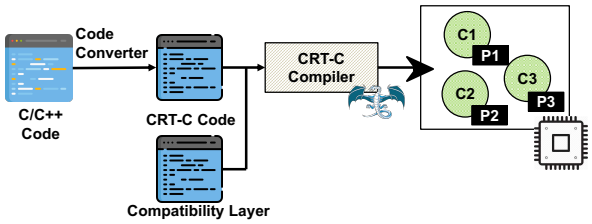


Fig. 1: CRT-C workflow. Existing code can be converted to CRT-C code using the compatibility layer to generate code in different compartments (C), which are subjected to specific policies (P).

should be confined within itself, e.g., its own compartment, without influencing other components within the system.

Trust Model. The system software or the RTOS, including the scheduler, interrupt handlers, etc., and the bootstrap code to start up the whole embedded system are considered part of our Trusted Computing Base (TCB). We anticipate the firmware (system software plus user-defined applications) to be statically linked, i.e., it does not contain any dynamically linked or shared libraries, which is a common practice for embedded systems. We assume the availability of firmware source code and a proper boot-up mechanism to ensure the boot-time integrity of the firmware. We do not assume the availability of hardware privilege separation (e.g., multiple MCU execution modes) or memory protection hardware (e.g., MPU), which further lowers the deployment requirements in the real world. In a nutshell, CRT-C creates isolated compartments in the firmware to confine the potential vulnerabilities and attacks within a compartment without influencing other compartments via isolating the code, data, and peripherals of all compartments.

Out-of-Scope. We do not aim to defend against local attacks, e.g., attacks within a compartment during the runtime, which have a number of different solutions, such as control flow integrity (CFI) and data flow integrity (DFI). Instead, our goal is to confine the attack within the compartment, preventing whole-system compromise. We do not consider undefined behaviors introduced by the embedded firmware, nor do we cover race conditions bugs within the firmware. A clear and even formally-verified specification might be needed to eliminate undefined behaviors. Recent work on detecting race conditions [53] could be integrated if needed. Finally, both side-channel [56], [61], [69] and physical [26], [64], [76] attacks are orthogonal to our threat model, and thus outside the scope of this paper.

IV. Overview

CRT-C is a language-based compartmentalization solution that uses program analysis to identify different compartments in the system. Each compartment is subjected to different policies which ensure strong isolation between different compartments. In case CRT-C is not able to guarantee isolation, it either instruments the program with runtime checks or points the user to the offending instruction during compilation.

To facilitate deployment, CRT-C provides tools to convert legacy programs to CRT-C programs. CRT-C includes an RTOS-specific compatibility layer that transparently redirects kernel APIs to safe interfaces. These interfaces ensure backward compatibility with existing applications, provide sanitization of input to the kernel, and maintain proper ownership transfer of memory resources. The compatibility layer can be used by the converted legacy code to safely use the legacy RTOS system API. These tools enable developers to convert legacy code to CRT-C code in a semi-automatic fashion. The CRT-C compiler parses through the code to enforce compartment-based policies in the system and generates the compartmentalized firmware which can be run on a micro-controller, as shown in Figure 1.

V. Design

We follow the classic system design principle of separating the mechanism from the policy. We first present the different compartments defined by CRT-C, followed by the policies that achieve isolation and privilege separation among different compartments. Lastly, we describe the mechanisms to achieve these policies.

A. Compartments

A typical RTOS consists of various compartments, such as the kernel, middleware, driver, etc., as shown in Figure 2. However, due to the weak isolation implemented in these systems, the compartment boundaries are usually blurred between those compartments. Developers frequently access objects across different compartments as the whole firmware has the same privilege level. As a result, a user task can access any device in the system. Instead, CRT-C uses program language constraints to create different compartments in the system. We first define the different compartments in the system created by CRT-C. CRT-C categorizes the compartments into three categories:

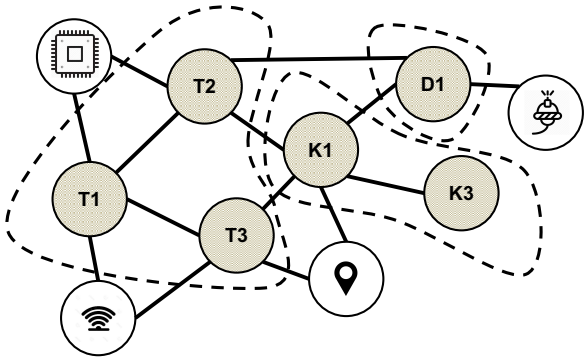


Fig. 2: A view of traditional RTOS: RTOS is divided into various compartments, such as kernel (K), driver (D), threads (T), etc. However, these compartments are weakly isolated from each other.

Kernel: We define the system software managing the system resources as the kernel compartment. This includes the scheduler, middleware, and low-level architecture-specific code.

Threads: Tasks or threads are the unit of execution in an RTOS. For our purposes, we use the term threads for both tasks and threads. The threads compartment consists of all the threads in the firmware. Within the thread compartment, each thread is contained within its compartment.

Device Drivers: The software stack responsible for managing devices constitutes the device driver compartment, within which every driver is kept within its compartment.

With well-defined compartments in the system, we assign different capabilities to each type of compartment by defining a different dialect for each compartment.

Kernel Dialect: As the kernel compartment consists of the RTOS code, we include this compartment inside our TCB. This is the highest privilege compartment in the system. Kernel dialect is the closest to the normal C code, except that kernel code cannot access IO directly. In other words, kernel code is not allowed to create or use hard-coded pointers for MMIO access.

Thread Dialect: In CRT-C, threads in the system have the least privilege in the system. Code in the thread compartment can only access objects explicitly assigned to the particular thread. Furthermore, the thread compartment is also not allowed to create or use hard-coded pointers.

Device Drivers Dialect: CRT-C gives the same privilege level to device drivers as threads. The only difference is that in this compartment, code can manipulate IO directly. Moreover, each device driver is associated with a physical device and can manipulate only that particular device directly. CRT-C provides mechanisms to associate a device with a driver at compile time. Based on this association, the driver code can create and access pointers to the IO regions that belong to the device associated with the driver.

B. Policies

With well-defined compartments, we design different policies that are enforced on each compartment. Using these policies we can instantiate the specialized dialects mentioned above. We derive our policies from classic compartmentalization models [57], [68] used by operating systems. These policies ensure attributing the least privilege to each compartment in the system and essentially provide a unique programming environment for each compartment in the system.

Kernel Policies: As we consider the kernel a part of our TCB, we do not enforce any restrictions on the kernel, which is free to manage resources in any manner it wants. However, we restrict the programming environment inside the kernel for IO accesses. More specifically, we enforce the following policy on the kernel compartment.

Policy 1: The kernel should not directly manipulate any devices manually.

This ensures that only device drivers can manipulate their respective devices and helps us establish a clear boundary between the kernel and the device driver compartments. As the kernel is still free to invoke any device driver API, this policy does not affect the privilege of the kernel compartment. Code manipulating devices directly is identified as a device driver.

Thread Policies: RTOS provides APIs to create threads, which are specific to each application and do not go through the same scrutiny as the kernel code. As a result, thread code is more susceptible to introducing vulnerabilities. We enforce the following policies on threads to ensure that they are the least privileged compartment in the system:

Policy 2: Thread compartments should be memory-safe.

This policy is vital to achieving isolation and privilege separation from other compartments in the system. This policy guarantees that buffer overruns [63], [66] are not able to escape any policies enforced at compile-time.

Policy 3: Thread compartments should only access objects assigned to the respective compartment.

This policy guarantees that no thread can access any objects that belong to other compartments, e.g., kernel. Furthermore, it ensures that no thread can access any objects that belong to any other thread in the system. Policy 2 and Policy 3 work together to ensure that no thread can access anything in the system that is not assigned to it explicitly.

Policy 4: The thread compartments should not directly manipulate any devices manually.

Similar to Policy 1, this policy ensures that any thread in the system is not allowed to access any device in the system directly. This policy, combined with Policy 3, ensures that only the device drivers have fine-grained control of all the devices in the system.

Device Driver Policies: Device drivers have the same level of privilege as threads, except that we allow them to manipulate I/O directly. We enforce the following rules on the device drivers.

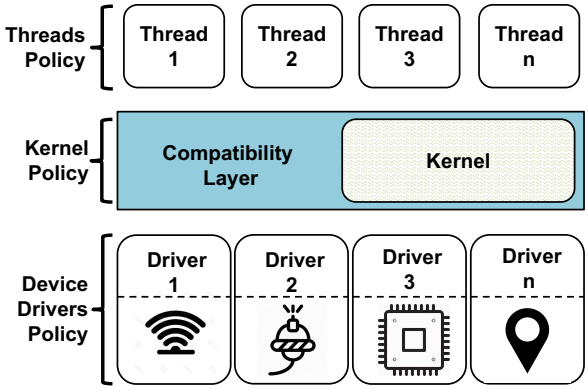


Fig. 3: CRT-C enforces different policies based on the compartment. Using these policies we can compartmentalize a firmware into different privilege levels.

Policy 5: Each device driver should be only allowed to manipulate devices assigned to them.
This policy is a relaxed version of Policy 1 and Policy 4. Device drivers are free to manipulate MMIO regions directly. However, in the absence of Policy 5, any device driver can manipulate any device in the system, thus breaking isolation. Instead, every device should be assigned to a single device driver owner to maintain the least privilege for the driver.

