Securing Real-Time Microcontroller Systems through Customized Memory View Switching

Chung Hwan Kim*§ Taegyu Kim† Hongjun Choi† Zhongshu Gu‡
Byoungyoung Lee† Xiangyu Zhang† Dongyan Xu†
*NEC Laboratories America †Purdue University ‡IBM T.J. Watson Research Center
*chungkim@nec-labs.com †{tgkim, choi293, byoungyoung, xyzhang, dxu}@purdue.edu ‡zgu@us.ibm.com

***Abstract*—Real-time microcontrollers have been widely adopted in cyber-physical systems that require both real-time and security guarantees. Unfortunately, security is sometimes traded for real-time performance in such systems. Notably, memory isolation, which is one of the most established security features in modern computer systems, is typically not available in many real-time microcontroller systems due to its negative impacts on performance and violation of real-time constraints. As such, the memory space of these systems has created an open, monolithic attack surface that attackers can target to subvert the entire systems. In this paper, we present MINION, a security architecture that intends to virtually partition the memory space and enforce memory access control of a real-time microcontroller. MINION can automatically identify the reachable memory regions of real-time processes through off-line static analysis on the system's firmware and conduct run-time memory access control through hardware-based enforcement. Our evaluation results demonstrate that, by significantly reducing the memory space that each process can access, MINION can effectively protect a microcontroller from various attacks that were previously viable. In addition, unlike conventional memory isolation mechanisms that might incur substantial performance overhead, the lightweight design of MINION is able to maintain the real-time properties of the microcontroller.**

I. INTRODUCTION

A *microcontroller* is a small computer that contains a processor, memory, and various input/output (I/O) peripherals. Among various types of embedded systems based on microcontrollers, *real-time microcontroller systems (MCS)* are designed to meet the deadline constraints of the real-time processes that run atop. The primary goal of a real-time MCS is not for high throughput, but for the responsiveness—any failure to meet the response deadline may lead to non-trivial safety impacts. In fact, many cyber-physical/embedded systems today are relying on real-time MCS: for example, manned and unmanned vehicle systems, rocket and satellite systems, robot control systems, and real-time health-care systems. Unfortunately, we note that the security of real-time MCS has not yet received enough attention although real security threats [12], [27], [32], [33], [39], [51] have increasingly been reported.

§Work conducted when he was a Ph.D. student at Purdue University.

Network and Distributed Systems Security (NDSS) Symposium 2018
18-21 February 2018, San Diego, CA, USA
ISBN 1-1891562-49-5
http://dx.doi.org/10.14722/ndss.2018.23107
www.ndss-symposium.org

Among computer systems security methods, *memory isolation* is one of the most common and effective techniques. In this paper, we focus on the following two general and popular memory isolation techniques: process and kernel memory isolation. *Process memory isolation* isolates each process's virtual memory space from other processes. Due to this restricted memory accessibility, it is difficult for an attacker to launch memory corruption attacks to other processes from a victim process that he or she has already compromised. *Kernel memory isolation* separates kernel memory space from a user process context via privilege separation, rendering privilege escalation attacks difficult to perform by exploiting memory corruption issues in the kernel (i.e., illegally acquiring the kernel execution context from the user execution context). These two memory isolation techniques constitute the fundamental security principle, the principle of least privilege, preventing an attacker from subverting the entire system even if he or she successfully compromised a subset of the system.

However, it is challenging for real-time MCS to support such memory isolation in practice [31], due to hardware and performance constraints [11]. According to our survey of 67 commodity real-time MCS, *none of them is designed to employ process memory isolation or kernel memory isolation*. Table I summarizes the results of our survey of various types of real-time MCS including unmanned aerial vehicle (UAV), unmanned ground vehicle (UGV), remotely operated underwater vehicle (ROV), real-time 3D printer, and real-time Internet of Things (IoT) devices. Process memory isolation through virtual memory is not supported, because they are built on hardware that does not provide a memory management unit (MMU). Also, kernel memory isolation is not employed due to its negative impact on real-time performance induced by frequent privilege mode switching (between kernel and user modes). In fact, several RTOSs [14], [20], [22] running on real-time MCS optionally supports kernel memory isolation, but it is often turned off in practice as it tends to violate the real-time constraints, as shown in our study (§II-C).

Without strong memory isolation, MCS expose a large attack surface to attackers. More precisely, without virtual memory support, these systems simply layout everything into *a single memory space* (illustrated in Fig. 1). Non-volatile and volatile memory, such as a flash ROM and an on-chip SRAM, are hard-wired to fixed locations, and peripheral devices are mapped to specific areas of the memory space through memory-mapped I/O (MMIO). In addition, all software modules, such as applications, libraries, device drivers, and the OS, are executed in the same privilege mode (i.e., the privileged mode) and have access to the entire shared memory space. This makes the memory space a large *attack surface* open to attackers who

TABLE I. AVAILABILITY OF PROCESS AND KERNEL MEMORY ISOLATION IN COMMODITY REAL-TIME MCS. P: PROCESS MEMORY ISOLATION SUPPORT. K: KERNEL MEMORY ISOLATION SUPPORT.

Type	RTOSs	Manufacturers	# of MCS	P	K
UAV	NuttX	3DR, enRoute, Virtual Robotix, ...	18	✗	✗
	FreeRTOS	Storm Racing Drone, RISE	30	✗	✗
	ChibiOS	Parrot	6	✗	✗
UGV	NuttX	Erie Robotics, HobbyKing	2	✗	✗
ROV	NuttX, OpenROV	BlueRobotics, OpenROV	2	✗	✗
3D Printer	Marlin	D-creator, BQ WitBox, Wombot, ...	7	✗	✗
IoT	FreeRTOS	Mongoose, Particle	2	✗	✗

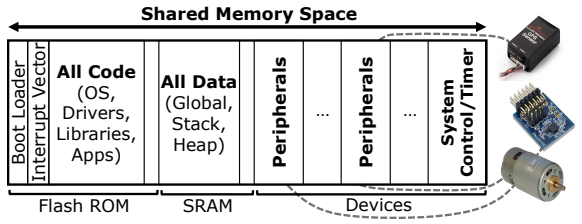


Fig. 1. Memory space layout of real-time MCS.

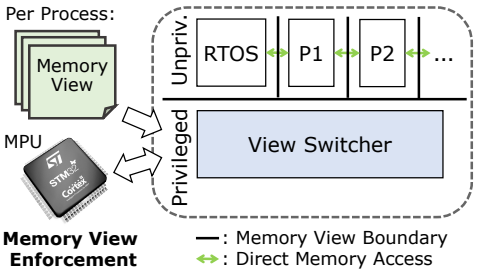


Fig. 2. Overall architecture of MINION.

could successfully compromise any of the software modules by exploiting a memory corruption vulnerability in real-time MCS, as demonstrated in previous incidents [1]–[4], [40], [49]. For instance, an attacker may exploit a buffer overflow vulnerability in a telnet server through Wi-Fi in a UAV [40] and navigate through the memory space to corrupt critical software and hardware components, such as the flight control program and actuators, to maliciously operate the vehicle. For such systems, any vulnerable software module becomes an "Achilles heel" of the entire system. Compromising one of the software modules is equivalent to taking over the whole system, including both the software stack and its associated physical and networking components.

Although there have been extensive research efforts to provide isolation to programs in embedded devices [21], [28], [42], [48], [52], they have limitations in achieving *both* real-time performance and memory isolation. Most of them require manual efforts to identify which subject can access what memory regions, because it is difficult to accurately identify the memory boundaries between processes in the shared, monolithic memory space. Recent work [31] proposed an approach to automatically identifying sensitive instructions and escalating the privilege mode for hardware-based access control before these sensitive instructions are executed. Unfortunately, to preserve real-time properties, it is not always acceptable to escalate execution privilege for all sensitive instructions, as those instructions may be executed very frequently in real-time MCS. Thus, it would severely affect the responsiveness of the system, similar to how current RTOS with kernel memory isolation violate the real-time constraints as shown in our study (§II-C). For example, peripheral I/O accesses are security critical (e.g., reading GPS signals and controlling the actuators of a UAV), and real-time MCS frequently perform such accesses to closely control the peripherals. To properly protect such accesses, the existing technique would have to escalate the privilege mode for every peripheral's I/O operation, failing to meet real-time requirements.

In this paper, we propose MINION, which automatically reduces memory spaces of real-time MCS while meeting real-time constraints. The key idea behind MINION is *an efficient memory view switching mechanism for memory isolation*, which avoids frequent privilege mode switching. Fig. 2 illustrates the overall architecture. The workflow of MINION consists of the following three tasks. First, MINION captures the approximated minimum set of memory regions essential to correctly operate each process. MINION performs a conservative static program analysis based on points-to analysis, which comprehensively considers all memory types in real-time MCS (i.e., code, data, and peripheral-mapped memory). Next, MINION constructs a *per-process memory view* by assembling the previous analysis results. To overcome the limitation of current memory protection hardware in real-time MCS, MINION performs a clustering analysis, tailoring the memory views to be compatible with the underlying hardware mechanism. Finally, MINION enforces memory isolation based on the tailored per-process memory view using the *view switcher*. The view switcher is the *only* trusted computing base (TCB) that runs in the privileged mode, and it is isolated from the RTOS and processes running in the unprivileged mode.

The unique architecture of MINION allows the target real-time MCS to meet both security and real-time requirements. With respect to satisfying real-time constraints, the RTOS and processes can avoid expensive privilege mode switching while interacting with each other through direct memory access, as MINION simply runs them in the same (unprivileged) processor mode. In terms of security guarantees, since each process's memory accessibility is strictly limited to its own memory view using the view switcher, MINION reduces the attack surface of each process and supports memory isolation as desired. As a result, when a compromised process makes an illegal access that breaks the constraint of the memory view, the view switcher detects this as a violation using *hardware-based memory isolation*.

In comparison with conventional systems based on virtual memory, the memory blocks that form each memory view identified by MINION are *not contiguous* as the memory blocks for different processes mingle with each other in the monolithic memory space. In addition, unlike an OS kernel that frequently intervenes in the process execution and elevates the privilege mode (through hardware interrupts and system calls), the view switcher is only executed whenever there is a context switching between processes. We note that this only adds *a little overhead* since the RTOS has to interrupt process execution anyway to handle context switching, regardless of the view switcher. According to our evaluation, the prototype of MINION was able to meet the real-time constraints while significantly reducing memory attack surfaces (76% with the hardware limitation and 93% without it).

