# M2MON: Building an MMIO-based Security Reference Monitor for Unmanned Vehicles

Arslan Khan[†], Hyubgsub Kim[†], Byoungyoung Lee[*], Dongyan Xu[†], Antonio Bianchi[†], Dave (Jing) Tian[†]

[†]Purdue University, {khan253, kim2956, dxu, antoniob, daveti}@purdue.edu
[*]Seoul National University (SNU), byoungyoung@snu.ac.kr

## Abstract

Unmanned Vehicles (UVs) often consist of multiple Micro Controller Units (MCUs) as peripherals to interact with the physical world, including GPS sensors, barometers, motors, etc. While the attack vectors for UV vary, a number of UV attacks aim to impact the physical world either from the cyber or the physical space, e.g., hijacking the mission of UVs via malicious ground control commands or GPS spoofing. This provides us an opportunity to build a unified and generic security framework defending against multiple kinds of UV attacks by monitoring the system's I/O activities. Accordingly, we build a security reference monitor for UVs by hooking into the memory-mapped I/O (MMIO), namely M2MON. Instead of building upon existing RTOS, we implement M2MON as a microkernel running in the privileged mode intercepting MMIOs while pushing the RTOS and applications into the unprivileged mode. We further instantiate an MMIO firewall using M2MON and demonstrate how to implement a secure Extended Kalman Filter (EKF) within M2MON. Our evaluation on a real-world UV system shows that M2MON incurs an 8.85% runtime overhead. Furthermore, M2MON-based firewall is able to defend against different cyber and physical attacks. The M2MON microkernel contains less than 4K LoC comparing to the 3M LoC RTOS used in our evaluation. We believe M2MON provides the first step towards building a trusted and practical security reference monitor for UVs.

## 1 Introduction

Unmanned Vehicles (UVs), such as Unmanned Aerial Vehicles (UAV) and Unmanned Ground Vehicles (UGV), start to play an important role in our daily life. For instance, Amazon is planning to use drones for package delivery [13]. Since these systems are in continuous interaction with the physical world, they often consist of multiple Micro Controller Units (MCUs) as different peripheral devices, besides their own main MCUs. For example, a UV is usually equipped with a fail-safe module, a Wi-Fi module, a GPS module, actuators for controlling propellers, and different sensors for attitude control (such as gyroscope, accelerometer, barometer, telemetry radio, rangefinder, camera, etc.) [1]. These peripherals communicate with the main MCU using: 1) I/O registers that are mapped directly into the memory regions of the system, i.e. Memory Mapped I/O (MMIO), or 2) an external, memory-mapped, bus.

Unlike traditional computer systems, attacks against UV can happen from both the cyber and the physical worlds [23, 29, 30, 32, 54, 70]. For example, attackers can send out malicious ground control commands to crash a UV [30] via MAVLink [36], or spoof GPS to disrupt the road navigation [70] or hijack the flight mission of a UV. Existing defenses range from using cryptography [12,37,60] and runtime compartmentalization [16, 17, 29] to fingerprinting [14, 15] and physical modeling [21, 52]. Unfortunately, these security solutions are mainly designed with a dedicated threat model, and none of them can prevent UV from most of the attacks, let alone a unified security solution defending against all known UV attacks.

We observe that a number of UV attacks aim to impact the physical world, such as modifying the trajectory, destabilizing the UV [30], or simply crashing a UV [29]. All these attacks involve some form of malicious communication between different MCUs, thus they result in malicious I/O-level activities. This fact provides us an opportunity to build a unified and generic security framework defending against multiple kinds of UV attacks by monitoring the system's I/O.

Accordingly, we build a security reference monitor for UV by hooking into the MMIO layer, namely M2MON. Instead of building upon existing RTOS, we implement M2MON as a microkernel running in the privileged mode mediating every MMIO access from within the system while pushing the traditional RTOS and applications into the unprivileged mode. This design reduces the Trusted Computing Base (TCB) from 3M Lines of Code (LOC) of a commercial RTOS to less than 4K LoC of the M2MON microkernel.

Using M2MON, we further instantiate an MMIO firewall detecting intrusions with the UV, and demonstrate how to

implement a secure Extended Kalman Filter (EKF) within M2MON. We also provide a post-detection response mechanism within M2MON to gracefully handle attacks. We implement and evaluate M2MON on a real-world UV system. Our evaluation shows that the M2MON-based firewall is able to defend against different UV attacks with 8.85% runtime overhead without violating the system's software deadlines. We believe M2MON provides the first step towards building a trusted and practical security reference monitor for UV.

In summary, the contributions of this paper are as follows:

- **UV Attacks and Defenses Study.** Our UV attacks study encompasses various attacks on popular UV systems. The study shows that none of the existing security solutions can defend against all the attacks in the survey. In addition, it shows that all these UV attacks demonstrate I/O-level activities and even variances.
- **M2MON Design and Implementation.** Based on the observation above, we design and implement a security reference monitor able to mediate every MMIO access, namely M2MON. We implement M2MON as a microkernel running in the privileged mode while pushing the traditional RTOS and applications into the unprivileged mode. We further instantiate an MMIO firewall using M2MON and demonstrate how to implement a secure Extended Kalman Filter and post-detection response mechanism within M2MON.
- **M2MON Evaluation.** Our evaluation on a real-world UV system demonstrates that the M2MON-based firewall is able to defend against all the UV attacks mentioned earlier. This evaluation shows that M2MON introduces a low overhead (8.85%). At the same time, the usage of M2MON reduces the TCB from 3M LoC of a commercial RTOS to less than 4K LoC.

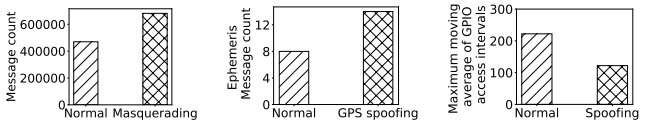
To further development in this direction, we made the source publicly available (https://github.com/purseclab/M2MON).

## 2 Motivation

Our hypothesis is that *for a UV attack to have a concrete effect, it needs to introduce some I/O activities and that these activities can be detected.* These I/O activities are due to the necessity for the attack to interact with peripheral MCUs to, ultimately, have an impact on the physical world.

To empirically prove this hypothesis, we select a series of UV attacks and reproduce these attacks on real-world systems, as shown in Table 1. Except for the CAN bus masquerading attacks, all the other attacks are tested on a 3DR IRIS+ UAV platform [1]. Further details on the attacks can be found in Section 6.1.