Policy 6: Device drivers should be memory-safe.
Similar to the Policy 2, this policy ensures that device drivers are not able to break the policies at runtime because of memory safety violations.

Figure 3 shows the overview of an embedded system with our policies enforced. CRT-C can create three different software privilege levels in the system, with each privilege level confined by its specific policies. Threads are only able to access the objects allocated to them while maintaining isolation among different threads. The kernel compartment is the only trusted compartment in the system without direct access to devices. The device driver compartments are in charge of all peripheral devices. Each device driver can only access the device it owns.

C. Mechanisms

With the six policies confining and isolating different compartments, we describe how CRT-C fulfills these policies leveraging static analyses and CheckedC.

Policy 1: The kernel should not directly manipulate any devices manually. To ensure policy 1, we need to discover all of the IO accesses done within the firmware. Embedded systems use MMIO to access peripherals. MMIO directly maps IO devices into the same linear address space as memory. As embedded systems usually do not employ virtual memory, MMIO accesses use hard-coded pointers to a specific memory address as shown in Listing 1. We call the pointers pointing to MMIO addresses *MMIO pointers*. To discover all of the IO accesses in the firmware, we need to find all the MMIO pointers in the system.

Listing 1: A hard-coded pointer to access MMIO.

```
1  *(volatile unsigned int *) 0xFE100000 = 0x0;
```

To distinguish MMIO pointers from normal pointers, we can refer back to Listing 1 to find the unique features of MMIO pointers. Firstly, we note that the MMIO pointers have the volatile qualifier and secondly, the pointer is created from a hard-coded literal. The volatile qualifier tells the compiler that the pointed memory may update between accesses and ensures that the compiler does not optimize away any operations with this pointer. The qualifier is necessary for the correct usage of MMIO pointers, as IO memory can be updated asynchronously from the device. We can use this qualifier as a heuristic to find all MMIO pointers in the firmware. However, in C/C++, it is valid to use the volatile qualifier on any variable. Such a pass would wrongly classify normal pointers with `volatile` qualifier as MMIO pointers.

Instead, we use the second heuristic, i.e., find all pointers that use a hard-coded base address. However, this heuristic also faces a problem as shown in Listing 2. The pointer `p` is created using a hard-coded address so it will be classified as an MMIO pointer based on the heuristic. However, the pointer `alias` is created using the pointer `p` and therefore will escape our analysis. Due to this *Pointer Propagation* problem, an MMIO pointer can alias with other pointers in the system.

Listing 2: An MMIO pointer without using a hard-coded address.

```
1  volatile unsigned int * p = 0xFE100000;
2  volatile unsigned int * alias =  p;
```

Solution: To overcome these challenges, we design an *MMIO Discovery* static analysis pass to find all MMIO pointers in the system by walking the use-def chains for each pointer. If a pointer is defined using a constant literal, the analysis registers that pointer as an MMIO pointer. Observing that MMIO pointers are rarely copied in embedded system firmware, we restrict any copying of such pointers and reject any firmware that copies MMIO pointers to avoid the pointer propagation problem. With this restriction, we only go through the definition sites of all permitted MMIO pointers to find all pointers in the system, instead of parsing through all the pointers uses in the code, to avoid the pointer aliasing problem.

The *MMIO Discovery* analysis takes in firmware code and returns a set of MMIO addresses used by the firmware. For Policy 1, we invoke this analysis with the kernel compartment, resulting in the set K_D, i.e., the set of all devices accessed by the kernel compartment. To enforce Policy 1, we ensure that the following condition holds:

$$K_D = \varnothing$$

i.e. the kernel compartment can neither create nor access any MMIO pointers. In other words, if some code accesses MMIO, it is treated as a device driver.

Policy 2: Thread compartments should be memory-safe. To enforce Policy 2, we need to identify all the threads in the

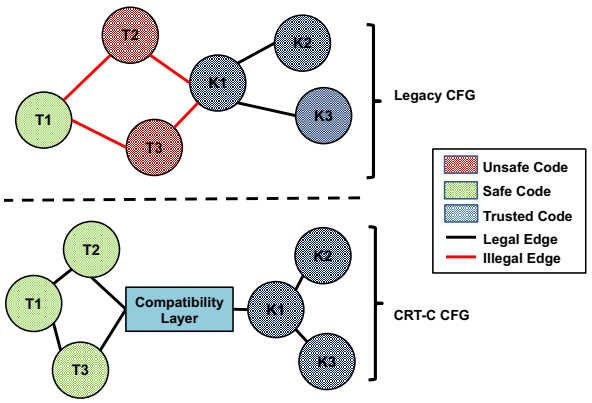


Fig. 4: Code conversion for thread compartment. T nodes are basic blocks in the thread compartment, while K nodes are kernel code. The code is incrementally converted to checked code.

system and ensure they are restricted to a memory-safe programming environment. However, each RTOS has its specific way of creating new threads. For instance, some RTOSs provide a thread creation API that takes in the entry point of the thread as an input argument, whereas some RTOSs allow creation of threads statically by placing the initialization information, including the thread entry point, in a linker section. The RTOS parses this information at boot time to create threads.

Solution: We design an RTOS-specific static analysis pass that identifies different threads in a firmware based on the thread creation APIs implemented by the RTOS. The *Thread Discovery* analysis walks the use-def chain on the thread creation API and the thread entry argument to extract all threads present in the firmware. Some RTOSs, like Zephyr, can create threads statically by placing thread definitions in a separate linker section. In this case, the *Thread Discovery* analysis parses the definition of these static threads to extract all of the threads in the firmware. The *Thread Discovery* analysis works on firmware level and returns T_A, the set of all threads in the system, i.e., $T_A = \bigcup_{i=1}^{n_{Threads}} t_i$ where t_i is an individual thread in the system, and $n_{Threads}$ is the total number of threads in the system.

Once we have identified all the threads in the system, we enforce the compiler to compile thread code in a memory-safe environment. More specifically, we enforce the code in T_A to be written in CheckedC's checked dialect. To achieve this we enforce the safety restriction on the thread entry function, as extracted by *Thread Discovery* and developers have to convert all of the called functions from the top-level function to CheckedC code as well, since in CheckedC checked code cannot call unchecked code.

To allow thread code to call into kernel code we provide a *Compatibility layer*, which is a bridge between thread code and kernel code providing a bounds-safe interface and checking any parameters passed between the two compartments. These safety restrictions propagate until the code calls into the *Compatibility layer* of the RTOS. Consequently, all functions in T_A are checked. This restriction ensures that all code belonging to T_A is memory-safe as CheckedC guarantees that code written in checked regions is free of any spatial memory-safety issues while maintaining maximum backward compatibility. Figure 4 shows the conversion of legacy C/C++ firmware to CRT-C. The CRT-C compiler catches all the illegal calls and restricts the programmer to use the correct compartmentalization required by the enforced policies.

Policy 3: Thread compartments should only access objects assigned to the respective compartment. As mentioned before, the implication of Policy 3 is two-fold. Firstly, no thread should be able to access kernel objects, a.k.a, *Kernel-Thread Isolation*. Secondly, no thread should be able to access objects belonging to other threads, a.k.a, *Inter-Thread Isolation*.

Solution: To achieve *Kernel-Thread Isolation*, we find all of the threads in the system using *Thread Discovery*. For each thread t_i, we use forward slicing [73] based on inter-procedural value flow analysis [70] to find the set of objects directly accessed by the thread. Furthermore, we use an off-the-shelf context and field sensitive *Points-to Analysis* [36], [37], [39] to find the set of objects accessed indirectly by each thread. We combine the objects obtained from these analyses to obtain the set of objects associated with the thread t_{O_i}. We perform a union of objects accessed by all the threads to find T_O, i.e., the set of all objects in the thread compartment.

$$T_O = \bigcup_{i=1}^{n_{TotalThreads}} t_{O_i}$$

Using a similar analysis, we find K_O, the set of objects accessed by the kernel. For *Kernel-Thread isolation*, we ensure the following condition holds:

$$T_O \cap K_O = \varnothing$$

i.e., we take an intersection to ensure that threads do not access any kernel objects. To achieve *Inter-Thread Isolation*, we have to guarantee that all of the threads exclusively access their own resources. Hence, we ensure that the following condition holds:

$$\forall i, j \in n | i \neq j, t_{O_i} \cap t_{O_j} = \varnothing$$

i.e., no thread in the system can access objects assigned to other threads.

Policy 4: Thread compartment should not directly manipulate any devices manually. We achieve Policy 4 using the same analysis as Policy 3. Using the *MMIO Discovery* analysis on the thread compartment obtained by the *Thread Discovery* analysis, we can get T_D, the set of all devices accessed by all the thread compartments. To guarantee Policy 4, we ensure that the following condition holds:

$$T_D = \varnothing$$

i.e., the thread compartment is not able to create or access MMIO pointers.