The contributions of this paper are summarized as follows.

- **Design:** We present a new security architecture that reduces the attack surface of real-time MCS and overcomes the challenges that conventional and previous memory isolation approaches did not fully address (§IV).
- **Implementation:** We implement the proposed architecture on a commodity real-time MCS hardware to demonstrate its practicality (§V).
- **Evaluation:** We provide quantitative evaluation results on various aspects: (1) the performance overhead of MINION and its impact on the responsiveness of real-time processes, (2) the effectiveness of MINION's security guarantee against realistic attack cases built based on one of the four new memory corruption bugs that we discovered, and (3) the rate of memory space reduction that MINION provides with and without the limitation of current memory protection hardware (§VI).

The rest of this paper is organized as follows. §II describes the background. §III defines our threat model. §IV presents the design of MINION in detail. Implementation and evaluation of MINION are presented in §V and §VI, respectively. §VII shows the differences between MINION and related work. §VIII discusses the limitations of MINION. §IX concludes this paper.

II. BACKGROUND AND MOTIVATION

In this section, we provide the background on real-time MCS and describe the motivation behind MINION.

A. ARM Cortex-M and Cortex-R

A majority of MCS are built based on the ARM Cortex-M [8] and Cortex-R [10] processors. Both Cortex-M and Cortex-R provide various features that real-time MCS can leverage to meet the deadline constraints of real-time processes. Specifically, Cortex-M provides a high-resolution timer (called *SysTick*) that generates precise timer interrupts with different priorities. RTOSs that support Cortex-M rely on the timer interrupts for the on-time scheduling of real-time processes. In addition, the interrupt controller is tightly coupled to the core to support fast interrupt handling, providing high responsiveness to hardware events such as timer interrupts and peripheral events. Cortex-R [10] is another example of ARM processors that support real-time MCS. Both processor families support privilege separation through two separate processor modes, namely *privileged and unprivileged modes*.

B. Memory Protection Unit

A memory protection unit (MPU) provides hardware-enforced memory isolation. All Cortex-M and Cortex-R processors based on the ARMv7 architecture (including its next versions) have an MPU [7], [8], [10], [13]. Unlike an MMU, it neither supports virtual memory nor controls memory accesses through page tables. Instead, an MPU provides a fixed number of hardware registers, each of which specifies a per-region access control rule for physical memory. This allows an MPU to quickly read access control rules as it does not need to traverse page tables located in memory. The access control rule for each memory region has two permission sets: one is used when the processor is in the privileged mode and the other is used when it is in the unprivileged mode. Each permission set can include read, write, and execute permissions. If a memory region is assessed without the required permissions by the corresponding access control rule, the processor raises a permission fault (similar to a page fault in an MMU-based system). Memory accesses in privileged and unprivileged modes are controlled separately as the access control rule has distinct permission sets for them. The number of supported MPU regions varies from 8 to 16 depending on the model and configuration of the processors.

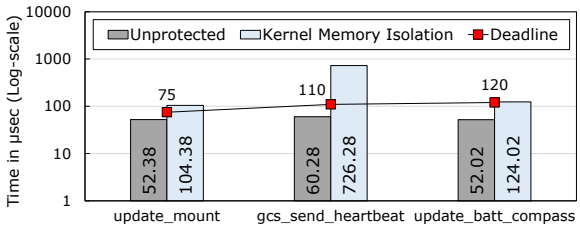


Fig. 3. Limitation of kernel memory isolation. The deadline constraints of the three real-time tasks on 3DR IRIS+ UAV are violated due to frequent privilege mode switching.

C. Motivation of MINION

Limitations of Current RTOSs. A number of RTOS's [14], [20], [22] optionally provide kernel memory isolation (but not process memory isolation) using the privilege separation and MPU that the ARM processors support. However, kernel memory isolation is seldom employed by real-time MCS [31], due to its negative impact on performance constraints [11]. Since the software and physical components of a real-time MCS are highly interactive, software modules perform frequent peripheral I/O operations. However, if kernel memory isolation is enforced, only the kernel has exclusive access to the devices. Software modules in the user space, such as applications and libraries, are forced to request the kernel to perform I/O and wait for the outcome (typically through a system call). This leads to frequent privilege mode switching between the kernel and user spaces, which in turn incurs the significant overhead caused by the kernel's intervention for all peripheral I/O operations. Furthermore, user space programs often have to execute other kernel functions that do not perform peripheral I/O: for example, to use the heap memory, to synchronize with other software modules, or to send a signal to a process. These all contribute to the fact that real-time MCS do not employ kernel memory isolation to avoid disrupting the responsiveness, the critical operational factor of real-time processes.

As a more concrete demonstration of the negative impact of kernel memory isolation, we manually enabled kernel memory isolation in a popular UAV system, a 3DR IRIS+ quad-copter and observed whether it meets real-time constraints. We note this UAV system by default does not enable kernel memory isolation. More specifically, we randomly selected a number of real-time tasks in a 3DR IRIS+ quad-copter and measured the impact of kernel memory isolation on their real-time constraints by manually enabling it with a firmware modification (Fig. 3). We observed that the system failed to meet the deadline

constraints of the real-time tasks due to frequent switching between the kernel and flight controller for device I/O and kernel function execution. As such, current real-time MCS tend to allow any software module to directly execute kernel code and perform peripheral I/O through direct MMIO without switching to the kernel.

MINION Approach. Motivated by this problem, the key idea behind MINION is to keep all software modules in the same privilege mode. Unlike the current architecture of real-time MCS that runs all software modules in the privileged mode, MINION *runs them in the unprivileged mode*. This architecture is also different from previous protection work [21], [28], [31], which runs OS in the privileged mode and other software modules in the unprivileged mode. MINION's lightweight software module, called the *view switcher*, is *the only program that runs in the privileged mode*. By doing this, MINION avoids the costly privilege mode switching overhead. To reduce the memory space of each real-time process that can be used as a large attack surface, MINION statically analyzes the firmware of the MCS and identifies per-process *memory views*. The view switcher leverages the MPU to effectively enforce the memory views with minimal run-time overhead. Since the memory views are enforced per process, the view switcher only has to take place once at every context switching time to re-configure the MPU. Due to the limitation of current MPUs with a small fixed number of supported memory regions (§II-B), the memory ranges in each memory view are clustered to be compatible with the hardware.

III. THREAT MODEL AND ASSUMPTIONS

This paper aims to neutralize attacks with illegal memory corruption capabilities against real-time MCS. In particular, we assume that attackers have found a memory corruption vulnerability in any of the underlying software stack layers, and they are capable of launching a memory corruption exploit. Once the attackers have successfully launched such an exploit, they can either inject code or reuse existing code in the target MCS to achieve their goal. Since all processes share the entire physical memory space, the attackers can do anything with the code, data, and peripheral devices mapped to the memory space in order to subvert the system at this point. For example, they can either execute safety-critical code in the system, manipulate security-sensitive data, or maliciously control peripheral devices by directly overwriting data to specific memory locations.

Unlike modern commodity operating systems and devices, largely due to its limited hardware supports and inherent performance constraints, real-time MCS lacks most of the state-of-the-art security hardening features—from process and kernel memory isolation to more advanced techniques including control-flow integrity (CFI) and memory safety. In this respect, we believe that such security threats against real-time MCS are realistic and require immediate attentions from security communities. There have been numerous memory corruption vulnerabilities in such systems [1]–[4], [40], [49]. In addition to the memory corruption issues that we discovered from the real-time MCS that we tested (§VI-B1), we also observed that almost 50% of all security-related patches on the MCS firmware were related to resolving memory corruption vulnerabilities [19]. This shows that memory corruption vulnerabilities are real threats to real-time MCS. In §VI-B, we also show that such vulnerabilities are exploitable in various attack scenarios with serious safety issues.

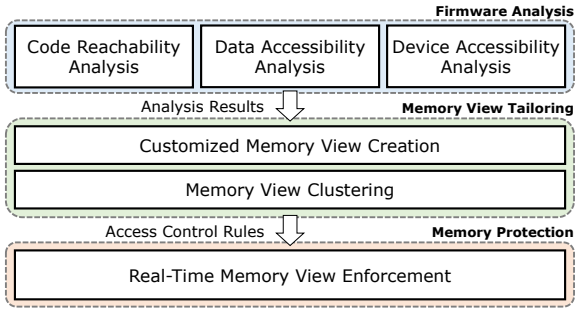


Fig. 4. Workflow of MINION.

We assume that the real-time MCS is equipped with an MPU and supports hardware-based privilege separation (i.e., more than one privilege mode). Both of the hardware features are already market-available in most microcontrollers including ARM based microcontrollers [31].

IV. DESIGN OF MINION

This section presents the design of MINION. Fig. 4 illustrates the overall procedures of MINION. First, MINION performs the off-line static analysis on the firmware of the target real-time MCS to capture a required set of memory regions to run each real-time process (§IV-A). Next, a memory view per process is created by assembling the results of the firmware analysis, and each memory view is further tailored through a clustering algorithm to generate a set of *access control rules* to overcome the hardware limitation of the MPU (§IV-B). Then a secure and lightweight software module, the view switcher, takes the access control rules as input, enforcing the memory views on the corresponding real-time process using the MPU (§IV-C). Lastly, we describe how MINION allocates isolated stack and heap memory space based on previous analysis results (§IV-D)

A. Firmware Analysis

Our firmware analysis aims at identifying which resources are accessed at run-time by each real-time process. More precisely, we categorize the resources of MCS into three types: code, data, and peripheral-mapped memory regions. We design the analysis targeting each of these resource types, namely code reachability analysis (§IV-A1), data accessibility analysis (§IV-A2), and device accessibility analysis (§IV-A3). The analysis is performed on *bitcode*, the intermediate representation (IR) of the Low Level Virtual Machine (LLVM [45]).

We choose static analysis over dynamic analysis techniques [5], [26], [46], [47] so as to avoid incompleteness of our analyses on memory accesses to code, global data, and peripherals[1]. One of the well-known limitations of dynamic analysis techniques is that it cannot provide completeness since their analysis scope only covers a subset of a program that is executed during training runs. This limitation may lead to a *critical run-time issue* if used for MINION—if the memory

[1]It is worth noting that stack and heap spaces are not subject to static analysis. For them we leverage run-time profiling.

view of a real-time process does not contain one of the memory regions required for the real-time process to operate correctly, the process would result in undefined run-time behaviors. This problem becomes even more severe when the real-time MCS is used as a safety-critical system.In contrast, although static analysis may over-estimate the memory views, the analysis results are conservative and thus achieve completeness by design. Therefore, the target MCS would not suffer from such a security issue under MINION's protection.