During this study, we found two previously unknown vulnerabilities related to the Wi-Fi module and a flight control program in the UAV, respectively. 3DR IRIS+ uses an




**(a)** Changed number of messages sent by BCM under the ECU attack.

**(b)** Changed number of GPS messages under the GPS attack.

**(c)** Changed the maximum moving average of the GPIO access interval (ms) under the radio attack.

**Figure 1:** Changed I/O access patterns under various attacks.

ESP8266 Wi-Fi module, which is a popular Wi-Fi module in both UAVs and IoTs. However, we noticed that the ESP8266 modules do not securely conduct the Over-The-Air (OTA) firmware update because they fail to check the integrity of the update [41]. This enables an attacker to conduct network-based attacks such as DNS cache poisoning [31], ARP spoofing [69], and/or Man-in-the-Middle attacks (MitM) [61,62] to flash malicious firmware to the Wi-Fi module. Once the Wi-Fi module is compromised, using MitM attack techniques, we conduct different attack scenarios such as flash patch attack, gyroscope attack, and barometer attack.

3DR IRIS+ also uses ArduPilot [7], a popular flight control program used by most UAV platforms. While studying the effects of GPS spoofing on I/O patterns, we found that the GPS module reports potential spoofing to the flight controller, but ArduPilot ignores the warnings from the GPS module.

During our study of these UV attacks, we confirm that we could observe unique I/O activities by monitoring the MMIO layer. For example, the main system performs some specific I/O accesses only during the booting phase, e.g., to set up timers and IRQ handlers. These specific I/O accesses should not happen again once the system is running until the timer or IRQ override attack happens. Similarly, we observe changes in the number of CAN messages received by the ECU under the CAN masquerading attack as shown in Figure 1a. We can also spot the changed pattern of I/O accesses for the GPS spoofing attack (details: Section 6.1) in Figure 1b and for the radio replay attack (details: Figure 6.1) in Figure 1c. In summary, this survey demonstrates the potential of MMIO-layer monitoring to defend against a variety of UV attacks.

While there exists work to tackle some of these attacks, each proposed defense mechanism works only on a subset of them. As summarized in Table 1 crypto-based methods [12, 37, 60] have been proposed to defend against CAN masquerading, GPS Spoofing and radio replay. However, such methods suffer from the overhead of heavy computations. Furthermore, they don't work well against other surveyed attacks such as the timer attack or malicious sensors. Compartmentalization solutions [16, 17, 29] can detect the Timer and the IRQ attacks but they are unable to detect other attacks such as spoofing and masquerading. Voltage and clock skew fingerprinting [14, 15] only applies to CAN bus environ-

| | Crypto [12,37,60] | Compart. [16,17,29] | Finger-Printing [14,15] | Physical Modeling [21,52] | I/O Activity |
|---|---|---|---|---|---|
| Timer Attack [29] | – | ■ | – | – | ■ |
| IRQ Override [29] | – | ■ | – | – | ■ |
| CAN Masquerading [32] | ■ | – | ■ | – | ■ |
| Radio Replay [54] | ■ | – | – | – | ■ |
| Malicious Sensor [23] | – | – | – | ■ | ■ |
| Flash Patch Attack [29] | – | – | – | – | ■ |
| GPS Spoofing [70] | ■ | – | – | – | ■ |
| Gyroscope Attack | – | – | – | ■ | ■ |
| Barometer Attack | – | – | – | ■ | ■ |

**Table 1:** Survey of existing UV attacks and defenses. We did not find any defense that can defend against all of the studied attacks. However, in all the attacks we noticed some I/O activities involved. ■ shows defenses that work against some particular attacks.

ment. Physical modeling [21,52] helps to detect anomalies from within sensors via building a model of the physical world, predicting the expected measurements based on histories. Unfortunately, none of the existing solutions could defend against all UV attacks, thus motivating the need for a generic and systematic defense for UV.

## 3 Security Model

We target a variety of UV attacks as shown in Section 2. Adversaries can launch these attacks simply by sending out malicious commands or spoofed messages via the network. They could also compromise a peripheral MCU (e.g., exploiting a vulnerability within the peripheral firmware) or installing a malicious component inside these devices. More importantly, these attacks, once compromising the UV, will impact the physical world via changing the system behavior, which will be reflected at the I/O level. A passive attacker staying stealthy and quiet without impacting the system behavior is out of the scope of our threat model.

Our Trusted Computing Base (TCB) includes the main MCU of a UV, the Memory Protection Unit (MPU) provided by the MCU, the bootstrap code to boot up the MCU (e.g., ARM Trusted Firmware [9]), and the code constituting M2MON and its plugins. We also assume a secure communication channel between the system owners and M2MON, allowing the owner to configure different security policies. Note that the RTOS and its applications are not inside our TCB, since our approach allows us to execute them in the unprivileged mode. In this paper, we consider side-channel attacks (such as timing attacks) and attacks resulting from a malicious control program, such as Stuxnet [33], out of scope.

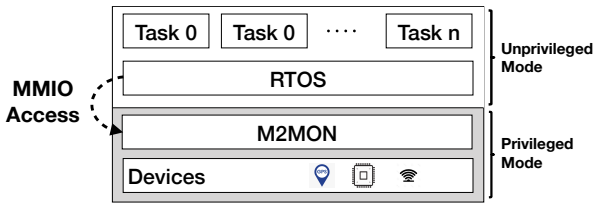



**Figure 2:** M2MON Microkernel Design: MMIO is configured as a privileged resource while moving the whole software stack to unprivileged mode. M2MON runs in privileged mode while managing the MMIO requests.

## 4 Design

The architecture of M2MON is shown in Figure 2. M2MON microkernel runs in privileged mode, mediating all MMIO accesses from different peripheral MCUs, separating itself from the RTOS and applications, and providing an interface to system owners for loading policies.

We start this section by explaining the design goals of M2MON (See Section 4.1), and finally demonstrate how we achieve these goals via trade-offs and optimizations (see Section 4.2).