Policy 5: Each device driver should be only allowed to manipulate devices assigned to them. To ensure policy 5, we need to identify all device drivers in the system and ensure that these device drivers are only able to access a certain device. In other words, we need to 1) find all the MMIO regions are accessed by a device driver, and 2) associate MMIO regions with the device configuration of the platform. For 1), *MMIO Discovery* can be used to find the MMIO pointers in the system. However, it does not tell anything about the MMIO address the MMIO pointer is pointing to. For 2), there is no easy way to associate an MMIO region with a device during compilation, as embedded system development platforms have different memory and device configuration. This configuration is decided by the SoC vendor during design and the vendor supplies this information in the datasheet or the user manual for the particular platform.
Solution: To find MMIO regions accessed by the firmware we design an *MMIO Points-To Analysis*. As shown in Listing 1, the address pointed by the pointer is hard-coded. Hence, we can find the address pointed by the MMIO pointer. This analysis goes through the MMIO pointer definitions. Using constant propagation [72], we can find the base addresses of all MMIO pointers except where the base address is offset by a variable.

Listing 3: An expanded macro to access MMIO with a variable offset in an embedded firmware.

```
1  *(volatile uint *) (0xFE10 + offset) = 0x0;
```

Listing 3 shows an example of such code. Here we can statically determine the base address 0xFE100000, however, it is generally an intractable problem to determine the value of `offset` at compile time. Furthermore, the offset variable could be local or could be passed in as a function argument.

Listing 4: Constraining the offset variable to contain MMIO access within a fixed range.

```
1  if (offset < 0x10) {
2   *(volatile uint *) (0xFE10 + offset) = 0x0;
3  }
```

While determining the offset at compile time is generally intractable, in some cases, it is possible to find the value of the offset variable. Listing 4 shows one such pattern. The value of the variable `offset` is restricted by the check at line 1. Hence, `offset`'s value can range from `0` to `0x10` at line 3. Although we cannot tell the exact value of the MMIO access at line 3, we can conservatively determine that the MMIO access is in the range of `[0xFE10,0xFE20]`. Hence, we enforce the following restriction on firmwares:
"Policy 5a:All MMIO Pointer base addresses should be restricted within a fixed range at compile-time".

To find the range of MMIO accesses, we design a conservative inter-procedural *Value-Range Analysis* (VRA) [1], [43], that gives the range of a particular variable at compile-time as shown in Algorithm 1. First, we backtrack from the MMIO access instruction to the function entry point to find all the paths that result in the MMIO access. While walking the edge between basic blocks of a path, we collect the constraints required for reaching the successor. Using this information, we

Algorithm 1 Value Range Analysis

```
Result: range
Function getValueRange(value, known, instruction, ic): range
    ic ← ic + 1
     if value ∈ known then
        range = known[value]
    else if ic > IC_MAX then
        range = known[value]
    end
    else if type(op) == Argument then
        foreach callsite ∈ Callsites(func) do
            known ← getValueRange(value, callsite, known, ic)
        end
    end
    else
        foreach path ∈ func do
            if instruction ∈ path then
                /*Collect constraints that must
                satisfy to reach the instruction*/
                path.constraints ← collect(path, instruction)
                /*Get range of values based on the
                extracted constraints*/
                known ← solve(instruction, path.constraints)
                /*Propagate the newly found range
                information to known ranges*/
                foreach range ∈ Ranges do
                    foreach path.constraints do
                        propagate(range, known)
                    end
                end
            end
            foreach operand ∈ Instruction do
                known ← getValueRange(op, known, instruction, ic)
            end
            range ← solve(instruction, op, known)
        end
return
```

can find possible ranges for the target value and instruction. This process is done iteratively until either we have ranges for all the operands for the target instruction, or the iteration depth exceeds its budget. If we run into a function argument, we go through all call sites and construct a range to see if all call sites are invoked using a concrete range. Based on the target instruction, we calculate the final range. If the VRA can give a fixed range of the offset variable, we register all the possible MMIO regions that can be accessed with the returned range, with the compartment. Otherwise, CRT-C stops the compilation and points out the offending instruction to developers.

MMIO Region-Device Association: In order to associate the MMIO regions to actual devices CRT-C utilizes the ARM CMSIS System View Description (SVD) [21], which describes information about the platform, such as the base address, interrupt line assignments, and configuration for each device on a platform. We parse the SVD of the target platform to associate memory regions with different devices.
Driver Association: After *MMIO Region-Device Association*, we know the set of devices accessed by each compartment. In the case of two translation units accessing the same device, we need to associate a device with one of them. To this end, we design heuristic-based associations, including a frequency-based association or pattern matching. For frequency-based association, we regard the translation unit with the highest

Listing 5: Global variable sharing using attribute based tagging.

```
1  OWNER(taskA, taskB) QueueHandle_t x;
```

number of accesses as the owner and emit errors regarding the rest of the accesses, while pattern matching matches the device name with the directory hierarchy of the translation unit to establish an association.

Using *Driver Association*, we can get d_i, the set of devices accessed by the device driver dd_i and $n_{Drivers}$, the total number of drivers in the firmware. We ensure the following condition holds to enforce Policy 5 and 5a:

$$\forall i, j \in n_{Drivers} | i \neq j, d_i \bigcap d_j = \varnothing$$

i.e. for all drivers dd_i in the system, no more than one driver can manipulate the same device.

Policy 6: Device drivers should be memory-safe. To enforce this policy, we need to identify all drivers in the system. Fortunately, with *MMIO Points-To Analysis* and *Device Association Analysis* we have already identified all the drivers in the system, the compartment D:

$$D = \bigcup_{i=1}^{n_{Drivers}} dd_i$$

Similarly to Policy 2, we enforce that all device drivers must be written in a checked region.

D. Legacy systems adaption:

CRT-C enforces a strict set of rules on firmwares. However, in some cases, legacy firmwares may not be able to adhere to these rules. Furthermore, there exist some edge cases that cannot be captured by the mechanisms discussed. Therefore, to secure legacy systems, we provide the following extensions to CRT-C.

Object Sharing. Policy 3 strictly prohibits any sharing of objects. However, in real-world systems threads might need to share objects among themselves. Furthermore, the kernel might need to share objects with userspace threads. To this end, CRT-C provides support for both explicitly sharing objects among threads and between kernel and thread.

Kernel-Thread Sharing: CRT-C provides a new type qualifier `"userval"`. Kernel developers can annotate objects that are shared with user threads and device drivers to explicitly allow object sharing.

Inter-Thread Sharing: CRT-C enables Inter-Thread object sharing using two mechanisms: 1) Attribute-based sharing and 2) Q-Accessors. For 1), CRT-C provides `"OWNER"` attribute to annotate different owners. The attribute takes a comma-separated string of threads that can access the object. Developers can use the `"OWNER"` attribute to explicitly assign objects to threads. Listing 5 shows the usage of the attribute. The variable `x` is shared between `TaskA` and `TaskB`. Q-Accessors are compile-time generated artifacts that allow access to a shared object and are made the owner of the object, allowing CRT-C to prove the condition required for Policy 3. Q-Accessors can implement different sharing policies and by-default grant access to all threads in the system.

These sharing mechanisms ensure that all of the object sharing among compartments is explicit and any compartment cannot access any resource in the system that is not explicitly assigned to them.

Dynamic Object Tracking. CRT-C uses a conservative points-to analysis to track indirect accesses to objects at compile time. However, this analysis is not sound in general. For instance, dynamically allocated objects may escape this analysis as they are not created at compile time. To cater to this limitation, CRT-C adds runtime checks in the compatibility layer for ownership tracking. More specifically, CRT-C places all the objects in a separate section for each identified compartment, such as kernel, threads, etc. The compatibility layer checks all references passed to a compartment. If a resource is passed across a compartment, the compatibility layer throws a runtime error. Furthermore, as restricted compartments cannot create new objects within a checked scope, they use the compatibility layer to create new objects. The compatibility layer keeps track of each dynamically allocated object using an ownership map to ensure isolation among different compartments. As Policy 3 requires that any sharing between two threads should be explicit, the compatibility layer checks the resources passed and throws a runtime error if a resource is shared implicitly. We evaluate the soundness of these checks in Section VII.

External Devices. The MMIO discovery enables us to find all the devices interfaced using MMIO. However, embedded systems also use different external buses, such as SPI, I2C, etc. Code accessing a device behind an external bus can escape the MMIO discovery analysis, as the devices behind a bus can be accessed without any MMIO access using the bus controller driver. To this end, we design a key-based ownership model to associate devices behind the bus with a device driver. External device drivers have to ask for a device key from the bus controller driver before accessing a device during runtime, using `allocate_dev_bus` API. If the queried device is not owned by any device driver, the bus controller generates a fresh key for the device and returns the key to the external device driver. For future interactions with the external device, the external device drivers must provide the bus controller with the device key to gain access to the device. CRT-C uses `allocate_dev_bus` API to statically obtain users for devices behind an external bus, differentiating such device drivers from the rest of the RTOS.

Explicit Driver Association. CRT-C uses some heuristics to associate devices with different drivers in the system. If the association does not result in the desired association, we also enable users to explicitly associate devices with drivers. We extend SVD nodes to add a new custom attribute `"driver"` to the node which can explicitly associate a device with a device driver as shown in Listing 1. While parsing the SVD, if we find explicit device ownership using the `"driver"` attribute, we assign the ownership of the device without any

Listing 1 SVD peripheral node for a peripheral.

```
<peripheral>
    <name> UART0 </name>
    <driver> uart.c </driver> ...
```

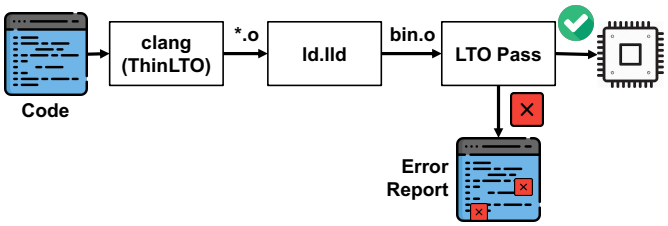


Fig. 5: Compilation and Analysis pipeline for CRT-C.

heuristics.