1) Code Reachability Analysis: The goal of this analysis is to conservatively identify code that a target real-time process may execute for correct operations. Without loss of generality, we analyze at the function granularity. In principle, the goal of this analysis is equivalent to finding all functions that are reachable from the *entry functions* of the process. Those entry functions include the *start function* of each process, which is the first function that the process executes, and a set of interrupt handlers that are triggered when hardware events occur.

In order to identify all reachable functions for each entry function, we first construct a call graph of the whole firmware. In the call graph, functions are connected through two types of control flow transitions, *direct* and *indirect* calls. It is straightforward to analyze direct call transitions in bitcode, since the call targets are explicitly stated. However, the sound analysis of indirect call targets is challenging. The firmware of a real-time MCS consists of the code and data of various software modules (e.g., applications, libraries, device drivers, and the OS) and indirect control transitions are heavily practiced across the entire code space since these modules are highly interactive. As such, we leverage inter-procedural points-to analysis [53] to resolve indirect call targets precisely and construct the call graph that includes both the direct and indirect calls. More specifically, our points-to analysis focuses on resolving the possible targets of the function pointer through both flow-sensitive and context-sensitive analysis.

Once a set of reachable functions for each entry function is identified, each function in the set is marked as executable such that a process with the function in its memory view can execute it at run-time. After that, these function sets are merged to generate a unified set of reachable functions for each process.

In order to obtain the input of this analysis (i.e., entry functions), we populate the list of the process start functions by running the code reachability analysis on the OS function responsible for creating a new process. Similarly, the list of interrupt handlers for each process is populated through our code reachability analysis on each interrupt handler of the RTOS and by checking if any of the reachable functions from the start function is also reachable from the interrupt handlers.

2) Data Accessibility Analysis: There are three types of data: global, heap, and stack objects. To generally analyze all of these data types, MINION takes two different schemes to identify per-process data accesses: (1) for global data, the analysis performs forward slicing augmented with points-to analysis; and (2) for stack and heap, our analysis identifies the location and size of each data segment.

Global Data. We leverage forward slicing to identify memory accesses to global data for each real-time process. A forward slice contains all the instructions that directly read and write each global variable (or constant) and those that indirectly affect the potential value of the variable through *pointer aliasing*. Direct memory accesses to the global data are identified by detecting all the loads and stores that use the global data as the operands. Our forward slicing is performed on the result of the inter-procedural points-to analysis to identify the aliasing of the global variables within a single function and across multiple functions through function arguments. We discard every forward slice that does not lead to the de-reference of an alias since it indicates that the aliased data are not accessed in the slice.

Based on remaining forward slices, we analyze which functions in the call graph directly or indirectly access the global data and build a list of accessible global data for each function. These results are then used to identify the accessibility of the global data per process. Using the result of the code reachability analysis, we mark only the global data that the reachable functions in the process access as readable and/or writable, depending on the type of the memory access—i.e., a *load* instruction indicates reading and a *store* instruction indicates writing the data, respectively.

Stack and Heap Data. In addition to global data, MINION also provides protection to the stack memory and heap memory of each real-time process. Although the locations of these memory regions are determined at run-time, the sizes of the stack memory and the heap memory pool for each process can be identified with some bounded engineering effort. Specifically, we identified these memory sizes in a set of profiling runs by annotating the stack and heap allocating functions of the RTOS. Using the profiling results, we first determine the locations of the stack memory and heap memory pool for each process off-line and then use these memory locations at run-time to allocate the stack and heap memory pool deterministically. When determining the locations, we ensure that the stack and heap pool of each process are adjacent in the memory space thereby both of the memory blocks can be contained within one memory region. The memory region is then marked as both readable and writable in the memory view, such that only the process that owns the stack and heap can access the memory exclusively. The memory allocator of the RTOS is slightly modified to allocate the stack and heap at the locations specified in the memory view (§IV-D).

3) Device Accessibility Analysis: In addition to the aforementioned analyses, we statically analyze the MCS firmware to identify memory accesses to peripheral devices through MMIO. Our static analysis leverages backward slicing to identify these MMIO accesses. It is feasible to statically identify these accesses because the addresses of peripheral-mapped memory regions are embedded as constants in the firmware code in practice. MMIO addresses are hard-coded in the firmware as they are coordinated by the hardware design and thus cannot easily be updated. There are three patterns that cover most MMIO accesses that we identified based on our study of diverse MCS firmware code. Fig. 5 presents example code snippets.

Case 1. Before accessing a peripheral-mapped memory region, a function (`enable_irq`) calls a "query" function (`irqinfo`) that returns the address of a corresponding peripheral device to a given device identifier (`irq`). The returned address is then used by the caller function to read and/or write

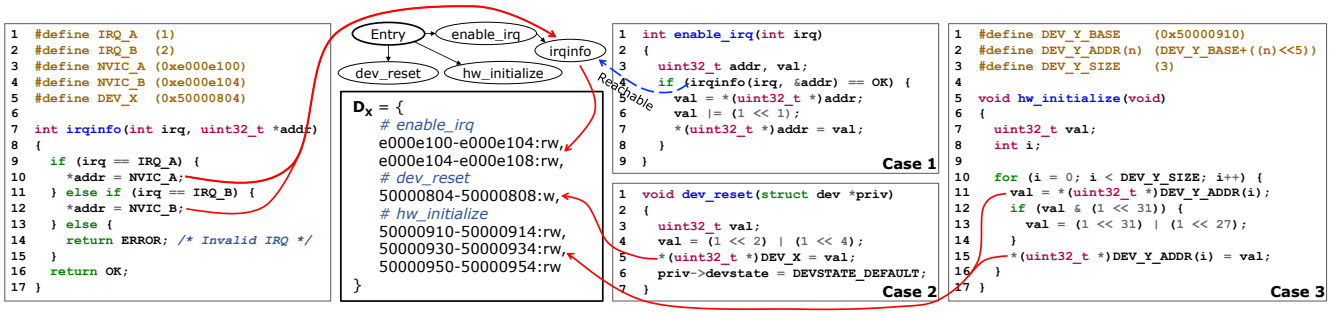


Fig. 5. Device accessibility analysis. Code snippets are taken from the NuttX RTOS and simplified for readability. D_X is a set of the peripheral-mapped memory regions accessible from process X.

to the peripheral-mapped memory region. We note that although the query function reads the MMIO address values, it does not dereference them to access the peripheral-mapped memory regions. Instead, its caller function uses the obtained address to access the memory region. To identify such propagation of MMIO addresses across function boundaries, our backward slicing is augmented to perform an inter-procedural analysis.

Case 2. An MMIO address is directly dereferenced to access the peripheral-mapped memory region without involving an inter-procedural data flow, as shown in `dev_reset`. Our study shows that this simple pattern is the most commonly adopted in most MCS firmware code.

Case 3. For some peripheral devices, programs directly access a number of adjacent memory regions in a loop, as shown in `hw_initialize`. Such a pattern can be addressed by our backward slicing as it can find the instructions that either directly or indirectly affect an instruction that uses an MMIO address.

Our analysis finds every load and store instruction that uses an MMIO address as an operand, and then uses backward slicing to identify the instructions that affect the operands of the loads and stores. The memory layout of the MCS is given as an input, as such, we can identify that a constant operand within a specific range of value is an MMIO address. Each backward slice includes all the instructions that affect the operands directly or indirectly. Using these slices, we build a list of accessible peripheral-mapped memory regions for every function in the firmware. Similar to the global data analysis, the result of the code reachability analysis is used to mark only those peripheral-mapped memory regions that the reachable functions in each process accesses for MMIO operations.

B. Memory View Tailoring

We define a memory view as a list of access control rules, each containing the location and permission type of an accessible memory region. MINION constructs a memory view based on the results of the firmware analysis (§IV-A).

1) Memory View Creation: Using the results of the firmware analysis, MINION constructs the memory view V_X of the process X, which represents legitimate access permissions for a given address. More formally, V_X is an associative array in which its key is an address and its value is a list of granted access permissions (i.e., $\{r, w, x\}$ where each letter represents readable, writable, and executable, respectively).

Algorithm 1 Creation of a memory view using the firmware analysis results.

```
1: function INSERTMEMORYREGION(V_X, start, end, perm)
2:     for addr ∈ {start...end} do
3:         v_i ← V_X[addr]
4:         v_i.perm ← v_i.perm ∪ perm
5: function CREATEMEMORYVIEW(X)
6:     V_X ← new hash table
7:     C_X ← ∅
8:     for f ∈ E_X do
9:         C_X ∪ reachable functions from f
10:    for f ∈ C_X do
11:        INSERTMEMORYREGION(V_X, f.start, f.end, {x})
12:    for {f, mem, perm} ∈ G_X do
13:        if f ∈ C_X then
14:            INSERTMEMORYREGION(V_X, mem.start, mem.end, perm)
15:    INSERTMEMORYREGION(V_X, S_X.start, S_X.end, {r, w})
16:    INSERTMEMORYREGION(V_X, H_X.start, H_X.end, {r, w})
17:    for {f, mem, perm} ∈ D_X do
18:        if f ∈ C_X then
19:            INSERTMEMORYREGION(V_X, mem.start, mem.end, perm)
20:    return V_X
```

Similar to V_X, the firmware analysis results can be denoted as associate array—i.e., C_X, G_X, S_X, H_X, and D_X, each of which represents the results of the code reachability, global data, stack memory, heap memory, and device accessibility analyses for process X, respectively. In this associate array denoting the firmware analysis, its value is either the specific access permission inherent to the analysis type or a combination of access permissions per element determined by the analysis. For example, the access permission for C_X is $\{x\}$. Moreover, each element of G_X is a combination of $\{r, w\}$, depending on the type of the access determined by the analysis. Similarly, we use the access permission $\{r, w\}$ for S_X and H_X to allow both reads and writes to the stack and heap memory, and a combination of $\{r, w\}$ for each element of D_X based on the access type. For each address, V_X is populated through enumerating all firmware analysis results (i.e., C_X, G_X, S_X, H_X, and D_X) while taking the union of permissions per address. Algorithm 1 presents the pseudo-code to construct a memory view using the results of the firmware analysis.