### 4.1 Design Goals

Due to the intrinsic constraints and requirements of an embedded system environment, we need to face the following challenges: (1) no typical protection hardware available (e.g., MMU/IOMMU), (2) fragmentation of RTOS implementations per vendor/model (e.g., MBed, Zephyr, FreeRTOS, ThreadX, etc.), (3) RTOS and applications running in the privileged context together (to reduce context switches for performance considerations), and (4) no violation of the real-

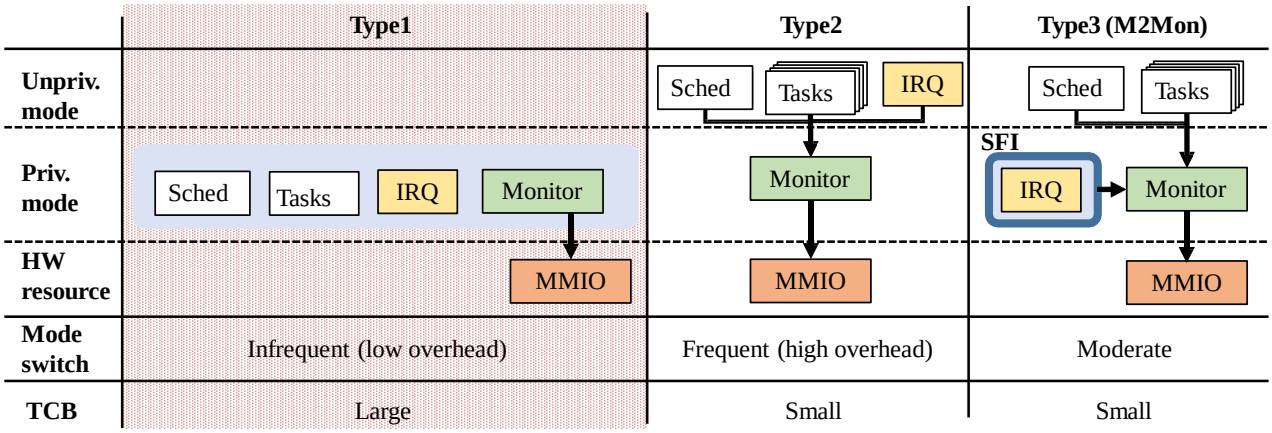



**Figure 3:** Different possible designs for MMIO reference monitor. **Type 1** monitors MMIO as a kernel service, however this does not satisfy reference monitor requirements. **Type 2** runs entire software stack in unprivileged mode at cost of performance. **Type 3 (M2MON)** runs most OS stack in unprivileged mode, with I/O intensive code in privilege mode inside a sandbox.

time requirements. To tackle these challenges, while providing strong security guarantees, we derive our design goals as follows:

- **G1 Complete Mediation** Our monitor should be able to mediate all MMIO accesses within the system. It has to be *non-bypassable* and *always invoked*.
- **G2 Tamperproofness** Our monitor needs to be tamperproof from threats and attacks outside the Trusted Computing Base (TCB). For instance, if we assume applications are not trusted and thus outside our TCB, we need to defend against attacks from them, as well as RTOS since they are often coupled together.
- **G3 Verifiability** Our monitor and the whole software TCB have to be small, e.g., comparing to typical RTOS implementations, thus allowing manual analyses and tests for verification.
- **G4 Generality** Our monitor cannot depend on a specific RTOS implementation. It should be general enough to be applied to any existing system.
- **G5 Programmability** Our monitor needs to provide a user interface enabling system owners to configure the policy and runtime behaviors as needed.
- **G6 Real-Time Satisfaction** Our monitor could only introduce a minimum runtime overhead, without violating the real-time requirements of the system.

The first three design goals are guaranteed by using a reference monitor [6]. However, our implementation goes beyond the reference monitor concept by considering practical deployment and runtime issues. The resultant system is a small microkernel running in privileged mode and mediating MMIO accesses at low overhead.

## 4.2 M2MON Micro Kernel

We now explain M2MON design. During M2MON design, we catered to the constraints specific to embedded systems. Using existing techniques such as SFI [65] and hardware extensions, we fulfilled each one of the aforementioned design goals. To evaluate our design, we ran and tested on real hardware.

**M2MON Isolation:** A naive design is to implement M2MON inside the RTOS, which runs in privileged mode and has control over all MMIO accesses, as shown in Type1 of Figure 3.

An RTOS includes a scheduler, tasks, and interrupt handlers (IRQs). As we mentioned earlier, applications are often running within a privileged context to reduce context switches. While this design is straightforward, it inevitably leads to having a TCB including both RTOS and applications, meaning that a vulnerability within an application might compromise M2MON. This design also heavily depends on the implementation of the RTOS, since M2MON is one of its components. To reduce the TCB size and get rid of the dependency of the RTOS implementations, we designed M2MON as a self-contained and single-purpose microkernel, running inside the privileged context.

Left with only two execution modes, we pushed both the RTOS and applications into the unprivileged mode, as shown in Type2 of Figure 3. This left the privilege execution mode for M2MON. Since the only task of this microkernel is mediating MMIO accesses, its codebase is small enough for manual analysis and testing, achieving the design goal **G3** *Verifiability*. Accordingly, this design supports running in different RTOS implementations, thus achieving **G4** *Generality*.

**MMIO Isolation and Protection:** Given a system memory map, we need to identify the MMIO regions and isolate them from other parts of the memory. Often, vendors declare memory regions associated with peripheral memory in the system memory map, using either device tree sources or technical

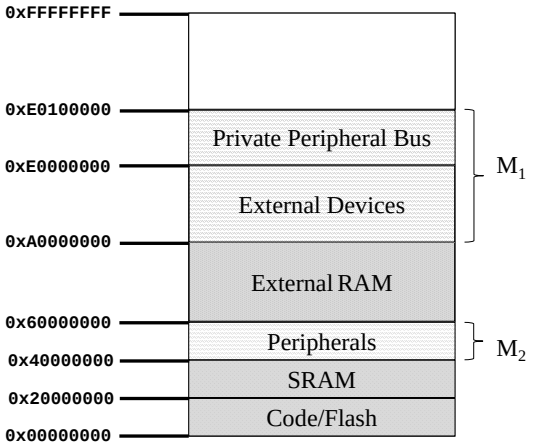



**Figure 4:** M2MON ARMv7-M Memory Layout. $M_1$ and $M_2$ are the sub-regions allowed by ARM for MMIO.

reference manuals. However, such memory regions also have to reside inside the specific regions defined by the architecture specification.

For example, all ARMv7-M compliant processors (which our experiment is based on) rely on the system memory map as shown in Figure 4. In this map, MMIO can be mapped into two clusters, annotated as $M_1$ and $M_2$. We call $M$ as the union on these two clusters (i.e., $M = M_1 \cup M_2$), representing the address range M2MON needs to protect. It is worth noting that addresses responsible for MMIO accesses may be sparsely populated, but all MMIO regions must reside within $M$.