VI. Implementation

We implement CRT-C by extending the LLVM/clang [58] infrastructure. Figure 5 shows the implementation pipeline. We configure clang to emit code with ThinLTO [23] metadata, which ensures that the input sections are placed properly in the output binary. With the ThinLTO metadata, we link the files into the final firmware bitcode. We ensure that the firmware follows all the policies by implementing a link-time optimization pass. If the firmware passes the analyses, CRT-C emits the final binary object for the target, which can be run on the physical hardware. Otherwise, the compiler generates error messages regarding the offending instruction in the firmware.

clang. We use CheckedC clang [16] as the initial point for our implementation. We extend clang to understand the type information added by CRT-C, which is written to the bitcode file and can be processed using the LLVM LTO pass later. We extend CheckedC to transmit information about checked regions to the LLVM IR bitcode helping us to enforce the policies in the LTO pass.

LTO Pass. We modify the entire compilation pipeline to convey the type and metadata information from source code to bitcode and implement the major analyses in an LTO pass. This pass is responsible for Inter-Thread isolation, Kernel-Thread isolation, and device driver-related isolation. We utilize Static Value Flow (SVF) [70] for the points-to analysis.

Compatibility Layer. The compatibility layer consists of two parts: 1) implementation of bounds-safe interfaces for kernel APIs, and 2) a wrapper header [24] that redirects the legacy functions to call the bounds-safe interface using macro redirection. Using the compatibility layer and the 3C tool [3] we can port existing applications to CRT-C code in a semi-automatic manner. Listing 6 shows the "diff" between CRT-C and the normal C version of a FreeRTOS task. The only difference is in line 1, where the pointer type is modified. All the kernel APIs are called with the same prototype and transparently redirected to the bounds-safe version of the API using macro redirection. Table II shows the SLOC for CRT-C.

Component	SLOC
LTO Pass	2.3 K
Compatibility Layer (FreeRTOS)	382
Compatibility Layer (Zephyr)	182

TABLE II: SLOC for different components of CRT-C. The compatibility layer is specific to the RTOS.

Listing 6: Diff between safe and unsafe version of `QueueSendTask`.

```
1 >> static void QueueSendTask(ptr<void> pv)
1 << static void QueueSendTask(void * pv)
```

VII. Evaluation

We evaluate CRT-C on two RTOSs: FreeRTOS [30] and Zephyr [25] using the STM32F407G-DISCOVERY evaluation board. We first evaluate the security guarantees of CRT-C, followed by case studies of FreeRTOS and Zephyr. We also demonstrate the memory and performance overhead incurred by CRT-C. Lastly, we provide implementation overhead to gauge the effort to adapt CRT-C with legacy code.

A. Security Evaluation

We evaluate the security guarantees provided by CRT-C by examining various common vulnerabilities in embedded systems with the help of CRT-C.

Buffer Overflow: Memory-unsafe programming languages, like C/C++, allow direct manipulation of memory buffers using pointers. If a buffer is indexed without proper checking, an attacker can control the unchecked index to manipulate arbitrary program memory resulting in attacks such as malicious code execution. CRT-C uses CheckedC's bounds checking for avoiding buffer overflows as CRT-C enforces that all restricted compartment (i.e., thread and driver) code should be written in a checked scope. If a buffer overflow can be detected at compile time, the compiler throws a compilation error. For instance, consider the following code:

```
1  char buf checked[17]; buf[17] =0;
```

As the size of the buffer is 17 bytes, the access to `buf` byte 18 is outside the bounds of the buffer. When compiled, the compiler throws the following compile-time error:

```
1   error: out-of-bounds memory access
2          buf[17] =0;
```

If the index cannot be determined at compile-time, e.g.,

```
1  buf[i] =0;
```

CRT-C compiles the code and instruments the access with runtime checks. A runtime error is thrown if the buffer is accessed outside the bounds.

Privilege Escalation: An attacker can exploit vulnerabilities in the system to manipulate privileged resources used by the kernel to conduct privilege escalation attacks. As CRT-C considers the kernel compartment part of its TCB, we define privilege escalation as accessing privileged resources from the thread and the driver compartment. More specifically,

attackers try to access different resources across restricted compartments (thread and drivers). For instance, we pass a kernel resource, `pxCurrentTCB`, to a user thread. The FreeRTOS kernel uses this variable to keep track of the running thread. A malicious thread can modify the `pxCurrentTCB` and call `vTaskDelay` to suspend the rest of the threads in the system and unfairly monopolize CPU time. Fortunately, CRT-C disallows access to this variable using type safety and static analyses. More specifically, the compiler would reject unsafe type accessing the `pxCurrentTCB` variable, resulting in the following error:

```
error: global variable used in a checked
scope must be safe.
```

Even if the compatibility layer exposes the privileged resource as a safe variable or the kernel chooses to use a safe type for privileged resources, CRT-C compiler detects that different compartments are sharing resources implicitly and terminates the compilation by throwing the following error:

```
error:main accesses kernel var:x/tasks.c:343
```

The error points to the definition of the `pxCurrentTCB` variable. Lastly, if the kernel tries to pass its privileged resource as a thread argument, the compatibility layer checks throw a runtime error and halt execution. The compatibility layer and compile-time checks ensure that a compartment is not able to access any resource it does not own.

Race Condition: If the firmware accesses a global resource without synchronization primitives, an attacker can achieve malicious effects by exploiting race conditions vulnerabilities, such as Time-Of-Check-To-Time-Of-Use (TOCTTOU). CRT-C currently does not enforce any protection against race condition attacks. However, the Q-Accessor mechanism can implicitly be used to synchronize access to shared objects.

Format String: Format string attack uses a malicious format specifier string to access arbitrary memory locations during the interpretation of the format string specifiers to access and even write to arbitrary memory locations. To this end, CRT-C does not allow the usage of format specifier functions. More specifically, CRT-C does not allow the usage of variadic function, i.e. function that takes in a variable number of arguments. For instance:

```
printf("Test");
```

results in:

```
error: cannot use a variable arguments
function.
```

To overcome this limitation, we provide substitute functions in the compatibility layer to variadic functions. For instance, we provide `printChar`, `printInt`, etc. to provide similar functionality as printf.

Malformed Pointer Access: As mentioned earlier, memory-unsafe languages allow direct manipulation of memory using pointers. As the correct usage of the pointers is left to the programmer, a pointer could be uninitialized, null, or dangling. A malformed pointer could point to memory belonging to some other compartment and can result in arbitrary memory access. To this end, CRT-C mandates an initializer for the constrained compartment pointers. For instance, compiling the listed code:

```
ptr<int> tmp;
```

results in the following error:

```
error: variable 'tmp' must have initializer.
```

If we initialize the variable with a NULL initializer, the firmware passes the compile-time check and CRT-C instruments the code with a null check for runtime protection.

B. Case Studies:

FreeRTOS. FreeRTOS implements various data structures to help application development, such as `queues` [28] and `stream buffers` [29]. To allocate such a buffer, a user can request the size of the particular data structure. Listing 7 shows the API used for creating a `stream buffer`. The parameter `size`[1] is used to specify the size of the buffer. However, as disclosed in CVE-2021-31571 [10] and CVE-2021-31572 [11], if the size requested is large, it can wrap around before the allocation request as the API increases the allocation request to maintain the buffer metadata, as shown in Listing 8. Hence, an integer overflow will result in allocated memory less than the destination type. The usage of such a buffer can result in unexpected behavior such as a crash or arbitrary code execution.

Listing 7: Stream Buffer Creation API implemented in FreeRTOS.

```
StreamBufferHandle_t
    xStreamBufferCreate(size_t size,
                        size_t triggerlvl);
```

Listing 8: Allocation of buffer metadata is allocated with the buffer.

```
allocated = ( uint8_t * ) pvPortMalloc
  ( size + sizeof( StreamBuffer_t ) );
```

Using CRT-C, we move the `stream buffer` utility into the thread compartment forcing it to use the compatibility layer, which treats the arguments and return values as unsafe values and does the appropriate checking including NULL and buffer overrun checking. Once the buffer is allocated, it carries the bounds information with it, eliminating the two CVEs. A runtime assertion will be triggered if during a memory copy the destination and source sizes do not match.

Zephyr. Zephyr has a shell subsystem that implements a command-line interface to take inputs from the user. However, as disclosed in CVE-2017-14202 [9], the shell implementation does not protect against buffer overruns. It uses a history buffer to save past commands. `SHELL_HISTORY_DEFINE` is used to define the history buffer as shown in Listing 9. It takes in the name of the shell, the size of the commands (`block_size`) and the number of past commands saved (`block_count`)

[1]Variable renamed from `xBufferSizeBytes` for brevity.

as shown in Listing 9. In the Zephyr shell, the configuration parameter, `CONFIG_SHELL_CMD_BUFF_SIZE` is used to configure `block_size` and is used in the code for checking input size sanitization as well. However, in the buggy version, instead of `CONFIG_SHELL_CMD_BUFF_SIZE`, a hard-coded value of 128 is used to allocate the buffer as shown in Listing 10. As the `CONFIG_SHELL_CMD_BUFF_SIZE` is configurable, in case `CONFIG_SHELL_CMD_BUFF_SIZE` is set greater than 128, saving the current command to the history buffer causes a buffer overflow, allowing a serial or telnet-connected user to cause a crash, possibly with arbitrary code execution.