2) Memory View Clustering: In order to enforce memory views, MINION leverages an MPU to strictly observe the real-time constraints of the MCS. Since MPU-based memory isolation is highly efficient, the overhead of MINION's memory view enforcement would be very low, and thus the real-time processes can meet their deadline constraints while having significantly reduced per-process memory space. The challenge, however, is in the limited number of access control rules that

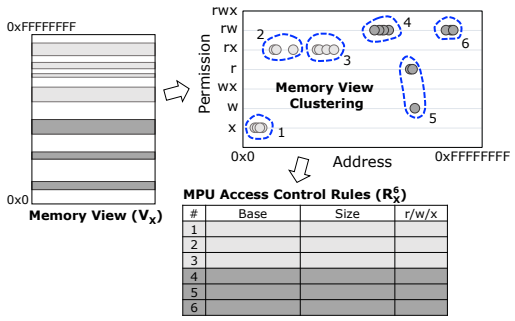


Fig. 6. Memory view clustering. R^6_X is a set of access control rules for an MPU that supports 6 protected memory regions at maximum.

Algorithm 2 Generation of MPU access control rules using k-means clustering.

```
1: function GETPERMISSION(P)
2:     perm ← ∅
3:     for p ∈ P do
4:         perm ← perm ∪ p
5:     return perm
6: function ACCESSCONTROLRULES(V_X, N)
7:     S ← new |V_X| * 2 array
8:     for v_i ∈ V_X do                          ▷ Project V_X into a 2-D surface
9:         S[i][0] ← v_i.addr
10:        S[i][1] ← v_i.perm
11:    L ← KMeans.cluster(S, N)                  ▷ k-means clustering
12:    C ← new hash table
13:    for l_i ∈ L do
14:        C[l_i].I ← new list
15:        C[l_i].P ← new list
16:    for l_i ∈ L do                            ▷ Re-organize using the cluster labels
17:        C[l_i].I.append(S[i][0])
18:        C[l_i].P.append(S[i][1])
19:    R^N_X ← new list
20:    for c ∈ C do                              ▷ Generate MPU access control rules
21:        base ← min(c.I)
22:        size ← max(c.I)+1 − base
23:        perm ← GETPERMISSION(c.P)
24:        R^N_X.append({base, size, perm})
25:    return R^N_X
```

can be enforced through MPU. As described in §II-B, MPU only supports the small fixed number of memory regions.

To overcome this challenge, MINION uses clustering to group similar objects together. MINION designs this clustering algorithm to minimize the number of access control rules. Each access control rule represents the clustered group with similar addresses and the same access permission type. More specifically, MINION clusters the elements of V_X into a constant number of access control rules based on the *k-means clustering* algorithm. Fig. 6 illustrates how MINION tailors a memory view through *memory view clustering* into a set of access control rules to overcome the limitation of the current MPUs (§II-B). V_X is projected into a 2-dimensional space where the X-axis is memory addresses and the Y-axis is permission types. The projected V_X bytes are then grouped into a set of clusters (U). For each cluster in U, the memory addresses and permission types of member bytes are re-organized to generate a set of MPU access control rules (denoted as an element of R^N_X where N is the number of supported MPU regions). The base address ($base$) and the size ($size$) of the memory region are calculated based on the lowest and highest memory addresses of each cluster. The permission type ($perm$) is conservatively determined by computing the union of all

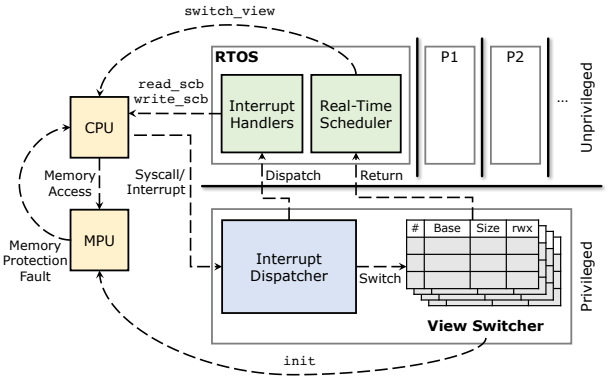


Fig. 7. Detailed architecture of memory view enforcement.

permission types in the cluster, for the same reason why we use a union for memory view creation (§IV-B1). Algorithm 2 provides the pseudo-code that clusters a memory view into a number of MPU access control rules.

C. Memory View Enforcement

Given the MPU access control rules (i.e., a memory view per process) generated in the previous phase, MINION enforces the memory views on the real-time processes while the MCS is running, as illustrated in Fig. 7. The view switcher is running in the privileged mode, responsible for securely enforcing the memory views using the MPU access control rules. Since the RTOS and real-time processes run in the unprivileged mode, they cannot tamper with the view switcher. The MCS firmware is instrumented with a small number of *system calls* that MINION provides. Whenever a MINION system call is invoked by a software module in the unprivileged mode, the processor elevates the privilege before the view switcher receives the control. The processor drops the privilege when the system call returns. The details of the MINION system calls are as follows.

`init` is invoked when the MCS firmware bootstraps. More precisely, it is called immediately after the RTOS finishes the device initialization and before the first real-time process is created. When invoked, it configures the MPU such that the memory regions that contain the view switcher's code and data are inaccessible from the unprivileged mode (i.e., the RTOS and real-time processes). The view switcher keeps the input MPU access control rules in a protected memory region, so any unprivileged access to them will raise a memory protection fault and in turn the view switcher will be notified. The location information of the stack memory and heap memory pool per process, determined during the firmware analysis, is loaded into a read-only memory region that no real-time process can access but the RTOS. This is because the RTOS needs to use this information to allocate stack and heap following the memory views (§IV-D). The interrupt vector table is configured in a way that all interrupts are first handled by MINION and then dispatched to the RTOS's interrupt handlers after dropping the privilege. After the bootstrapping, the privilege is dropped and the RTOS receives the control back to proceed.

`switch_view` is invoked by the real-time scheduler of the RTOS whenever it performs context switching between two real-time processes, from an old process to a new process to be scheduled in. The address of the new process's start function

is leveraged by the view switcher as an identifier to find the corresponding memory view among all the memory views loaded in the protected memory region. Using the access control rules in the memory view, the view switcher re-configures the MPU such that the new process can only access to the memory regions within the view. After the configuration, the new process can directly execute the code and control the peripherals through MMIO for benign operations *without a delay*, as the corresponding memory regions are included in the memory view. On the other hand, any unprivileged memory access that does not comply with the access control rules (i.e., out of the memory view) will raise a memory protection fault. Once the view switcher finishes the MPU configuration, the control is returned to the real-time scheduler after a privilege drop and the RTOS finishes the scheduling procedure.

`read_scb` and `write_scb` allow software modules running in the unprivileged mode to securely access a special memory area, called the *System Control Block (SCB)*, through a restricted channel. Some software modules, including the RTOS, benignly accesses the SCB to control the processor (e.g., to configure the SysTick timer). Unlike other memory regions, any memory access to the SCB in the unprivileged mode generates a hardware fault by the hardware design, even though the MPU configuration allows the access. To this end, MINION introduces these two system calls, allowing software modules to access a memory region in the SCB only if the memory view of the current process includes it. We leverage the result of our firmware analysis (§IV-A3) to identify all instructions that access the SCB, and then replace each of these instructions with a call to `read_scb` or `write_scb`, depending on the access type. In our prototype, we find that only a small number of RTOS functions access the SCB and none of the real-time processes access it. When invoked, `read_scb` simply reads the requested four-byte data from the SCB and `write_scb` writes the provided data to it.

The privilege mode switching caused by the system calls does incur some performance overhead. However, we observe that the overall performance impact of MINION is small (2% on average) and does not affect the responsiveness of a commodity real-time MCS with deadline constraints for the following reasons. First, `init` is invoked only once when the system bootstraps and thus does not affect the performance of the MCS during the production run. Second, `read_scb` and `write_scb` are not infrequently invoked during production but only invoked at a high frequency while the RTOS is accessing the SCB to initialize devices. Lastly, although the main source of the overhead is `switch_view`, the impact is not substantial. Since the RTOS has to interrupt process execution anyway to handle context switching, regardless of MINION, `switch_view` only adds little computation to the context switching handler. The privilege mode is elevated by the microprocessor when the interrupt occurs and `switch_view` simply utilizes it. We will further discuss the performance impact in §VI.

D. Stack and Heap Allocation

The memory allocator of the RTOS is slightly modified for each real-time process to have its own isolated stack and heap memory that comply with the memory view. When a new process is created, the memory allocator reserves a memory pool in the shared memory space to allow dynamic memory allocation. Instead of managing the stack and heap memory separately, it allocates both stack and heap memory using the same memory pool of the process. The stack memory is created by allocating a memory block of a constant size in the memory pool, in the same way how a heap memory block is allocated using `malloc`, and by letting the process use it as the stack. We add a small modification, only 20 lines of code (LOC), to the memory allocator, such that the memory pool of each process is placed at the location specified in the memory view, determined during the firmware analysis (§IV-A2), instead of the original location. Since the memory pool is reserved only once per process creation, the performance impact of the modification is trivial. Security-wise, each process can only access its own stack and heap memory exclusively during execution as the view switcher enforces the memory view. A compromised process cannot access the stack or heap of another process across the memory view boundary, whether or not the attacker has knowledge about the memory pool.

V. IMPLEMENTATION

Our proof-of-concept prototype of MINION is implemented on 3DR IRIS+ [16], a popular quad-copter UAV based on the 3DR Pixhawk microcontroller [17]. We note the design of MINION is general and it can be applied to any real-time MCS with an MPU and privilege separation.

Pixhawk contains an ARM Cortex-M4 processor (with 8 MPU regions), a 192KB static RAM (SRAM), a 64KB core-coupled RAM (CCRAM), a 16KB read-only memory (EEPROM) and a 2MB flash ROM. It also has an ARM Cortex M3 co-processor that runs a small extra piece of firmware, separate from the main MCS firmware. This separate piece of firmware runs a servo system that operates the four actuators at the command of the main firmware. It supports a *fail-safe landing* feature, which safely lands the UAV when the main firmware detects an undesired event, such as low battery power. The microcontroller is equipped with a handful of peripheral devices including: sensors (a gyroscope, a GPS with an integrated magnetometer, an accelerometer, and a barometer), four 950kV actuators, and a telemetry radio signal receiver. The telemetry device allows the UAV to communicate with a Ground Control Station (GCS) and an RC controller.