By monitoring a superset of MMIO regions, we achieve the design goal **G1** *Complete Mediation*. We further configure MPU to forbid accesses to $M_1$ and $M_2$ from unprivileged mode, thus achieving **G2** *Tamperproofness* against RTOS and applications. Note that the MPU and the control registers can only be configured from the privileged context. Thus, in our design, they can only be accessed by M2MON.

**Interrupt Handlers Hardening:** The design described until now still presents a major performance and security drawback. Specifically, it cannot handle efficiently MMIO accesses coming from interrupt (IRQ) handlers. Because these handlers are running in unprivileged mode, they need two extra context switches whenever an MMIO access happens, making them a performance bottleneck.

To solve this issue, we need to move these handlers to the privileged mode. One solution would be implementing these handlers within M2MON directly. On one hand, different platforms often use different IRQs, and we might end up having to implement every IRQ handler available on the architecture to support different SoCs. On the other, the RTOS used by a UV already implements all the necessary handlers. For these reasons, we decided to reuse the IRQ handlers provided by the RTOS and move them back to the privileged mode to avoid duplication and improve performance.

```
1: int hrt_tim_isr() __attribute__((irqbox));
2: int hrt_tim_isr() {
3:   volatile unsigned * CR1_ADDR = 0x40012c00;
4:   uint32_t status = *CR1_ADDR;
5:   *CR1_ADDR = ~status;
     ...
```

```
1: int hrt_tim_isr() __attribute__((irqbox));
2: int hrt_tim_isr() {
3:   volatile unsigned * CR1_ADDR = 0x40012c00;
4:   uint32_t status = getreg32(CR1_ADDR);
5:   putreg32(~status, CR1_ADDR);
     ...
```

**Figure 5:** M2MON MMIO Detection using static analysis. Walking use-def chains we can find if a particular pointer is created using a hard-coded address. For all such pointers, we replace direct access with a call to our monitor gateway.

Note that these handlers need to access the MMIO as well and could contain vulnerabilities due to their complexities. Therefore, instead of reusing their code directly, we designed a sandbox mechanism for these handlers.

Our sandboxing mechanism uses a compilation-time analysis of the handlers' code and Software Fault Isolation (SFI) techniques. In this way, we can ensure that IRQ handlers cannot bypass M2MON monitoring and, at the same time, that their data-flow and control-flow integrity cannot be subverted, as shown in Type3 configuration in Figure 3.

*Complete Mediation Reassurance:* As mentioned above, M2MON needs to identify all MMIO accesses within an IRQ handler. To this aim, we observe that it is common for RTOS to rely on hard-coded address values to access MMIO, because such addresses are dictated by the hardware specification and cannot change. Particularly, the offset of registers within devices are given by 3rd-party manufacturers, whereas the base address of the device is selected by SoC manufacturers and cannot be changed if an MMU is not available.

Therefore, at compile-time, we perform use-def analysis [5] on the IRQ handlers source code, using the hard-coded MMIO addresses to locate all the instructions accessing MMIO. When an instruction is detected accessing MMIO addresses, it is replaced with a call to the M2MON monitor gateway. The M2MON monitor will take care of performing the original memory access, while, at the same time, enforcing the needed security policies. Figure 5 shows an example of how M2MON enforces such mediation in ArduPilot.

*Data-Flow Integrity:*

To sandbox the execution of interrupt handlers, we enforce the following policy: all data access from interrupt handlers should be restricted within the handler itself. In other words, interrupt handlers' code, although it runs in privileged mode should not be able to interfere with M2MON code.

To achieve this property, we analyze the memory layout of the target board (i.e., PixHawk FMU Board in our experiment) and mask all direct/indirect memory accesses to stay within

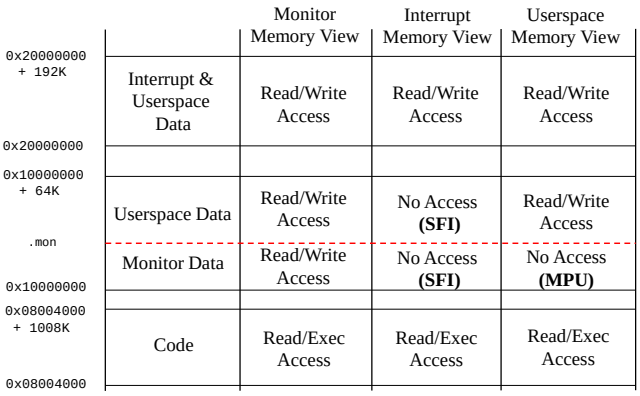



**Figure 6:** Memory Layouts fabricated using SFI and MPU. Each column shows the view for particular components in system. `.mon` is the section reserved for monitor. Each section shows which mechanism is used for isolation.

```
1: int dmainterrupt(int) __attribute__((irqbox));
2: int dmainterrupt(int irqno){
3:     struct dma_chan * = &gdma[irqno];
4:     int *channel = dma_chan->channel;
5:     ...
```

```
1: int dmainterrupt(int) __attribute__((irqbox));
2: int dmainterrupt(int irqno){
3:     struct dma_chan * = &gdma[irqno];
4:     dma_chan &= ~(1<<28);
5:     int *channel = dma_chan->channel;
6:     ...
```

**Figure 7:** M2MON Data Flow Integrity for M2MON monitor. All indirect accesses are instrumented so that the 28th bit is clear, ensuring sandbox cannot access `0x10000000` - `0x1FFFFFFF`.

the handlers. More specifically, Figure 6 shows the memory layout on PixHawk FMU Board. It has two RAM chips installed, *Closely Coupled SRAM (CCSRAM)* at `0x10000000` of size 64KB and an *SRAM* at `0x20000000` of size 192KB. We keep M2MON related data in a special section called `.mon` at start of CCSRAM, and further move interrupt handler related data to SRAM. Since the SRAM address range spans from `0x20000000` to `0x20030000`, we only need a logical AND with the address to clear bit 28 ensuring that interrupt handlers cannot access CCSRAM. Figure 7 shows an example of this application.

*Control-Flow Integrity:* To sandbox the execution of interrupt handlers we also need to sandbox their control flow. Specifically, we enforce the following policy: All instructions executed from interrupt handlers should belong to the interrupt handler itself. In other words, interrupt handlers' code, although it runs in privileged mode, should not be able to jump to M2MON code.