Listing 9: Macro used to define the shell history buffer in the Zephyr Shell.

```
1  #define SHELL_HISTORY_DEFINE(_name,
2                  block_size, block_count)
```

Listing 10: Invocation of `SHELL_HISTORY_DEFINE` in the buggy version of the Zephyr shell.

```
1  SHELL_HISTORY_DEFINE(_name, 128, 8)
```

Using CRT-C, we move the shell from the kernel to the thread component. Since this compartmentalized shell is subjected to the thread policies, all index-able buffers have bounds information. Furthermore, we explicitly check the results in the compatibility layer for any allocations. Lastly, the memcpy used in checked regions is bounds-aware and triggers a runtime exception if the copy operation results in a buffer overflow which avoids the root cause of CVE-2017-14202 by default.

C. Overhead Evaluation

We evaluate different types of overhead incurred by CRT-C in this section. Our evaluation dataset is categorized into math, device, and Inter-Process Communication (IPC) applications. We prefix the application name with the host RTOS, i.e., F for FreeRTOS and Z for Zephyr. Math applications include Compression (Z-COMP) and Integer (F-INT). Device applications include Console (Z-CNSL), Echo (Z-ECHO), Thread Flash (Z-THFL), FatFS-uSD (F-FAT), and Co-routine Flash (F-COFL). Lastly, IPC applications include Stream Buffer (F-SBUF), Queue Set (F-QSET), and Recursive Mutex (F-RECM). Further details about our dataset are given in Section B. We show the memory and performance overhead incurred by CRT-C on the evaluation dataset, followed by implementation overhead to evaluate the porting effort for existing applications.

Memory Overhead. We evaluate the memory overhead incurred by analyzing different sections in a binary and the number of dynamic objects utilized by the firmware. We establish the original C/C++ code as the baseline against CRT-C code. We present the overhead seen in code memory (Text), initialized data memory (Data), and uninitialized data memory (BSS).

Figure 6 shows the memory overhead for different applications. CRT-C outperforms existing systems (as shown in Table I) in terms of memory overhead by incurring only an average overhead of 1.75% for all applications. CRT-C imposes

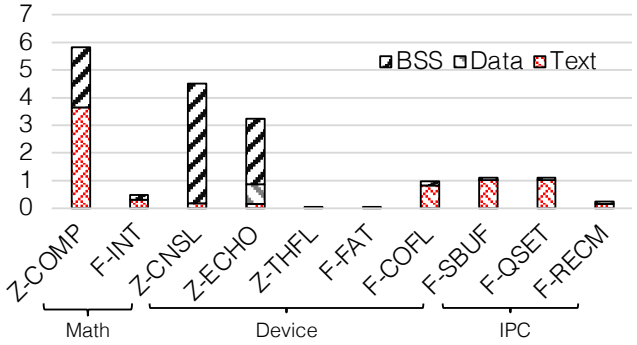


Fig. 6: Memory overhead incurred by using CRT-C. Y-axis shows the percentage increase, while X-axis shows the application.

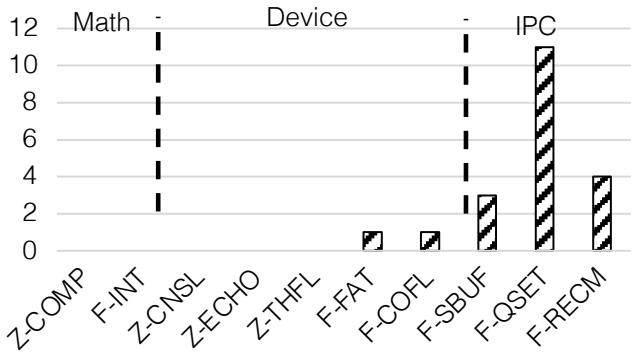
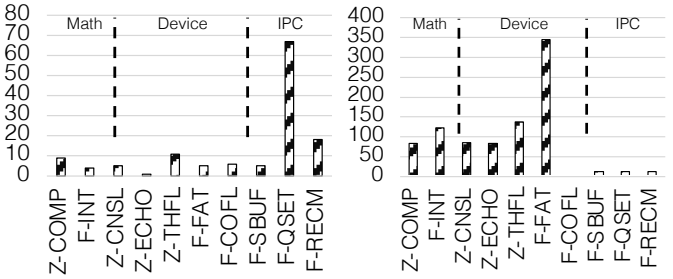


Fig. 7: Number of dynamic objects used by each application. Y-axis shows the number of objects, while X-axis shows the application.

minimum overhead by avoiding the usage of hardware memory protection, such as MPU, resulting in zero fragmentation overhead. The main overhead is seen in the text section and BSS section caused by the runtime instrumentation (bounds checking) and the ownership tracking data structures in the compatibility layer. The modified external bus controller drivers also incur overhead in the BSS section to keep track of external device ownership. For math applications, compression demo incurs the highest overhead of 5.81%. For device applications, console incurs the highest overhead of 4.5%. For IPC applications, both Recursive Mutex and Stream Buffer incur an overhead of 1.1%.

Figure 7 shows the number of dynamic objects used by each application. For each dynamic object, we create a wrapper object that is passed to the calling function. Therefore, the wrapper object is the overhead incurred for dynamic objects. On ARMv7-M, the size of the wrapper object is 12 bytes. Math applications do not use any dynamic objects. For Device applications, the FatFS-uSD and Coroutine Flash applications utilize dynamic objects. IPC applications make heavy use of dynamic objects with the Queue Set application using 11 dynamic objects, resulting in 132 bytes overhead. Note that, Zephyr allows the static creation of objects, such as threads, and most applications default to using this mechanism. Due to this reason, zephyr applications show zero dynamic objects overhead.



(a) Number of Compatibility Layer calls.

(b) Number of runtime checks.

Fig. 8: Performance overhead sources for evaluated application.

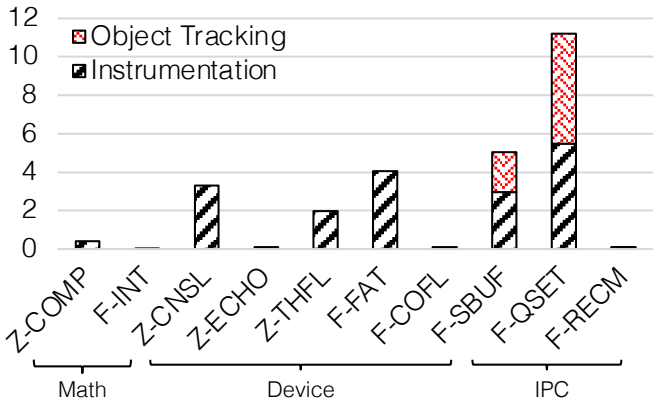


Fig. 9: Execution overhead incurred by using CRT-C. Y-axis shows the percentage increase, while X-axis shows the application.

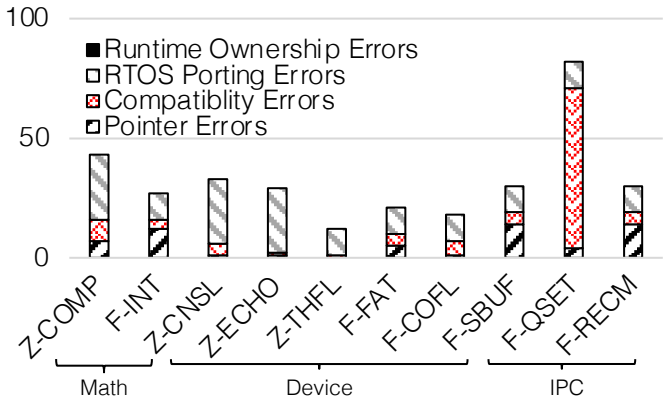


Fig. 10: Compilation resolutions. Y-axis shows the number of compilation resolutions, while X-axis shows the application.

Performance Overhead. We evaluate the performance overhead incurred by CRT-C by comparing the time taken to establish the C/C++ native code as the baseline against CRT-C code for different applications. For both Zephyr and FreeRTOS, we used ARM Systick [22] timer to benchmark the execution times. For Zephyr, existing timing functions [18] were used, whereas, for FreeRTOS, we implemented our benchmarking framework that works similarly to Zephyr timing functions.

CRT-C's performance overhead is mainly incurred by the runtime checks instrumented by CRT-C, the checking in the compatibility layer for input sanitization and object tracking. To this end, we statically count the number of calls to the compatibility layer and the number of runtime checks added by CRT-C, as shown in Figure 8. FatFS-uSD had the largest amount of instrumentation, i.e., 134 checks, as it extensively uses pointers. IPC applications issue the highest number of compatibility layer calls, with Queue Set making compatibility layer calls.