Our prototype leverages the fail-safe landing feature when the view switcher detects that a real-time process makes an illegal memory access out of the memory view. When triggered, it lands the UAV by slowly and repeatedly decreasing the altitude until it touches the ground without changing its horizontal position. We note that MINION can also be deployed on top of the co-processor to protect the small, separate piece of fail-safe landing firmware.

Similar to most of the unmanned vehicles based on 3DR Pixhawk, the firmware of 3DR IRIS+ contains the ArduPilot flight controller [18], the UAVCAN CAN bus library [23], device drivers from the PX4 Autopilot module [15], and the NuttX RTOS [22]. We added 787 LOC for the view switcher and additionally added and modified 87 LOC of the RTOS code, including those added for modified stack and heap allocation. There was no modification in the flight controller, library, or device drivers. In addition, we wrote extra 1,590

LOC to implement the off-line firmware analysis based on an interprocedural points-to analysis tool [53], and memory view creation and clustering based on LLVM 3.9 and Python 2.7.6. All of the experiments in §VI were performed on the main firmware of the real-time MCS (3DR IRIS+), with or without the protection of MINION.

VI. EVALUATION

In this section, we evaluate our MINION prototype implemented on a commodity real-time MCS (3DR IRIS+ described in §V) to answer the following questions:

- What is the performance impact of MINION on real-time constraints of real-time MCS (§VI-A)?
- Can MINION effectively protect real-time MCS against realistic attacks (§VI-B)?
- How much memory space can MINION reduce (§VI-C)?
- What is the correlation between the limitation of MPUs and MINION's memory space reduction (§VI-D)?

A. Performance Impact

1) Real-Time Benchmarks: Fig. 8 shows the performance of 31 real-time tasks in comparison with the deadline constraints, with and without the protection of MINION. We used a high-resolution performance counter equipped in the microcontroller to measure the performance of the real-time tasks. These tasks run under the context of a flight controller process (`main_loop`), which handles the key operations of the real-time MCS (i.e., the flight control operations). Similar to the concept of a thread, a process may have more than one *real-time tasks* and each real-time task handles a specific job. Each real-time task executes a loop and is assigned two pre-determined time constraints: *an interval* and *a deadline*. The loop iterates at the given interval and it performs a specific operation at each iteration. For instance, the `rc_loop` task receives radio signals from an RC controller at every iteration of the loop with the interval of 100 Hz and it is assigned the deadline of 130 μsec. Similarly, `update_GPS` reads GPS signals and is assigned the interval of 50 Hz and the deadline of 200 μsec. The performance overhead of MINION must be small enough for the real-time tasks to meet the deadlines. Otherwise, it will have a negative impact on the UAV's flight.

For all of the tasks, the deadline constraints are not violated with or without the protection of MINION. The overhead in percentage (Fig. 8) shows average extra latency that each real-time task under protection spent within the time window of the deadline. The overhead of all 31 tasks was only 2% on average. The overheads of individual tasks ranged from -0.87% to 10.85%. Some of the tasks showed slightly better performance with the protection. We found that the execution time of these tasks are fairly short, and the presence of MINION did not increase their execution time. The non-determinism of the physical inputs (e.g., GPS signals), which we could not fully control during the benchmarks, should have caused the slightly increased performance results when the protection was enabled. On the other hand, 5 of the real-time tasks (e.g., `update_batt_compass`) showed larger than 5% overhead. Unlike the tasks with small overhead, the total latency of these tasks are generally longer than others regardless of the protection, so the chance of context switching during task execution is higher for them. As we aforementioned, MINION switches the memory view at every context switching time, and the overhead is proportional to the number of context switching events. According to our worst-case execution time (WCET) analysis with end-to-end measurements, the overhead of the tasks under protection was 6.13% on average. In all cases, the deadline constraints of the 31 real-time tasks were met with the protection of MINION while preserving the schedule-ability of the MCS.

TABLE II. MEMORY CORRUPTION BUGS OF THE ARDUPILOT FIRMWARE THAT WE FOUND WHILE WORKING ON MINION.

Module	Bug ID	Type	Confirmed	Patched
NuttX RTOS	S1*	Global buffer overflow	✓	✓
PX4 drivers	S2◇	Stack buffer overflow	✓	✓
PX4 drivers	S3†	Null pointer de-reference	✓	✓
PX4 drivers	S4‡	Null pointer de-reference	✓	✓

Bug reports: * https://goo.gl/C4VmY6 ◇ https://goo.gl/Q7HkDz
† https://goo.gl/9uB85o ‡ https://goo.gl/VBFF7K

2) Micro-Benchmarks: The latency of init is 10 μsec and it does not have an impact on the overall performance of the MCS since it is invoked once only when the system bootstraps. The overhead of `switch_view` is 15 μsec and it mainly comes from searching for the right memory view of a process and configuring the MPU through MMIO. `read_scb` and `write_scb` both have small overheads, 4 and 5 μsec respectively, from checking whether the current access control rules allow the requested access to the SCB and performing the access if it is allowed.

B. Security Experiments

1) Memory Corruption Bugs: While working on our prototype, we found 4 new memory corruption bugs in the firmware of the MCS and reported them to the developers of the firmware modules (the NuttX RTOS and PX4 drivers). Table II summarizes the bugs. We detected S1 and S2 using a source code analysis tool [9] and S3 and S4 *using our initial memory protection*, respectively. *All of the bugs have been confirmed and patched by the developers in response to our reports.*

S1 is a memory corruption bug in the RTOS's system console that can be reproduced by an arbitrarily long input string to the terminal. When the program copies the user input to a fixed-size global buffer it does not check the length of the input, which may cause a buffer overflow.

S2 is a stack buffer overflow bug in a PX4 device driver. Similar to S1, one of the device driver functions calls the `sscanf` library function to copy an input string to a fixed-size local buffer without a proper length check. Unlike S1, S2 can be exploited as a vulnerability more easily since it causes a stack overflow. Specifically, an attacker may call the vulnerable function with a carefully crafted input string to overwrite the return address in the stack and this can be used to execute an injected code or as the basis of code-reuse attack. In our evaluation with 8 attack cases, we exploit this vulnerability to hijack the control flow and then execute the malicious payloads.

S3 and S4 are both null pointer de-reference bugs caused by a PX4 device driver. This driver implements

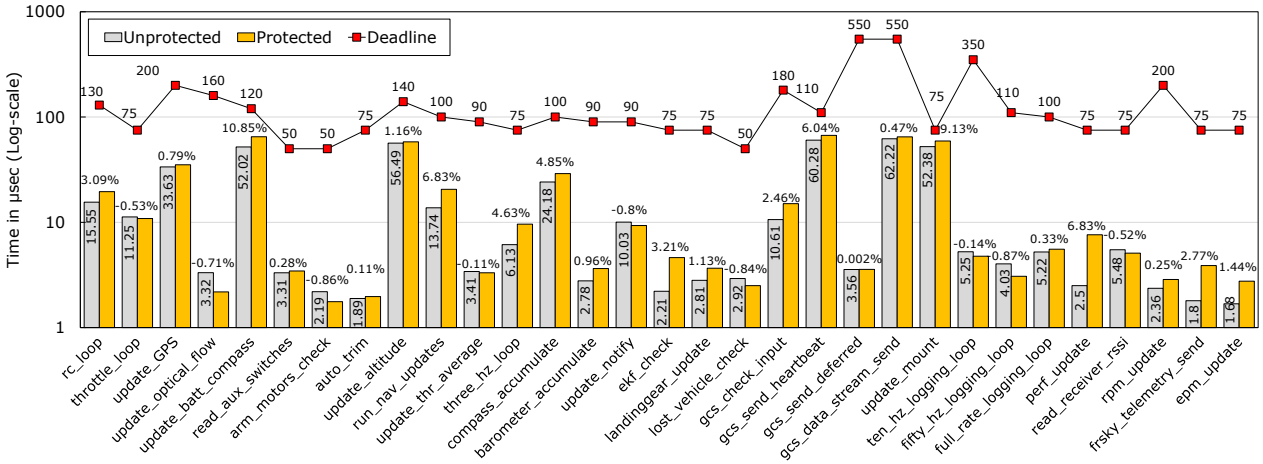


Fig. 8. Performance impact of MINION on real-time tasks with deadline constraints. The overhead introduced by MINION is marked on top of every bar that represents the execution time with MINION. The results are the average of more than 100 runs.

three functions, namely `cpuload_initialize_once`, `sched_note_stop`, and `sched_note_switch`. The RTOS calls these functions to let the driver measure the CPU usage of each real-time process. Although `sched_note_stop` and `sched_note_switch` are designed to be called only after the initialization is completed by `cpuload_initialize_once`, we found a certain case that the RTOS calls the two functions before `cpuload_initialize_once`. If this happens, `sched_note_stop` (S3) and `sched_note_switch` (S4) de-references uninitialized null data pointers without checking if the pointers are initialized.

Interestingly, we did not find these bugs manually. MINION *detected the bugs when we applied our initial memory protection to the MCS*. Because the real-time processes that use the driver did not have access to the memory region at address zero in its reduced memory view, MINION detected the null pointer de-references as memory access violations. These bugs were hidden until we found them because the MCS did not provide any memory protection and the null pointers were silently referenced without a visible symptom for a long time.

2) Attack Cases: Due to the absence of a known real-world attack on the specific MCS we tested, we created a set of new attack cases based on the real-world attacks against other similar real-time MCS [12], [32], [33], [39], [51]. All 8 attacks that we created are based on the exploitation of the S2 memory corruption bug (Table II) within a victim real-time process (i.e., `ver`), which outputs the version information of the software and hardware modules of the MCS. Once the vulnerability is triggered, the return address is modified such that an attack payload function is invoked when the vulnerable returns. To summarize, whereas the MCS without MINION were vulnerable to all of these cases, the MCS with MINION successfully detected all (shown in Table III). We include the number of real-time processes that have the attack surfaces in their memory view (P), to clearly represent the effectiveness of MINION's attack surface reduction from the attacker's perspective. Overall, out of a total of 49 processes, only zero to five processes contain the attack surfaces exploitable by attackers. Zero process indicates that none of the processes perform MMIO operations on these devices since they are accessed only by the specific RTOS functions for hardware initialization and real-time process scheduling. In the following, we describe the details of these attacks.