To achieve this property, during compilation, we apply Control-Flow Integrity (CFI) [2] techniques to these handlers. Figure 8 summarizes our approach. Traditionally, CFI is

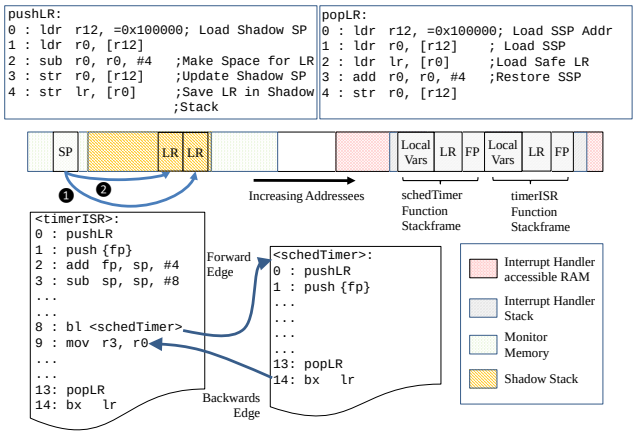



**Figure 8:** Example function branch with forward and backward control flow integrity in place. `pushLR` code snippet saves current LR on safe stack, while `popLR` restores from current safe stack pointer.

defined in terms of forward control (caller to callee branches, such as function calls) flow and backward control flow (callee to caller branches, such as return from function instructions). Our approach needs to take care of both forward and backward control flow.

To prevent forward control flow violations, we do not allow indirect branching using function pointers from handlers. For example, given any indirect branch/jump, we unroll it with all the potential targets (i.e., enumerating all potential targets using switch like statements), enabling branch verification at compile time.

To prevent backward control flow violations, we use a shadow stack [63]. Since only M2MON monitor can access its own data region (since we enforce Data-Flow Integrity, as explained in the previous section), we place the shadow stack in the monitor data section. We further modify the function epilogues and prologues used by the interrupt handlers to save the return address on entry and enforce the safe return address saved on the shadow stack on return.

It is worth noting that one may use SFI for all software modules (such as schedulers and tasks) and run them in privileged mode. However, besides bloating the TCB, employing SFI over all the system modules requires a large amount of code instrumentation, which would raise severe performance issues. For instance, in a system running the ArduPilot control software with NuttX RTOS, we measured 21,836 indirect references.

Conversely, the indirect references present in interrupt handlers are only 48. For this reason, by running only the interrupt handlers in privileged mode and applying SFI during compilation time only to their code, the speed overhead is minimal. This design choice allows us to achieve the design choice **G6** *real-time satisfaction* without compromising security guarantees. This solution can be applied to any compiler framework, independently of the RTOS implementation, thus

also achieving **G4** *Generality*.

To further reduce the overhead of MMIO accesses from the unprivileged mode, we leverage some key observations to group multiple MMIO accesses into a single syscall, reducing extra context switches. These optimizations help us achieve **G6** *real-time satisfaction*.

In particular, we notice that it is common to observe the following three MMIO accesses to the same MMIO address in order: read, modify, and write, composing a Read-Modify-Write (RMW) operation. RMW operations are commonly used to perform stateful interactions with peripherals, reading the state of the device, modifying the state, and finally updating the device state. Some devices export information as bitfields to users, and manipulating such bitfields also requires RMW operations. To coalesce these multiple MMIO requests, we design special syscalls that perform an RMW operation entirely. These syscalls are examples of flight controller code, which exercise such patterns, e.g., interrupt enabling/disabling routines, etc.

A similar case is communicating with other devices over external buses such as I2C and SPI, following a well-defined protocol. We call each transaction of such communication a *bus transfer*. After bus arbitration, data transfer starts through message packets using MMIO. A message packet could be as small as a byte depending on the bus payload capacity. As such, to transfer four bytes, four separated syscalls might be needed thus incurring extra context switches. We note that the sequence of operation is always the same (as it is defined by the bus protocol) and can be grouped into one syscall, taking in the device ID and the data to be transferred over the bus. For this reason, we design the `SPI_filter_transaction` syscall to transfer data over SPI, taking, as arguments, the Device ID for the device we want to communicate with, and two buffers for sending and receiving data together with their corresponding lengths.

**Hooks and Policy Enforcement:**

To allow end-users to customize M2MON and load policies during both compile-time and runtime, we design a set of user interfaces enabling both low-level API-based programming and high-level command-line-based management. This user interface achieves the **G5** *Programmability* design goal.

In particular, the low-level API-based programming interface allows owners to register and monitor actions upon certain I/O accesses for a given device. The device can be wither memory mapped (MMIO) or installed behind a bus (e.g., SPI).

The API consists of the following four functions:

```
typedef void (*EXEC)(uint size, bool is_write,
                     uint32_t value);
void register_action(uint addr, EXEC exec);

typedef void (*EXEC_SPI)(uint8_t *data);
void register_action_spi(uint device_id,
                         EXEC_SPI exec, bool egress);
```

```
typedef void (*EXEC_SYNC_CALL)(uintptr_t parm1,
              uintptr_t parm2, uintptr_t parm3);
void register_sync_call(uint call_id,EXEC_SYNC_CALL exec);
void sync_call(uint32_t call_id, uintptr_t parm1,
      uintptr_t parm2, uintptr_t parm3);
```

In `register_action`, `addr` determines the MMIO address to monitor, while `exec` is a callback function pointer invoked on each access. Inside the `exec` callback, `size` tells about the bit width of an access, `is_write` indicates whether the access is read or write, and `value` is the value to be written, which is only used in the case of write access (*i.e.,* `is_write` is true).

Similar to `register_action` for MMIO addresses, we design different APIs for a variety of buses to intercept the bus accesses. For instance, system owners can register an SPI filter using `register_action_action`, where `device_id` is the address of the device on SPI bus [1] for which data transfers will be monitored, `egress` selects the path of filtering (i.e., *Egress* monitors all data transfers from CPU to device over the bus, and *Ingress* monitors data transfers from device to CPU over the bus.), and `exec` is the callback invoked on each `data` transfer over the bus.

Lastly, to register synchronous callbacks in M2MON we also provide the API: `register_sync_call`. Unlike previous hooks which are only called on relevant MMIO access, synchronous callbacks can be triggered on demand. To achieve this, we provide `sync_call`. User can call `sync_call` with the relevant callback's id and parameters to trigger the service. This is similar to syscall machinery. In `register_sync_call`, `call_id` determines the id for the synchronous call, whereas `exec` is the callback invoked when `sync_call` is invoked with the `call_id` used to register this call. In `sync_call`, `parm1, parm2, parm3` are used to pass arguments to the relevant callback.