We break down the performance overhead into object tracking and instrumentation overhead. The object tracking overhead is the time spent by CRT-C for tracking ownership of dynamically created objects, whereas instrumentation overhead stems from CRT-C bounds checking and calls to the compatibility layer. Figure 9 shows the runtime overhead obtained using the benchmarking framework. The IPC applications incur the largest runtime overhead, as they extensively use the compatibility layer and dynamic objects. Among the IPC applications, the Queue Set application incurs the highest overhead, where 51% of the overhead is contributed from object tracking, as the Queue Set application uses the highest number of dynamic objects as shown in Figure 7. Device applications show a moderate overhead. FatFS-uSD incurs the highest overhead in this set. While the FatFS-uSD does not extensively use dynamic objects, Figure 8 shows that this application has the highest number of runtime checks. For math applications, we see a low overhead. Based on our performance overhead evaluation, to achieve a low overhead, applications should use static objects and minimize the usage of pointers and the compatibility layer.

Implementation Overhead. CRT-C provides a semi-automatic mechanism to compartmentalize legacy embedded systems. Most of the work is automated using static analyses. If user intervention is required, CRT-C throws an error and points to the violating instructions to help users adapt existing firmware to CRT-C. These errors can be categorized into three main categories: RTOS Porting Errors, Compatibility Errors, and Pointer Errors. RTOS porting errors are raised during an initial port of an RTOS, such as unsafe drivers, illegal usage of MMIO pointers, etc. Compatibility errors arise when applications try to directly call kernel APIs. Lastly, pointer errors raise when safe code tries to use raw/unsafe pointers. Compatibility and pointer errors are dependent on applications, whereas RTOS porting errors depend on the RTOS only.

Figure 10 quantitively estimates the implementation overhead for each application. Queue Set application raised the highest number of errors, as it made several calls to the kernel. Furthermore, porting Zephyr raised more errors compared to FreeRTOS. Lastly, we did not see any runtime ownership errors for the evaluated applications. Based on our experience, we conjecture that CRT-C users can port an existing RTOS and application to CRT-C in 1-3 man-days. Furthermore, with each application, this time should go down, as newer applications may use services from already ported applications. We describe case studies about the detailed porting process in Section C.

VIII. Discussion

Implications of CRT-C Assumptions: As mentioned in Section III, CRT-C makes some assumptions about the input firmware. Without those assumptions, an attacker might be able to bypass CRT-C guarantees. For instance, an undefined behavior such as integer overflow can result in unsoundness of the CRT-C analyses. On the other hand, the hardware-enforced compartmentalization systems, listed in Table I, will most likely result in a runtime protection error [2]. Fortunately, each attack vector can be mitigated using existing defenses. The defenses to each of our assumptions are orthogonal to this work. For instance, undefined behavior sanitizer (UBSAN) can be used to find integer/floating point overflows in the firmware. Similarly, existing tools [53] can be used to find data races in firmwares.

CRT-C adaption: We design CRT-C with adaption to legacy code in mind. As shown in Section VII, CRT-C adaption is categorized into RTOS-specific and application-specific porting. The RTOS-specific porting includes porting the device drivers. Depending on the application, CRT-C users can incrementally port the device drivers for an RTOS. Similarly, application-specific porting requires, 1) converting the code to the safe dialect, and 2) providing safe interfaces for the RTOS. For 1) the 3C tool [3] can assist in the conversion of unsafe code to safe code. For 2), users can port RTOS interfaces incrementally, depending on the application. During our evaluation, we noticed a significant amount of boilerplate code for input arguments sanitization in the safe interface, which can be generated automatically.

Minimizing the System TCB: As CRT-C considers the kernel inside the TCB, the monolithic nature of existing RTOSs can bloat the code inside the TCB. The situation can be mitigated using a microkernel design for RTOSs. However, commodity RTOS are often monolithic in design due to the high cost of privilege separation incurred by using compartmentalization at the micro-kernel level. For CRT-C, we move drivers and threads out of the TCB, essentially reflecting the microkernel design for RTOS.

Temporal Memory Safety: CRT-C ensures spatial memory safety in the system but does not protect again temporal memory safety. To this end, we recommend using no-free dynamic allocators, i.e., once a memory region is allocated it cannot be released or freed. Especially for safety certification and automotive coding standards, this is the norm, as dynamic memory allocation is the antithesis of determinism.

Direct Memory Access (DMA): DMA directly accesses memory without CPU intervention. While DMA is programmed using MMIO registers, the memory transfer carried out by the DMA controller is external to the CPU core and escapes the CPU's memory protection mechanisms. While CRT-C can explicitly control the ownership of the DMA controller, the DMA driver can access any memory belonging to any compartment in the system using DMA. To this end, the DMA driver can take advantage of the ownership tracking from the compatibility layer to ensure that the DMA controller does not transfer memory across compartments. Furthermore, DMA controllers can be exploited by malicious peripherals to initiate memory access across different compartments. To mitigate attacks from malicious peripherals, an IOMMU is needed.

Extending Compartments: Currently CRT-C only allows pre-defined compartments in the system. We have modeled the existing RTOS compartments with the least privilege as the security goal for CRT-C. While these compartments should be enough to model all required compartments in the system, we can bring more flexibility to our design by dynamically adding a new type of compartment and programming the allowed capabilities for it. We plan to explore this direction in future work.

[2]Attackers might be able to carry out race conditions vulnerabilities, such as Time-Of-Check-To-Time-Of-Use (TOCTTOU), without a memory fault.

IX. Related Work

Embdded Security Frameworks: General purposes systems employ various techniques, such as architecture-based countermeasures [48], safe languages [46], [62], static [44], [74] and dynamic code checkers [27], [32], [45], [52], [75], sandboxing [40], [41], etc., to reduce the attack surface of the system. However, embedded systems present a different set of challenges, such as a lack of a memory management unit (MMU), fewer execution modes, and a low-power microcontroller. To mitigate this situation, ACES [34] compartmentalizes an embedded system using an LLVM pass based on user specification. EPOXY [35] identifies privilege operations in firmware and only runs those operations in the privilege mode. M2MON [49] creates a reference monitor for embedded systems to control access to IO memory at runtime. Wang et. al, [71] uses Minix to provide privilege separation with the help of TPM [54] to verify external agents. Compared to CRT-C, these solutions are runtime solutions and incur a large overhead, as shown in Table I.

Language-based Systems: Language-based isolation has been used in past to secure different systems. Singularity OS [42] uses language-based protection instead of traditional hardware security features to create processes in a flat address space. Every process goes through static analysis before it can be run with Singularity OS. Tock OS [59] implements an RTOS using rust [55]. In userspace, only safe rust is allowed. SafeTCL [60] creates a language-based restriction on TCL scripts. They use multiple interpreters based on the capabilities attributed to a script.

Memory Safety: There has been existing work to mitigate the memory safety issues in C. Cyclone [46] is a safe dialect for C that adds fat pointers and never-null pointers in the system. CCured [62] provides a whole type system for pointers, including safe pointers, sequence pointers, and dynamic pointers. Each pointer can have different kinds of operations based on the pointer type. CCured uses garbage collection for temporal memory safety. RefinedC [67] uses type refinements that can impose restrictions on the type. While existing work solves the problem of memory safety in C/C++, they do not guarantee privilege separation in the system. CRT-C builds upon existing memory-safe language

techniques while providing whole-system compartmentalization by adding awareness about privilege in the language.

X. Conclusion

In this paper, we present CRT-C, a low-cost compile-time compartmentalization mechanism for embedded systems to achieve privilege separation without hardware memory protection using specialized programming language dialects. We evaluate CRT-C on two real-world RTOSs, FreeRTOS and Zephyr, and show that CRT-C incurs an average of 2.63% runtime overhead and 1.75% average memory overhead which makes CRT-C a practical solution to secure real-world embedded systems firmware.

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] Andrewmacleod/ranger - gcc wiki. https://gcc.gnu.org/wiki/AndrewMacLeod/Ranger. (Accessed on 10/14/2022).

[2] Balanced red/black tree — zephyr project documentation. https://docs.zephyrproject.org/3.0.0/reference/data_structures/rbtree.html. (Accessed on 09/28/2022).

[3] . checkedc-clang/readme.md at master · microsoft/checkedc-clang. https://github.com/microsoft/checkedc-clang/blob/master/clang/tools/3c/README.md. (Accessed on 03/21/2022).

[4] Cmsis – arm developer. https://developer.arm.com/tools-and-software/embedded/cmsis. (Accessed on 09/28/2022).

[5] . Cve - cve-2002-2041. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-2041. (Accessed on 10/09/2021).

[6] . Cve - cve-2002-2120. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-2120. (Accessed on 10/09/2021).

[7] . Cve - cve-2006-0621. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0621. (Accessed on 10/09/2021).

[8] . Cve - cve-2013-2688. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2013-2688. (Accessed on 10/09/2021).

[9] . Cve - cve-2017-14202. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-14202. (Accessed on 10/08/2021).

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

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

[12] Devicetree. https://www.devicetree.org/. (Accessed on 09/28/2022).

[13] Freertos co-routines. https://www.freertos.org/croutine.html. (Accessed on 10/17/2022).

[14] . Freertos real time kernel (rtos) / bugs / #174 freertos+io circular buffer overflow. https://sourceforge.net/p/freertos/bugs/174/. (Accessed on 10/08/2021).

[15] Github - 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 10/14/2022).

[16] . github.com. https://github.com/microsoft/checkedc-clang. (Accessed on 09/30/2021).

[17] . Home. https://nuttx.apache.org/. (Accessed on 03/19/2022).

[18] Kernel timing — zephyr project documentation. https://docs.zephyrproject.org/3.1.0/kernel/services/timing/clocks.html#. (Accessed on 10/17/2022).

[19] . 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).

[20] . 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).