Process Termination. The RTOS provides a set of POSIX functions for the convenience of application, library, and device driver developers who are familiar with Unix-like systems. The `kill` function is supported by the RTOS and it can be used to terminate an arbitrary process. Similar to a *return-to-libc* attack, our process termination attack reuses `kill` to terminate the flight controller process (`main_loop`) that runs the key real-time operations of the MCS. The attack overwrites the return address of the vulnerable function in the victim process's stack memory with the address of `kill` and modifies the hardware register to terminate the target process. The attack was successful because the victim process had free access to any function in the firmware including the `kill` function. With the protection of MINION, this attack was detected (and the UAV safely landed by the fail-safe landing feature) because the memory view of the victim process did not include `kill` that the process should not use in benign operations.

Servo Operation. The servo system of 3DR IRIS+ runs on a separate co-processor whereas the firmware runs on the main microprocessor. A PX4 device driver in the firmware has a function (called `up_pwm_servo_set`) to send signals to the servo system through general-purpose I/O (GPIO). Similar to the two previous attacks, our servo operation attack hijacks the control flow of the victim process and invokes `up_pwm_servo_set` to maliciously control the four actuators of the UAV. In our experiment, we confirmed that the attack can easily disrupt the flight by invoking the servo controlling function with the two argument values (`channel` and `value`), which represent different types of UAV movement/mode and the speed of the actuators, respectively. The attack was detected and prevented by MINION since `up_pwm_servo_set` is not reachable from the victim process's entry functions and thus the memory view did not include it.

Control Parameter Attack. The flight controller adjusts control variables for the desired control response. The *control parameters* used in the controller are constant values which determine the performance of the controller for the optimum

TABLE III. ATTACK CASES USED TO EVALUATE THE EFFECTIVENESS OF MINION. ✓: DETECTED. ✗: UNDETECTED. S: SIZE OF THE ATTACK SURFACE IN BYTES. P: NUMBER OF THE PROCESSES THAT CONTAIN THE ATTACK SURFACE IN THE MEMORY VIEWS OUT OF TOTAL 49 PROCESSES.

Attack Case	Attack Surface					Detection	
	Type	Name	Memory Area	S	P	Without MINION	With MINION
Process Termination*	RTOS function	`kill`	Code	78	2	✗	✓
Servo Operation	Driver function	`up_pwm_servo_set`	Code	84	5	✗	✓
Control Parameter Attack◇	Control parameter	`pid_rate_roll`	Data (Global)	4	1	✗	✓
RC Disturbance†	RC configuration data	`channel_roll`	Data (Heap)	4	1	✗	✓
Soft Timer Attack‡	SysTick hardware timer	`STK_LOAD`	Device	4	0	✗	✓
Hard Timer Attack⋆	SysTick hardware timer	`STK_LOAD`	Device	4	0	✗	✓
Memory Remapping	Flash patch and breakpoint unit	`FP_REMAP`	Device	32	0	✗	✓
Interrupt Vector Overriding	Interrupt vector table	`VTOR`	Device	4	0	✗	✓

Demo videos: * https://goo.gl/1aSEf1 ◇ https://goo.gl/sJRPFg † https://goo.gl/5gAuvs ‡ https://goo.gl/jFHgYL ⋆ https://goo.gl/dMQ2YZ

flight behaviors and stability. Therefore, the control parameters are carefully chosen with a tuning process to find the optimum values before the actual flight. If the control parameters are chosen incorrectly, the behavior of the UAV can become unstable. In our target MCS, the parameters are stored in the read-only memory (EEPROM) first, and then loaded into the shared memory (SRAM) as *global* data objects when the controller process is initialized. Our control parameter attack is a non-control data attack [29] that corrupts one of the critical control parameters (`RATE_ROLL_P`) in the memory space. This parameter is used as the coefficient for *the roll control*, the control of the rotation around the axis that runs from the nose to the tail. Since the coefficient decides the strength of the UAV's reaction to angular change, maliciously updating the parameter with an incorrect value in the memory would have a negative impact on the stability of flight. In our experiment, the attack overwrote the original global value (0.15) with a considerably high value (15.0) in the shared memory space from the victim process while the UAV is in the air. The effect was that the UAV immediately started oscillating, reacting to the roll change with excessive response and overshoot repeatedly. Under MINION's protection, however, the same attack did not take effect since the victim process does not access to the global data in normal operations and the view switcher could detect the illegal memory access using the memory view.

RC Disturbance. As another example of a non-control data attack, our RC disturbance attack corrupts two configuration values that the RC receiver relies on to handle RC signals. Specifically, the upper and lower bounds of RC channel values are allocated in the *heap* memory of the controller process. A real-time task (`rc_loop`) in the process reads these values to constrain the incoming RC channels value within the bounds. By overwriting the bounds (originally 1100 and 1900) with incorrect small values (0 and 2) in the heap memory of the controller process, our RC disturbance attack disrupts the UAV's handling of the RC signals while in the air. Among different RC channels, our attack targets *the roll channel*, through which the signals are used to control the roll behaviors of the UAV. When the attack was launched, the UAV tilted toward one side and kept flying toward that side until it crashed. In contrast, when we launched the attack under MINION's protection, it could successfully prevent the attack by detecting the writes to the heap memory across the boundary between the victim and controller processes. Under the protection of MINION, each process only has access to its own stack and heap memory to comply with the memory view.

Soft Timer Attack. SysTick is a 24-bit system timer embedded in ARM Cortex-M. While the MCS is bootstrapping, the RTOS configures the timer by writing to the memory regions to which a number of SysTick registers are mapped, including the SysTick reload value register (`STK_LOAD`). The RTOS relies on `STK_LOAD` for the scheduling of real-time processes. Specifically, SysTick raises timer interrupts at the interval that `STK_LOAD` stores and the real-time scheduler works based on the timer interrupts. Our timer attacks maliciously modifies the interval value in `STK_LOAD`. For the *soft* timer attack, we increase the value to 2X of the original value (0x2903F) set by the RTOS. This degrades the responsiveness of the real-time processes since the scheduler now works based on "a slower clock." In our experiment, the UAV could not stabilize itself and quickly dropped its altitude when the attack was deployed. The attack was detected and prevented by MINION since SysTick registers are mapped to memory regions to which the victim process does not have access in its memory view.

Hard Timer Attack. The *hard* timer attack works in the same fashion with the soft timer attack. The only difference is that the hard timer attack writes the maximum 24-bit interval value (0xFFFFFF) to `STK_LOAD`. When the attack was deployed, we completely lost our control of the UAV. The flight controller kept increasing the altitude of the UAV and it did not respond to our RC controller. It would have been a critical incident if we did not use a tether to restrict the UAV's flight range. We analyze that the extremely low responsiveness of the real-time processes, caused by the attack, prevented the flight control and RC signal handling operations from being executed, so the servo system was running based on the last signal that the flight controller sent before the attack. As it did for the soft timer attack, the protection of MINION prevented this attack by detecting an illegal access to `STK_LOAD` out of the memory view of the victim process.

Memory Remapping. ARM Cortex-M3 and M4 processors have a feature to support debugging and run-time firmware updates, called Flash Patch and Breakpoint (FPB). This allows changing the default address space of the physical memory such that some portion of the flash ROM, which contains code, can be re-mapped to the SRAM. It also enables inserting hardware breakpoints into the code since the SRAM is mutable at run-time while the flash ROM is not. Our memory remapping attack exploits FPB to inject a malicious code into the target MCS. In our experiment, we replaced the `printf` function with a new function that outputs a certain string ("hacked") to the terminal after remapping the memory region. The remapping is accomplished by modifying the `FP_REMAP` register through MMIO. After the attack was deployed, the MCS repeatedly printed the string whenever `printf` was called in the system. This attack was detected by MINION since the victim process

did not have access to FP_REMAP in the tailored memory view.

Interrupt Vector Overriding. The firmware contains an interrupt vector table at a specific location of the physical address space, so the processor can execute the corresponding interrupt handler when an interrupt occurs. Our interrupt vector overriding attack swaps the original vector table with a new malicious vector table at run-time in order to steal hardware interrupts. The attack is accomplished by modifying the vector table offset register (VTOR) through MMIO, which contains the address of the vector table. The processor references VTOR when it delivers interrupts. In our experiment, we overrode the *reset* interrupt handler so that our attack payload (printing "hacked") becomes the first code that the MCS executes when it restarts. Similar to other attacks, MINION successfully detected the access to VTOR since it is out of the memory view assigned to the victim process.

C. Memory Space Reduction

We evaluate the effectiveness of MINION in reducing the memory space of real-time MCS. Without MINION, an attacker who successfully compromises a real-time process in the MCS can target any memory region in the entire physical address space. From the security perspective of 3DR IRIS+ that we evaluate, such memory areas include: code and read-only data areas mapped to the flash ROM, the data areas (global, stack, and heap) mapped to the SRAM and CCRAM, and other areas that are mapped to various peripheral devices and the SCB. The size of each memory area is retrieved from the hardware specifications. As a result, we use the total memory space of 2.9 MB as the baseline of our evaluation. Note that this is arguably conservative and fair estimation in comparison with the 4GB memory space that the 32-bit microprocessor can address, as our baseline only includes mapped memory areas remain active at run-time.

Fig. 9 shows how much memory space is reduced for each of the 49 processes in the MCS while Table IV presents the average sizes and reduction rates of the memory spaces. We combined the stack and heap sizes together (Stack+Heap) as their relatively small values make them unrecognizable in Fig. 9 and there is no difference between the stack and heap from the memory allocator's point of view (§IV-D). Our real-time MCS is equipped with an ARM Cortex-M4 processor that only supports 8 MPU regions. MINION performs clustering on the memory views to fit the memory views into the small MPU regions. For a comparison, we present the memory space reduction rates of MINION based on 8 MPU regions (Fig. 9a) and an unlimited number of MPU regions which current processors do not yet support (Fig. 9b). We discuss MINION's memory space reduction based on a varying number of MPU regions in §VI-D.

Overall, 76% and 93% of the memory spaces on average are reduced by MINION either with 8 MPU regions and with unlimited MPU regions, respectively. These are significant reduction rates, compared to the large baseline memory space of the MCS without the protection. More importantly, this is achieved with *zero impact on the real-time constraints* of the MCS, as shown in §VI-A.