## 5 Implementations

We start with how we build the M2MON microkernel as a generic security reference monitor, followed by the MMIO-based firewall built upon M2MON and secure Kalman Filter implementations within M2MON. Both the firewall and the KF plugin are "applications" of M2MON and applied to our evaluation to demonstrate their usefulness and effectiveness.

### 5.1 M2MON Microkernel

To build M2MON, we use Minion [29] as the starting point. We modify the NuttX kernel to push the RTOS into the unprivileged mode while leaving the privileged mode for M2MON. To access an MMIO address, the unprivileged mode uses a supervisor call (SVC) to trap into the privileged mode, and M2MON checks the access against existing policies if any. For privileged mode, exception handlers are running inside

[1] Chip Select for SPI Devices

a sandbox enforced by DFI and CFI, and M2MON mediates every MMIO access from them as well. As Figure 6 shows, Cortex-M can possibly have 0.5G distinct MMIO addresses. Due to the scarcity of available memory and performance reasons, we implement a hash map to index different policies and rules quickly. We also port both NuttX and ArduPilot to GCC 6.3.1 for mature plug-in support.

To move user code into M2MON, we create a custom compiler attribute that users can annotate code with. To implement different SFI mechanisms within GCC, we wrote three passes, `pass_sanitize`, `pass_safe_stack` and `pass_epi_prologue_fixup`. We schedule our passes as early as possible so that we can take the full benefit from the subsequent optimization passes. `pass_sanitize` detects all MMIO operations using the algorithm described in Section 4.2. It ensures no direct MMIO accesses inside the sandboxed code, instruments all indirect references for DFI, and guarantees no indirect forward edges (branching using function pointer) in the code. This pass is scheduled right after the SSA (Static Single Assignment) pass and uses the alias analyses provided by GCC. However, alias analysis resulted in high false positives. Since GIMPLE [2] is machine-independent, we used two RTL [3] passes to implement safe stacks. `pass_safe_stack` is scheduled right after GIMPLE to RTL expansion passes, e.g., `pass_expand`. `pass_safe_stack` adds instructions at function entry and exit for safe stack upkeep. This is done before GCC generate the epilogue and prologue (`pass_pro_epi_fixup`). Since `pass_pro_epi_fixup` is scheduled after `pass_safe_stack`, we schedule another pass, `pass_epi_prologue_fixup`, to remove any manipulation to the link register (register used to save return address in ARM architecture) by compiler-generated function epilogue and prologue.

During the initial evaluation we found out that even though the regular control loop is CPU intensive, the startup phase of the firmware is I/O intensive. Monitoring I/O during the startup caused a significant latency in the initialization time. To overcome this latency, we delay the enforcement of the MMIO monitoring till the startup phase completes. Note that this workaround does not violate our security guarantees, because remote attackers cannot change the configurations during the initialization by modifying the flash memory without a USB flash programmer, which requires physical access to the drone.

## 5.2 Access Pattern Based Firewall

One key observation we make is to leverage the MMIO access patterns as "fingerprints" of peripheral MCUs. Comparing to other domains, the MMIO access pattern is fairly stable in a UAV environment. In fact, a UAV control program usually

[2]Language independent C-like IR used internally by GCC.

[3]Register Transfer Language, a LISP-like machine-dependent IR used by GCC.

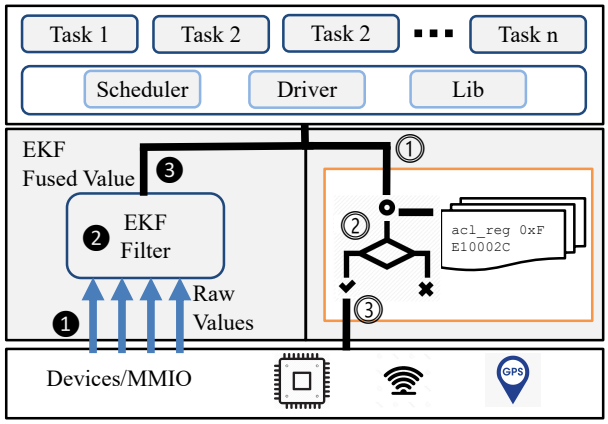



**Figure 9:** Retrofitting EKF module (left) and firewall (right). For EKF: 1) EKF module reads raw values from the sensor 2) processes them 2) and 3) provides a parameterized system call to update filters. For firewall: 1) Control loop issues an I/O requests 2) M2MON evaluates the request based on registered rules 3) If allowed the MMIO transaction is processed.

sets up its device configuration before the main control starts. Once inside the control loop, the program often follows the same flow, e.g., reading from sensor data registers, processing the input, and then writing back to actuator registers. During each loop, for each peripheral, only a limited number of MMIO addresses are accessed for data retrieval. Consequently, each device demonstrates a "fixed" MMIO access pattern under normal executions, and this pattern is repeated during every loop.

To get these MMIO patterns from the system, we use M2MON to log each MMIO access with a timestamp and extract patterns from these logs. An MMIO access pattern can reflect both spatial and temporal characteristics. Specifically, we consider three different features revealed by a pattern: *access list*, *access chain*, and *access frequency*.

We define the *access list* as an allowlist containing all the MMIO addresses used to access a device during normal executions. Access to the addresses within the access list is mandatory to operate a peripheral correctly, and each peripheral MCU has its own access list.

To capture the internal connections among different MMIO accesses, we also consider *access chains*. An access chain represents the ordering in which different MMIO accesses happen and is encoded as a directed graph, where each node represents a unique MMIO address and a directed edge between two nodes stands for a possible MMIO access sequence. This graph essentially captures the characteristics of certain protocols communicating with a peripheral MCU. MMIO profiler can automate the detection of this MMIO access sequence and code generation to enforce the access chains.

*Access frequency* records the inter-access time for a particular MMIO address or MMIO region, given a peripheral. Because of the real-time and deterministic nature of UAV

control software, we expect all the features to show stable statistics under normal conditions.

Once learned the access pattern given an MMIO address, we can generate C-based policies or rules automatically using our own Domain Specific Language (DSL) compiler, which enforce the access pattern for this MMIO address. Listing 1 shows the generated code for barometer sensor. These policies use the hooks provided by M2MON and compile together with M2MON during compilation time. More specifically callbacks can be registered against particular MMIO addresses using `register_action` and `register_action_spi` as described in Section 4.2.