[21] . System view description. https://www.keil.com/pack/doc/CMSIS/SVD/html/index.html. (Accessed on 09/27/2021).

[22] . Systick timer (systick). https://www.keil.com/pack/doc/CMSIS/Core/html/group__SysTick__gr.html. (Accessed on 10/08/2021).

[23] . Thinlto — clang 13 documentation. https://clang.llvm.org/docs/ThinLTO.html. (Accessed on 09/30/2021).

[24] . Wrapper headers (the c preprocessor). https://gcc.gnu.org/onlinedocs/cpp/Wrapper-Headers.html. (Accessed on 10/08/2021).

[25] . Zephyr project - zephyr project. https://www.zephyrproject.org/. (Accessed on 09/29/2021).

[26] Chuadhry Mujeeb Ahmed, Jianying Zhou, and Aditya P Mathur. Noise matters: Using sensor and process noise fingerprint to detect stealthy cyber attacks and authenticate sensors in cps. In *Proceedings of the 34th Annual Computer Security Applications Conference*, pages 566–581, 2018.

[27] Ali Almossawi, Kelvin Lim, and Tanmay Sinha. Analysis tool evaluation: Coverity prevent. *Pittsburgh, PA: Carnegie Mellon University*, pages 7–11, 2006.

[28] Richard Barry. Freertos - freertos queue api functions, including source code functions to create queues, send messages on queues, receive messages on queues, peek queues, use queues in interrupts. https://www.freertos.org/a00018.html. (Accessed on 10/13/2021).

[29] Richard Barry. Freertos stream buffers - circular buffers. https://www.freertos.org/RTOS-stream-buffer-example.html. (Accessed on 10/13/2021).

[30] Richard Barry et al. Freertos. *Internet, Oct*, 2008.

[31] Matěj Bartík, Sven Ubik, and Pavel Kubalik. Lz4 compression algorithm on fpga. In *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, pages 179–182. IEEE, 2015.

[32] Ulrich Bayer, Andreas Moser, Christopher Kruegel, and Engin Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[33] Wasim Ahmad Bhat and SMK Quadri. Performance augmentation of a fat filesystem by a hybrid storage system. In *Advanced Computing, Networking and Informatics-Volume 2*, pages 489–498. Springer, 2014.

[34] 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.

[35] 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.

[36] Saumya Debray, Robert Muth, and Matthew Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–24, 1998.

[37] Amer Diwan, Kathryn S McKinley, and J Eliot B Moss. Type-based alias analysis. *ACM Sigplan Notices*, 33(5):106–117, 1998.

[38] Archibald Samuel Elliott, Andrew Ruef, Michael Hicks, and David Tarditi. Checked c: Making c safe by extension. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 53–60. IEEE, 2018.

[39] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994.

[40] Ulfar Erlingsson and Fred B Schneider. Sasi enforcement of security policies: A retrospective. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 287–295. IEEE, 2000.

[41] David Evans and Andrew Twyman. Flexible policy-directed code safety. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy (Cat. No. 99CB36344)*, pages 32–45. IEEE, 1999.

[42] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. Language support for fast and reliable message-based communication in singularity os. In *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 177–190, 2006.

[43] William H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on software engineering*, (3):243–250, 1977.

[44] 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.

[45] 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.

[46] Trevor Jim, J Gregory Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: a safe dialect of c. In *USENIX Annual Technical Conference, General Track*, pages 275–288, 2002.

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

[48] Gaurav S Kc, Angelos D Keromytis, and Vassilis Prevelakis. Countering code-injection attacks with instruction-set randomization. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 272–280, 2003.

[49] 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.

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

[51] 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.

[52] 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.

[53] Taegyu Kim, Vireshwar Kumar, Junghwan Rhee, Jizhou Chen, Kyungtae Kim, Chung Hwan Kim, Dongyan Xu, and Dave Jing Tian. {PASAN}: Detecting peripheral access concurrency bugs within {Bare-Metal} embedded applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 249–266, 2021.

[54] Steven L Kinney. *Trusted platform module basics: using TPM in embedded systems*. Elsevier, 2006.

[55] Steve Klabnik and Carol Nichols. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[56] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1–19. IEEE, 2019.

[57] Maxwell N Krohn, Petros Efstathopoulos, Cliff Frey, M Frans Kaashoek, Eddie Kohler, David Mazieres, Robert Tappan Morris, Michelle Osborne, Steve VanDeBogart, and David Ziegler. Make least privilege a right (not a privilege). In *HotOS*, 2005.

[58] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE, 2004.

[59] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Shane Leonard, Pat Pannuto, Prabal Dutta, and Philip Levis. The tock embedded operating system. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–2, 2017.

[60] Jacob Y Levy, Laurent Demailly, John K Ousterhout, and Brent B Welch. The safe-tcl security model. In *USENIX Annual Technical Conference*, 1998.

[61] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, et al. Meltdown: Reading kernel memory from user space. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 973–990, 2018.

[62] George C Necula, Scott McPeak, and Westley Weimer. Ccured: Type-safe retrofitting of legacy code. In *Proceedings of the 29th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 128–139, 2002.

[63] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[64] Fabio Pasqualetti, Florian Dörfler, and Francesco Bullo. Cyber-physical attacks in power networks: Models, fundamental limitations and monitor design. In *2011 50th IEEE Conference on Decision and Control and European Control Conference*, pages 2195–2201. IEEE, 2011.

[65] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, 2003.

[66] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-oriented programming: Systems, languages, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 15(1):1–34, 2012.

[67] Michael Sammler, Rodolphe Lepigre, Robbert Krebbers, Kayvan Memarian, Derek Dreyer, and Deepak Garg. Refinedc: automating the foundational verification of c code with refined ownership types. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 158–174, 2021.

[68] Fred B Schneider. Least privilege and more [computer security]. *IEEE Security & Privacy*, 1(5):55–59, 2003.

[69] François-Xavier Standaert. Introduction to side-channel attacks. In *Secure integrated circuits and systems*, pages 27–42. Springer, 2010.

[70] 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.

[71] Xiaolong Wang, Masaaki Mizuno, Mitch Neilsen, Xinming Ou, S Raj Rajagopalan, Will G Boldwin, and Bryan Phillips. Secure rtos architecture for building automation. In *Proceedings of the First ACM Workshop on Cyber-Physical Systems-Security and/or PrivaCy*, pages 79–90, 2015.

[72] Mark N Wegman and F Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(2):181–210, 1991.

[73] Mark Weiser. Program slicing. *IEEE Transactions on software engineering*, (4):352–357, 1984.

[74] 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.

[75] 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.

[76] Mark Yampolskiy, Peter Horvath, Xenofon D Koutsoukos, Yuan Xue, and Janos Sztipanovits. Taxonomy for description of cross-domain attacks on cps. In *Proceedings of the 2nd ACM international conference on High confidence networked systems*, pages 135–142, 2013.

[77] Wei Zhou, Le Guan, Peng Liu, and Yuqing Zhang. Good motive but bad design: Why arm mpu has become an outcast in embedded systems. *arXiv preprint arXiv:1908.03638*, 2019.

APPENDIX A
CRT-C RESTRICTIONS ON CHECKEDC.

During our security evaluation, we also uncovered a bug in CheckedC implementation that allowed dynamic casting to incompatible types within the checked scope violating spatial memory safety. The following listing shows a minimal code to highlight the vulnerability.

```
1  typedef struct {
2        ptr<int> a;
3  } STRUCT;
4  void break(void) {
5     char temp checked[100];
```

```
6       ptr<STRUCT> s  = NULL;
7       s  = dynamic_bounds_cast<ptr<STRUCT>>
8                                (&temp[20]);
9       int a;
10      s->a = (ptr<int>)&a;
11      temp[20] = 0xAB;
12      temp[21] = 0xCD;
13      temp[22] = 0xEF;
14  }
```

On line 5, the code allocates a 100-byte large character buffer, `temp`, on the stack. Next, at line 7, the code casts `temp` to a structure `s` with a pointer field. However, the compiler does not complain about the incompatible typecast. As a result, the attacker has legitimate access to the temp buffer and can legally dereference the pointer field `a` in structure `s`. Therefore, attackers can perform arbitrary memory access within a checked scope by modifying `temp` and accessing the pointer field. Unlike CheckedC, we have restricted dynamic casting within CRT-C. The issue has been reported and confirmed by CheckedC developers.

Appendix B
Evaluation Dataset.

Our evaluation dataset consists of standard applications shipped with FreeRTOS and Zephyr RTOS. Furthermore, we also port some applications from STM32CubeF4 [15] firmware package to FreeRTOS. To ensure that our dataset is reflective of real-world applications, we pick applications that cover different facets of real-world applications. To this end, we categorize our dataset into three categories:

Math: These applications implement different mathematical operations and are generally CPU-intensive applications, including Compression and Integer. Compression implements the LZ4 [31] compression algorithm, whereas the Integer demo conducts a series of arithmetic operations.

Device: These applications interface with different peripheral devices, and in general, are IO-intensive applications. Echo implements a loopback on the Universal Asynchronous Receiver/Transmitter(UART) device. Similarly, Console implements a simple console on the UART device. Thread Flash and Coroutine Flash are General Purpose IO (GPIO) applications, implementing an LED flashing application. Thread Flash uses threads, whereas Coroutine Flash uses lightweight tasks called *Coroutines* [13] to achieve the same goal. Lastly, FatFS-uSD implements a File Allocation Table (FAT) [33] filesystem on an external MicroSD (uSD) card, interfaced using a Serial Peripheral Interface (SPI) bus.