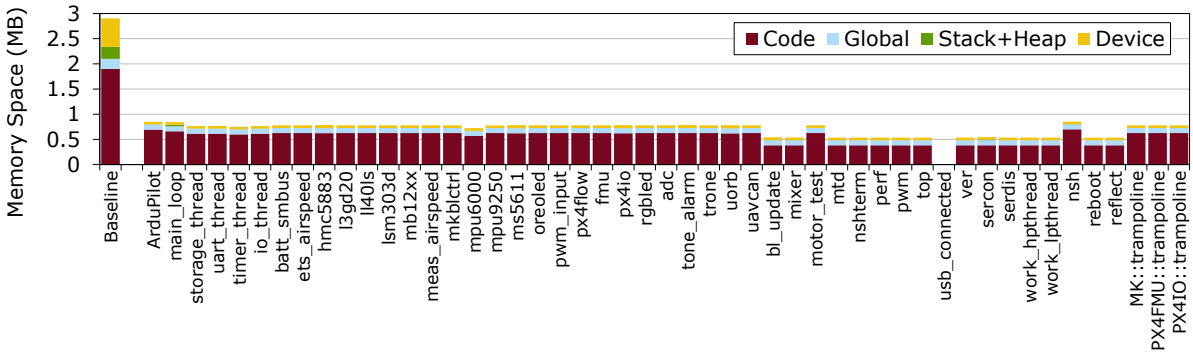
We note that the reduction rate of the code area is affected by the hardware limitation more than other areas—23% difference between the code reduction rates (Code) with and without the limitation of MPU regions (L and U) in Table IV. Our analysis found that code blocks used for each process are sparsely located in the address space, resulting in more conservative clustering of the memory blocks (i.e., code blocks used for other processes) compared to other memory areas. In addition, the global data area is reduced less than other areas even without the hardware limitation—62% reduction rate (Global) with an unlimited number of MPU regions (U) in Table IV. Based on our observation, we learned that a comparatively large portion of the global data is necessary to run normal operations of the processes in the MCS, resulting in a less significant reduction rate compared to other areas.

D. Hardware Support

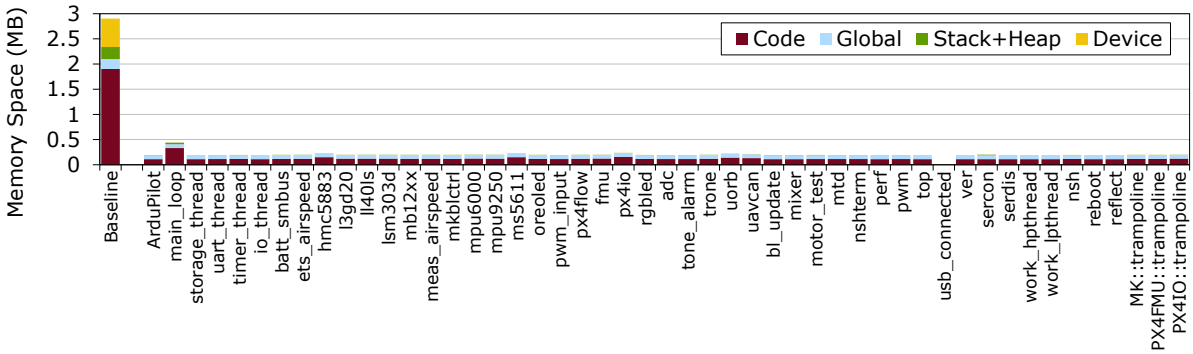
MINION performs a k-means clustering on a memory view to be compatible with today's MPUs that only support a limited number of MPU regions (§II-B). We show the rate of memory space reduction that MINION provides with an increasing number of MPU regions from 2 to 100 (Fig. 10). We note that the rate significantly grows as the number of MPU regions increases up to 8 and gradually converges to the maximum rate (i.e., the rate without the hardware limitation) after that point. Current microprocessors support either 8, 12, or 16 MPU regions depending on the hardware model and configuration. Based on the result, we learn that MINION can reduce a substantial amount of memory space (76%-80% in our prototype) under the limitation of the current hardware, and it will provide a higher reduction rate (93%) without the limitation. We hope that the results are beneficial to future designs of ARM processors and more secure real-time MCS, in terms of cost and security.

VII. RELATED WORK

Memory Isolation on Microcontrollers. There have been frameworks [21], [28], [42], [48], [52] to isolate sensitive programs from other programs in small embedded devices based on microcontrollers. Sancus [48] and SPM [52] provide trusted computing environments for resource-constrained embedded devices by extending hardware and annotating sensitive programs. TrustLite [42], TyTan [28], and Mbed uVisor [21] provide sandboxing between different programs in an MCS using an MPU. Although these frameworks can be used as primitives to isolate an application process from other program entities, they require manual efforts to designate the memory regions that belong to each application in the device's memory space and the programs to be developed based on the knowledge. EPOXY [31] addresses this problem by automatically identifying sensitive operations and instrumenting them with an SVC (for "supervisor call") to escalate the privilege mode for access control using an MPU. Unfortunately, these sensitive operations include privileged instructions and MMIO accesses to peripherals, which are executed at a very high frequency in MCS. Such a downside makes it unsuitable to be applied to real-time MCS where performance constraints are as important as security. In comparison, MINION focuses on reducing the memory spaces of real-time processes, which were entirely open to attackers previously, without sacrificing the performance constraints. The



(a) 8 MPU regions (ARM Cortex-M).



(b) Unlimited number of MPU regions.

Fig. 9. Memory space reduction in 3DR IRIS+, compared with the baseline memory space without MINION.

TABLE IV. SUMMARY OF MEMORY SPACE REDUCTION IN 3DR IRIS+. B: BASELINE MEMORY SPACE OF THE MCS WITHOUT MINION. L: REDUCED MEMORY SPACE USING 8 MPU REGIONS (ARM CORTEX-M). U: REDUCED MEMORY SPACE USING UNLIMITED NUMBER OF MPU REGIONS.

Type	Code		Global		Stack+Heap		Device		Total	
	Size (Bytes)	Reduction	Size (Bytes)	Reduction	Size (Bytes)	Reduction	Size (Bytes)	Reduction	Size (Bytes)	Reduction
B	1,993,588	-	207,128	-	250,612	-	592,148	-	3,043,476	-
L	572,149	71.30%	101,439	51.03%	2,851	98.86%	53,425	90.98%	729,863	76.02%
U	126,803	93.64%	78,157	62.27%	2,851	98.86%	392	99.93%	208,203	93.16%

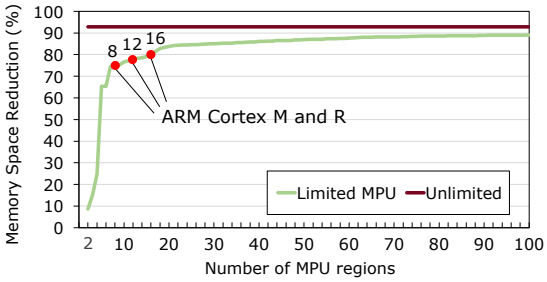


Fig. 10. Memory space reduction that MINION provides with an increasing number of supported MPU regions, compared to the result with unlimited MPU regions. The results are based on the average memory space of all real-time processes in Fig. 9.

unique architecture of MINION allows real-time processes to access code, data, and peripherals with zero overhead for benign operations while preventing illegal memory accesses across the memory view boundaries.

Compartmentalization and Isolation. There has been a body of research that decomposes programs into multiple isolated components in order to limit attack damage in conventional systems. Shreds [30] provides programming primitives to isolate fine-grained program segments from other segments against adversaries in the same process. SMV [41] is a programming abstraction that divides the virtual memory space of a process into multiple domains and enforces per-thread access control on these domains with least privilege. SOAAP [38] is an LLVM-based tool that allows developers to reason about various compartmentalization trade-offs using source-code annotations. Motivated by these works, MINION brings the concept of compartmentalization into MCS to identify and enforce the process boundaries (i.e., memory views) in the physical memory space where memory blocks of all processes mingle together.

Memory Space Reduction. There is a line of research that reduces the memory footprint of monolithic OS kernels to reduce the attack surface. Several researchers have proposed a kernel reduction approach that automatically generates compile-time configurations based on expected workloads [44], [50]. DRIP [37] eliminates malicious logic from a trojaned kernel driver by iteratively trimming away unnecessary code from the based on off-line profiling. Besides these off-line kernel reduction works, kRazor [43] is an OS mechanism that restricts accesses to kernel code from user-level applications based on run-time profiling of workloads. FACE-CHANGE [36] identifies the minimized kernel memory for each application based on run-time profiling and projects the memory while the application is in production using virtualization techniques. All these works

on monolithic kernels have motivated us to apply the principle of least privilege to real-time MCS, which was previously absent due to practical hardware and performance constraints.

Remote Attestation of Embedded Devices. There have been many solutions to improve the security of general embedded devices through memory isolation. Remote attestation techniques on low-cost embedded systems [24], [34], [35] focus on verifying the integrity of firmware using a trusted software or hardware anchor while meeting requirements for the device. MINION is complementary to these techniques and they can be leveraged to ensure the integrity of the view switcher against off-line code injection and modification attempts.

ARM TrustZone. It has recently been announced that TrustZone [6] will be supported by the new ARMv8 architecture for microcontrollers [13]. TrustZone provides a hardware method to divide program execution between trusted and untrusted execution environments. We believe MINION can be integrated with TrustZone, because its architecture is general and the implementation of the view switcher is compact. Our view switcher could be placed in the trusted environment in the strongest possible isolation from the RTOS and real-time processes running in the untrusted environment. Using techniques similar to [25], the view switcher can interrupt context switching events across the boundary between the two environments while keeping both its effectiveness and the strength of isolation.

VIII. DISCUSSION AND FUTURE WORK

Memory isolation and privilege separation are well-established security properties in conventional systems. The goal of MINION is to realize these properties in real-time MCS as closely as possible without sacrificing the performance constraints. As a result, MINION significantly reduces the chance of launching successful attacks through memory space, which was entirely open to adversaries in the past, while preserving the usability of these systems.

While achieving its goal, however, MINION does not provide the strongest notion of memory isolation, compared to traditional memory isolation in conventional systems, such as Linux and Windows. Specifically, memory views that MINION enforces may still contain small open windows that adversaries can abuse, due to its over-approximation on memory views. This implies that, if an attacker can abuse such a window, which is unprotected by MINION, he/she may be able to subvert other mission critical modules in the real-time MCS. For example, an attacker capable of identifying shared data between two processes may exploit the data to subvert one of the processes after compromising the other.