**Listing 1:** Generated C code for barometer sensor MMIO model

```
void BARO_filter(unsigned char * send) {
  static unsigned char
        lastTrans = INIT_VALUE;
  switch (lastTrans) {
  case MEASURE_CMD:
    if (send[0] != READ_CMD)
      trigger_failsafe();
    break;
  case  RESET_CMD:
    if (send[0] != READ_CMD)
      trigger_failsafe();
    break;
  case READ_CMD:
    if (send[0] != MEASURE_CMD ||
        send[0] != RESET_CMD)
      trigger_failsafe();
    break;
  case INIT_VALUE:
    break;
  }
  lastTrans = send[0];
}
```

Furthermore, for the online registration of new rules, we implement Command Line Interface (CLI) rules. These are basically callbacks with pre-defined behavior, parameterized with the MMIO address. On receiving command this predefined callback is registered based on the input address using `register_action`. For CLI we assume a secure communication channel. A system owner can issue the following commands:

```
sh> BLOC_register 0xE000E014
sh> FREQ_register 0xE000E014 3000
```

`BLOC_register` restricts access to a specified address (e.g., `0xE000E014`). Once the address is added M2MON denies all access to that address. `FREQ_register` registers the maximum access frequency for an MMIO address. For instance, `0xE000E014` is the address being monitored, and `3000` is the moving average of inter-access frequency.

Figure 9 shows the workflow of M2MON as an access pattern-based firewall. M2MON microkernel monitors the MMIO accesses from peripheral MCUs to detect anomalies based on the previously obtained "fingerprints." As we will see in Section 6, UAV attacks modify the MMIO access pattern in different ways. Our approach aims at finding these anomalies in MMIO access patterns.

## 5.3 Securing Kalman Filter

Kalman filtering [27] is an estimation technique that observes different sensor values over time to estimate some unknown variables. With basic Kalman filtering, we can only model linear systems, however, an extension to Kalman filters known as Extended Kalman Filter (EKF) can estimate non-linear systems as well. EKF is extensively used in control systems for sensor fusion. Sensor fusion is the process of getting values of some physical attribute from different sources. In case one of the values obtained from the sensor is malicious or faulty, the value computed by the EKF could still be correct since it can infer the correct value based on past values and on values acquired by other sensors.

In current UAV implementations, the EKF is implemented as part of the RTOS. However, as we have shown in Section 4.2, the RTOS is typically not secure given the sorry state of affairs in embedded security. For example, using what we explained in Table 2, we can trivially show that we can use the Flash Patch and Breakpoint unit to compromise the RTOS and bypass any check performed by the EKF.

To tackle this problem, we implement the EKF inside M2MON. Therefore, it runs separately from the RTOS and it is affected by its vulnerabilities. At the same time, M2MON guarantees that the EKF implementation can work efficiently (i.e., with low overhead) and safely (due to the usage of SFI).

Figure 9 shows how we implement the EKF module inside M2MON. Using `register_action` and `register_action_spi`, as described in Section 4.2, the user can record previous measurements of some particular sensor. To register a filter, the user can register a synchronous callback using `register_sync_call`, which can be invoked from userspace using `sync_call`. For instance, the hook for EKF check is registered using the following piece of code:

```
register_sync_call(EKF_UPDATE_ID, doEKFUpdate);
```

## 5.4 Post-Detection Response

M2MON-APF and M2MON-EKF can detect malicious activities in the system. However, once we have diagnosed such activities, we need to take defensive action. Essentially, for the continued operation of UAV, we cannot use the peripheral under attack. This situation is similar to a peripheral malfunction. We can use *Fail-Safes* to handle such scenarios. A fail-safe [50, 66] is a design feature of control systems that mitigates the effects of malfunctioning components. A control program can have multiple fail-safes designed around the malfunctioning peripheral. For instance, ArduPilot has multiple fail-safes such as Radio fail-safe, EKF fail-safe, GCS fail-safe, etc. Each fail-safe's behavior is dictated by the

malfunctioning peripheral. For instance, if a radio receiver malfunctions in a UAV, the radio fail-safe response could be to return to the home, as doing so does not require the radio link.

Upon attack detection, we can leverage the relevant fail-safe to continue operation by considering the peripheral under attack as malfunctioning. However, utilizing the fail-safe requires trusting the control program. Unfortunately, the control program is outside the TCB according to our design, since we want a minimal TCB size. In the balance of both security and usability at the same time, we design and implement a two-step *Post-Detection response* in M2MON, where the first step is triggering the typical fail-safe operation provided by the control program, and the second step is to start an *Emergency response*. Emergency response is a platform-specific attack response completely implemented within M2MON and is independent of any component outside M2MON. For UAVs, we choose deploying a parachute as our emergency response.

The design consideration of this 2-step post-detection response is two-folded. As we mentioned earlier, control programs usually implement different fail-safe operations to deal with malfunctioning peripherals. While we still do not trust control programs, triggering them in the first step is beneficial when the attack detected is a false alarm or the control program can execute the fail-safe correctly. Consequently, we only need to implement minimum code within M2MON to reuse the fail-safe operations of control programs instead of implementing all of them inside M2MON. Meanwhile, we continue monitoring the MMIO activities. If we still detect the attack after triggering the fail-safe operation of the control program, we infer that the control program didn't respond. Hence, we need to rely on the emergency response from within M2MON. Thanks to the privilege separation between M2MON and RTOS in our design, we can achieve secure emergency response handling without the need for another MCU by executing it within M2MON.

We modify the control program and move the fail-safe trigger functions inside M2MON. Similarly, users can use `register_sync_call` API to register a fail-safe. Existing userspace code can invoke the fail-safe using `sync_call`. Furthermore, We build our emergency response using the SATS-MINI system. SATS-MINI is an external peripheral used to deploy parachutes for UAVs for a safe landing. The SATS-MINI takes in an input signal of two ms wide pulse as a trigger signal from the main UAV system. This signal causes the SATS-MINI to deploy the parachute. Since we do not want any dependency on components outside M2MON, we write the code to generate the signal inside M2MON. The code consists of the routine to trigger the signal and drivers for relevant peripherals (such as timer and GPIO). Due to logistic constraints, we did not test with an actual parachute, but our implementation adheres to the SATS-MINI specifications. Furthermore, we verified the required signals using a digital oscilloscope. We provide an API `emergency_response` to trigger the emergency response signal.

# 6 Evaluation

We evaluate M2MON using the 3DR IRIS+ UAV platform explained in Section 2 aiming to answer two questions:

- **Effectiveness:** how effectively M2MON can defend against known and new attacks, and reduce the TCB size.
- **Overhead:** how much overhead M2MON introduces with respect to real-time constraints, micro-benchmarks, storage, and SFI instrumentation.