IPC: These applications demonstrate different inter-process communication primitives available on commodity RTOS. StreamBuffer demonstrates a single-writer single-reader queue to communicate between two tasks. Queue Set uses RTOS queues to communicate between two tasks. Lastly, recursive mutex utilizes mutex to synchronize among different threads.

We also present some insights about the firmware of each application, as shown in Figure 11. Most applications show similar statistics, whereas FatFS-uSD and Compression applications show a much higher pointer usage. For presentation purposes, we suffix the application name with the underlying RTOS. Applications with the suffix (Z) are Zephyr applications, whereas the suffix (F) is used for FreeRTOS applications.

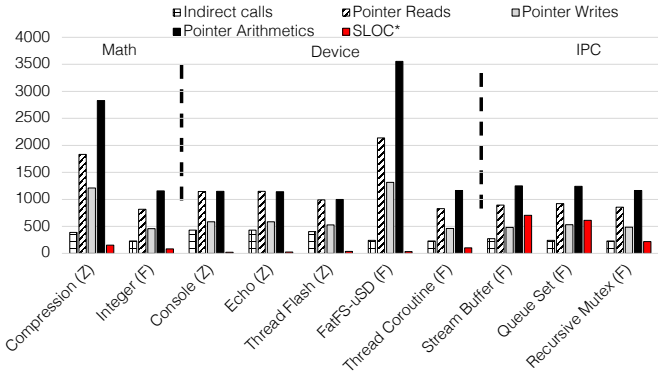


Fig. 11: Various statistics about the firmware of evaluated application. Y-axis shows the metric value, while X-axis shows the application (*SLOC is only for the application/thread code.).

Appendix C
Porting legacy firmware to CRT-C:

CRT-C provides a semi-automatic mechanism to compartmentalize legacy embedded systems. Most of the work is automated using static analyses. If user intervention is required, CRT-C points to the violating instructions to help users adapt existing firmware to CRT-C. In this section, we highlight a few case studies to demonstrate the porting effort for adapting CRT-C for an existing firmware. Once an existing RTOS is ported to CRT-C, future applications can use the modified RTOS. Therefore, we split the porting overhead into two categories: 1) RTOS porting, and 2) Application porting.

RTOS Porting - FreeRTOS. In this case study, we demonstrate the porting process for FreeRTOS. We start with `CORTEX_M4F_STM32F407ZG-SK` port and modify the code to make it compatible with *clang*. We consider this the starting point for our porting process. CRT-C is just a drop-in replacement for clang. CRT-C emits several errors listed below:

Unbounded I/O Errors: The first set of errors is regarding unbounded MMIO accesses. In FreeRTOS, we encountered four unbounded MMIO access errors. These errors are thrown when the VRA analysis does not converge. The first two errors are thrown from the Nested Vectored Interrupt Controller (NVIC) and External Interrupt Controller (EXTI). The NVIC driver uses an IRQ number as an index to offset the MMIO region to write the NVIC configuration register as shown below[3]

```
1  void NVIC_ClearPendingIRQ(IRQn_Type IRQn) {
2    NVIC->ICPR[IRQn] = PENDING_VALUE(IRQn);
3  }
```

We add a check on the IRQ number to keep it within a fixed range, resolving the unbounded IO error. Similarly, the EXTI driver uses the configuration parameter to offset into the IO region based as shown below.

[3]Code snippets are modified for brevity

```
1  tmp += EXTI_InitStruct->EXTI_Trigger;
2  *(__IO uint32_t *) tmp |= value;
```

We modify the driver to use a switch case on the configuration and access I/O using fixed addresses, hence resolving the error. [4] For the last two errors, we notice that they are thrown from the FreeRTOS kernel. To this end, we explicitly mark those routines as kernel routines.

Illegal MMIO Pointers Usage Errors: Another set of errors, similar to the unbounded MMIO errors, is illegal usage of MMIO pointers. CRT-C emits an error if the firmware tries to pass MMIO pointers as function arguments. In general, most device drivers do not expose such an interface, however, some do use this pattern. For instance, the GPIO driver uses the port base address as the input argument. To this end, we modify the interface to take an identifier for the port instead of an MMIO address. Within the driver, we use a mapping table to map the input identifier to the port address.

Driver Association Errors: After fixing the above errors, CRT-C can establish an association between devices and drivers. For FreeRTOS, all of the drivers, except NVIC, were correctly identified by CRT-C. Upon further investigation, we found that FreeRTOS uses macros to directly access the kernel for interrupt control. To this end, we redirect the macro to use the NVIC driver instead of directly accessing NVIC. In our firmware, CRT-C was able to find five device drivers including NVIC, GPIO, and EXTI drivers.

Checked Scope Errors: After proper association between devices and drivers, CRT-C mandates the usage of safe code for drivers and threads. CRT-C complains about the device drivers written in unchecked dialects. To convert the existing codebase, we utilize 3C [3] to semi-automatically convert unsafe code to safe code in an iterative manner. A detailed discussion about this process can be found in the 3C manual [16].

Kernel Resource Access Errors: During thread discovery, CRT-C finds all of the threads in the system. Some privileged threads are used by FreeRTOS for maintaining kernel bookkeeping, including the idle thread and the timer thread. Since these threads are considered part of the kernel, we mark them as privileged threads allowing them to use kernel dialect and freely call kernel functions.

```
1  KERNEL_THREAD
2  static portTASK_FUNCTION(prvIdleTask)
```

After these modifications, the firmware passes all of the CRT-C checks. Overall, CRT-C threw 11 device isolation-related errors and three kernel resource access-related issues for FreeRTOS. For FreeRTOS we modified around 500 SLOC for resolution of compile-time errors, which is negligible considering the codebase consisted of 79K SLOC[5]

We see similar errors for Zephyr, since both RTOS adapt their drivers from CMSIS [4] libraries. Therefore, we only highlight the major differences for Zephyr. The major changes were observed for MMIO pointer accesses in the kernel: In addition to the process creation and memory management library, Zephyr's balanced Red/Black Tree [2] library also utilized MMIO pointers, resulting in 15 violations in the kernel (compared to two in FreeRTOS). Moreover, Zephyr uses a device tree [12] based infrastructure instead of passing MMIO pointers in the driver interface, thus resulting in fewer illegal MMIO pointer usage errors. Lastly, we did not face any kernel resource access errors, as Zephyr uses an internal API (`z_setup_new_thread`) to create system-level threads, whereas FreeRTOS uses the same API for user and kernel threads. Overall, CRT-C threw 27 device isolation errors for Zephyr.

Application Porting - recmutex. CRT-C automatically finds all applications level code and enforces the thread policies which may lead to various compile-time errors. We use the recursive mutex application to demonstrate the application porting process. We group the errors into different categories and describe how we fix each error and get the application compiled by CRT-C.

Checked Scope Errors: The recmutex app uses three threads. For each thread, CRT-C reports that they are not written in a safe dialect. Similarly to RTOS porting, we can use the 3C tool to automatically convert the unsafe code to safe code.

Kernel Resource Access Errors: For new applications, CRT-C throws a type error when a user thread directly calls kernel API. For the recmutex app, CRT-C reports 18 calls to system code, with four distinct APIs. Instead of modifying the source code to replace the original call with the new safe interface, we design the new safe interface to be compatible with the original API, by redefining functions in the compatibility layer header file, as shown below:

For example, `xSemaphoreTakeRecursive` is an existing FreeRTOS API to take a semaphore. The API requires a handle to the mutex of type `SemaphoreHandle_t` and the number of ticks the API is allowed to wait. However, CRT-C does not allow user tasks to handle raw pointers to kernel objects. Therefore, the compatibility layer redefines the raw pointer to a safe pointer as shown below:

```
1  #define SemaphoreHandle_t const ptr<Queue>
```

This enables transparent redefinition of all mutex handles in the application. However, due to this redefinition, the old API becomes incompatible with the redefined handle type. Therefore, to this end, the compatibility layer defines a safe interface that is compatible with the safe handle as shown below:

```
1  #define xSemaphoreTakeRecursive(mut, tick)
2  SafeQueueTakeMutexRecursive(mut, tick)
```

The safe interface has the same interface as the legacy API but uses safe handles instead of raw pointers to kernel objects. However, since the raw pointers are already redefined, we can use existing code without any modifications by using the compatibility layer. The safe interface sanitizes the input

[4]Constraining the configuration input in the faulting function is also a viable solution.

[5]Complete FreeRTOS consists of 4M SLOC, we only consider the FreeRTOS kernel for source code comparison.

arguments before passing them to the kernel and tracks the ownership of any dynamic objects. To port the recmutex app, we extend the compatibility layer for all four APIs used by the application. We provide a helper library for object tracking. Note that, once the interfaces are added to the compatibility layer, future applications can easily use this function without any modifications, which drastically eases the adaption process for CRT-C.

Object Sharing Errors: CRT-C also reports any implicit sharing of data between different threads. For the recmutex app, CRT-C reported a total of six errors. In general, the resolution of each error is case by case. For instance, in the recmutex, all threads share a global variable to set the status of the demo. Therefore, we explicitly share the variable among all of the threads as shown in the following listing:

```
1 OWNER(taskA ,taskB , taskC)
2 BaseType_t xErrorOccurred = pdFALSE;
```