Such over-approximation issues for real-time MCS stem from the fundamental challenges in: (1) performing a precise program analysis and (2) the limited capability of current hardware protection units. It is well known that performing a program analysis with both completeness and soundness guarantees is challenging. From MINION's context which performs a points-to static analysis, leveraging the state-of-the-art static/dynamic analysis techniques in the future would decrease the over-approximation to close potential security holes. In addition, as we discussed in §VI-D, the over-approximation caused by the hardware limitation will hopefully be improved by supporting more number of MPU regions in the future microprocessors.

We would like to further emphasize that, without MINION, the attack surface was completely open to attackers. With MINION, however, we demonstrated that MINION is able to prevent realistic attacks while satisfying the time constraints on a commodity real-time MCS. Furthermore, our evaluation showed that over 75% of the attack surface is reduced by MINION even with the hardware limitation and over 90% without it, compared to the unprotected memory space.

IX. CONCLUSION

It is challenging to balance between the performance and security requirements of real-time MCS. Due to performance constraints, standard and state-of-the-art security mechanisms in the conventional computer systems are not commonly adopted in these systems. We have presented MINION, a new security architecture that bridges the security gap by bringing memory isolation and privilege separation into real-time MCS. By reducing the memory spaces of real-time processes which were entirely open to attackers previously, we demonstrated that MINION can effectively neutralize various malicious attacks in different attack vectors through memory view enforcement. Lastly, we showed that the lightweight design of MINION maintains the responsiveness of real-time MCS at its original level while significantly reducing the attack surface in the memory space.

ACKNOWLEDGMENT

We thank our shepherd, Ang Chen and the anonymous reviewers for their valuable comments and suggestions. This work was supported, in part, by ONR under Grant N00014-17-1-2045. Any opinions, findings, and conclusions in this paper are those of the authors and do not necessarily reflect the views of the ONR.

REFERENCES

[1] "CVE-2002-1633," 2002, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-1633.

[2] "CVE-2002-2041," 2002, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2002-2041.

[3] "CVE-2005-3928," 2005, https://cve.mitre.org/cgi-bin/cvename.cgi?name=2005-3928.

[4] "CVE-2006-0621," 2006, https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2006-0621.

[5] "Ftrace: Function Tracer," 2008, https://www.kernel.org/doc/Documentation/trace/ftrace.txt.

[6] *ARM Security Technology - Building a Secure System using TrustZone Technology*. ARM Limited, 2009.

[7] *ARMv7-M Architecture Reference Manual*. ARM Limited, 2010.

[8] *Cortex-M4 Technical Reference Manual*. ARM Limited, 2010.

[9] "Cppcheck - A tool for static C/C++ code analysis," 2010, http://cppcheck.sourceforge.net/.

[10] *Cortex-R4 and Cortex-R4F Technical Reference Manual*. ARM Limited, 2011.

[11] "Express Logic Introduces Memory-Protected Application Modules for ThreadX RTOS," 2011, http://www.eetimes.com/document.asp?doc_id=1316596.

[12] "Hijacking drones with a MAVLink exploit," 2015, http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.

[13] *ARMv8-M Architecture Reference Manual*. ARM Limited, 2016.

[14] "FreeRTOS-MPU," 2016, http://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[15] "Open Source for Drones - PX4 Pro Open Source Autopilot," 2016, http://px4.io/.

[16] "3DR IRIS+," 2017, http://3dr.com/support/articles/207358106/iris/.

[17] "3DR Pixhawk," 2017, http://3dr.com/support/articles/207358096/3dr_pixhawk/.

[18] "ArduPilot," 2017, http://ardupilot.org/.

[19] "ArduPilot: ArduPlane, ArduCopter, ArduRover source," 2017, https://github.com/ArduPilot/ardupilot.

[20] "Mbed OS," 2017, https://www.mbed.com/en/development/mbed-os/.

[21] "Mbed uVisor," 2017, https://www.mbed.com/en/technologies/security/uvisor/.

[22] "NuttX Real-Time Operating System," 2017, http://nuttx.org/.

[23] "UAVCAN," 2017, http://uavcan.org/.

[24] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: Control-Flow Attestation for Embedded Systems Software," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[25] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Scottsdale, AZ, Nov. 2014.

[26] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATC)*, Anaheim, CA, 2005.

[27] E. Bertino, "Data Security and Privacy: Concepts, Approaches, and Research Directions," in *Proceedings of the 40th IEEE Computer Society International Conference on Computers, Software & Applications (COMPSAC)*, Atlanta, GA, Jun. 2016.

[28] F. Brasser, B. El Mahjoub, A.-R. Sadeghi, C. Wachsmann, and P. Koeberl, "TyTAN: Tiny Trust Anchor for Tiny Devices," in *Proceedings of the 52nd Annual Design Automation Conference (DAC)*, San Francisco, CA, Jun. 2015.

[29] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-Control-Data Attacks Are Realistic Threats," in *Proceedings of the 14th USENIX Security Symposium (Security)*, Baltimore, MD, Aug. 2005.

[30] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-Grained Execution Units with Private Memory," in *Proceedings of the 2016 IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2016.

[31] A. A. Clements, N. S. Almakhdhub, K. S. Saab, P. Srivastava, J. Koo, S. Bagchi, and M. Payer, "Protecting Bare-Metal Embedded Systems with Privilege Overlays," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (S&P)*, San Jose, CA, May 2017.

[32] D. Davidson, H. Wu, R. Jellinek, V. Singh, and T. Ristenpart, "Controlling UAVs with Sensor Input Spoofing Attacks," in *Proceedings of the 10th USENIX Workshop on Offensive Technologies (WOOT)*, Austin, TX, Aug. 2016.

[33] E. Deligne, "ARDrone corruption," *Journal in Computer Virology*, vol. 8, no. 1, pp. 15–27, 2012.

[34] K. Eldefrawy, G. Tsudik, A. Francillon, and D. Perito, "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust," in *Proceedings of the 19th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2012.

[35] A. Francillon, Q. Nguyen, K. B. Rasmussen, and G. Tsudik, "A Minimalist Approach to Remote Attestation," in *Proceedings of the Conference on Design, Automation & Test in Europe (DATE)*, Dresden, Germany, Mar. 2014.

[36] Z. Gu, B. Saltaformaggio, X. Zhang, and D. Xu, "FACE-CHANGE: Application-Driven Dynamic Kernel View Switching in a Virtual Machine," in *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Atlanta, GA, Jun. 2014.

[37] Z. Gu, W. N. Sumner, Z. Deng, X. Zhang, and D. Xu, "DRIP: A Framework for Purifying Trojaned Kernel Drivers," in *Proceedings of the 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Budapest, Hungary, Jun. 2013.

[38] K. Gudka, R. N. Watson, J. Anderson, D. Chisnall, B. Davis, B. Laurie, I. Marinos, P. G. Neumann, and A. Richardson, "Clean Application Compartmentalization with SOAAP," in *Proceedings of the 22Nd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, Denver, CO, Oct. 2015.

[39] K. Highnam, K. Angstadt, K. Leach, W. Weimer, A. Paulos, and P. Hurley, "An Uncrewed Aerial Vehicle Attack Scenario and Trustworthy Repair Architecture," in *The 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshop (DSN-W)*, Los Alamitos, CA, Jun. 2016.

[40] M. Hooper, Y. Tian, R. Zhou, B. Cao, A. P. Lauf, L. Watkins, W. H. Robinson, and W. Alexis, "Securing Commercial WiFi-Based UAVs From Common Security Attacks," in *Proceedings of 2016 IEEE Military Communications Conference (MILCOM)*, Baltimore, MD, Nov. 2016.

[41] T. C.-H. Hsu, K. Hoffman, P. Eugster, and M. Payer, "Enforcing Least Privilege Memory Views for Multithreaded Applications," in *Proceedings of the 23rd ACM Conference on Computer and Communications Security (CCS)*, Vienna, Austria, Oct. 2016.

[42] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices," in *Proceedings of the 9th European Conference on Computer Systems (EuroSys)*, Amsterdam, The Netherlands, Apr. 2014.

[43] A. Kurmus, S. Dechand, and R. Kapitza, "Quantifiable Run-Time Kernel Attack Surface Reduction," in *Proceedings of the 11th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, Egham, UK, Jul. 2014.

[44] A. Kurmus, R. Tartler, D. Dorneanu, B. Heinloth, V. Rothberg, A. Ruprecht, W. Schröder-Preikschat, D. Lohmann, and R. Kapitza, "Attack Surface Metrics and Automated Compile-Time OS Kernel Tailoring," in *Proceedings of the 20th Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2013.

[45] C. Lattner and V. Adve, "LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation," in *Proceedings of the International Symposium on Code Generation and Optimization (CGO)*, Palo Alto, CA, Mar. 2004.

[46] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, IL, Jun. 2005.

[47] N. Nethercote and J. Seward, "Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, San Diego, CA, Jun. 2007.

[48] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens, "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base," in *Proceedings of the 22nd USENIX Security Symposium (Security)*, Washington, DC, Aug. 2013.

[49] S. Pastrana, J. Tapiador, G. Suarez-Tangil, and P. Peris-L´opez, "AVRAND: A Software-Based Defense Against Code Reuse Attacks for AVR Embedded Devices," in *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, San Sebastián, Spain, Jul. 2016.

[50] A. Ruprecht, B. Heinloth, and D. Lohmann, "Automatic Feature Selection in Large-scale System-software Product Lines," in *Proceedings of the 2014 International Conference on Generative Programming: Concepts and Experiences (GPCE)*, Västerås, Sweden, Sep. 2014.

[51] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim, "Rocking Drones with Intentional Sound Noise on Gyroscopic Sensors," in *Proceedings of the 24th USENIX Security Symposium (Security)*, Washington, DC, Aug. 2015.

[52] R. Strackx, F. Piessens, and B. Preneel, "Efficient Isolation of Trusted Subsystems in Embedded Systems," in *Proceedings of the 6th International ICST Conference on Security and Privacy in Communication Networks (SecureComm)*, Singapore, Sep. 2010.

[53] Y. Sui and J. Xue, "SVF: Interprocedural Static Value-flow Analysis in LLVM," in *Proceedings of the 25th International Conference on Compiler Construction (CC)*, Barcelona, Spain, Mar. 2016.