Throughout this section, M2MON refers to the M2MON microkernel. M2MON-APF refers to the access pattern-based firewall using M2MON. M2MON-EKF refers to EKF implementation on top of M2MON.

## 6.1 Security Evaluation

To verify the effectiveness of M2MON, we choose eight attacks out of the nine attacks that we surveyed in Table 1, and we can defend against all the eight attacks using M2MON. We do not include ECU attacks [32] in our evaluation because these attacks do not apply to our UAV platform. As shown in Table 2, we are able to defend these attacks using different detection features provided by the firewall and the EKF. Based on the nature of attacks, we categorize them into two categories: 1) *Signal Spoofing Attacks*, where the attacker attacks the UV by spoofing signals such as GPS, Radio, etc. 2) *Code Compromise Attacks*, where the attack payload includes running code on the flight controller. For each attack, we list the target MMIO activity and the details of the post-detection response. Furthermore, we discuss the possibility of circumvention of M2MON's defenses for each attack as well. In the case of the eighth attack, we cite existing research [3, 4, 39, 47] to show the EKF's efficacy against such physical sensor attacks (such as acoustic attacks [53]). In this section, we briefly explain a few of the case studies.

**Case Study: Timer Attack (case 1).** ARM Cortex-M series have the System Tick Timer (SysTick) which generates interrupt requests periodically to support multi-tasking. RTOS configures this period by writing to the SysTick reload value register `STK_LOAD`. To conduct the timer attack, attackers assign a larger value than the original value for the `STK_LOAD` using existing vulnerabilities found in NuttX [29], degrading the responsiveness of the real-time processes because the scheduler would then work based on the slower clock. As a result, a UAV would demonstrate unstable positions, drop its altitude, and eventually crash [29].

To detect the attack, we can add the address of `STK_LOAD` to the access list (blocklist) of M2MON. Since the RTOS writes the `STK_LOAD` only once during the bootstrapping, if attackers update the value of `STK_LOAD` after initialization,

| Case ID | Attack | Detection Feature | MMIO Register/Address | Attack Type |
|---|---|---|---|---|
| 1 | Timer Attack | Access List | Timer Load Register | Code Execution |
| 2 | IRQ Override | Access List | Vector Table Offset Register | Code Execution |
| 3 | Radio Replay | Access Frequency | GPIO Status Register | Signal Spoofing |
| 4 | Flash Patch Attack | Access List | FPB Control Register | Code Execution |
| 5 | GPS Spoofing | Access Frequency | UART Data Register | Signal Spoofing |
| 6 | Gyroscope Attack | Access List | Device ID 1 Command (SPI) | Code Execution |
| 7 | Barometer Attack | Access Chain | Device ID 3 Command (SPI) | Code Execution |
| 8 | Malicious Sensor values | Kalman Filtering | Data registers related to sensor values | Signal Spoofing |

**Table 2:** Attack cases used to evaluate the effectiveness of M2MON-based firewall and the usage of M2MON-based Kalman filter.

M2MON detects and denies the write operation against the `STK_LOAD`.

*Post-Detection Response:* Since the defense avoids the attack, we don't need to trigger any post-detection response.

*Rule Circumvention:* In this case we block access to I/O address essential to the attack, so even an attacker who is aware of M2MON defense will not be able to conduct this attack.

**Case Study: GPS Spoofing (case 5).** This attack allows an attacker to hijack and control a UAV by sending out spoofing GPS signals. Our GPS spoofing method follows a common setup [58, 70]. We used `GPS-SDR-SIM` [44] with HackRF One [20]. During our attack, we found the GPS module (`u-blox NEO-7N` [59]) cannot detect our GPS spoofing attack [4]. Previous GPS spoofing detection mechanisms [26, 49, 67] have utilized the `Ephemeris` and `Almanac` GPS packets as the criterion. These packets contain the location and orbital information about GPS satellites.

To detect the GPS spoofing attacks, we count the number of `Ephemeris` messages with a window of three minutes. During our five-hour long MMIO profiling on our UAV platform, we found that the GPS module receives a maximum of eight `Ephemeris` messages within three minutes under normal operations. However, under GPS spoofing attacks, we noticed the received number of `Ephemeris` messages is increased by fake GPS signals (minimum 12 and average 14).

To implement such a complex policy within M2MON, we use `register_action` to register a rule against the UART 4 data register, which is used by the GPS module to communicate with the main MCU. Using this we can infer the number of `Ephemeris` messages received and use the platform timer to measure the message frequency.

*Post-Detection Response:* Since the drone cannot reliably continue navigation without a GPS module, on detecting this attack we trigger the emergency response (See Section 5.4) to prevent UAVs from getting hijacked.

*Rule Circumvention:* Attackers can decrease the number of fake GPS satellites to evade such detection mechanism. To verify our detection method, we decreased the number of satellites and noticed that the attackers need to spoof a larger number of fake satellites than the number of benign satellites. For instance, nine fake GPS satellites were required to spoof a location of our UAV platform while it received GPS signals from eight benign satellites. Further, we also counted the `Ephemeris` messages with the nine fake GPS satellites. We noticed that our UAV platform receives a minimum of 10 (and average 13) `Ephemeris` messages. Accordingly, even if the attackers conduct stealthy GPS spoofing attacks by decreasing the number of fake GPS satellites, we could still detect GPS spoofing attacks using the expected maximum frequency of `Ephemeris` messages (e.g., eight) given a period time of operations (e.g., three minutes).

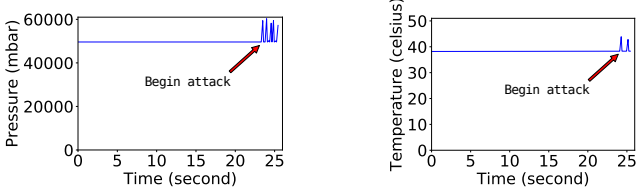
**Case Study: Barometer Attack (case 7).** This attack maliciously modifies the altitude value measured by a drone, thus influencing its altitude. Specifically, after compromising the ESP8266 Wi-Fi module, attackers can use special commands, such as Direct Comms SPI/I2C commands [8], to trigger actions in a barometer sensor. The platform used in our experiments uses the `ms5611` barometer sensor [18]. It measures both temperature values and pressure values to calculate altitude values. According to the datasheet of `ms5611` [18], the sensor requires 10 ms to correctly report the measurement.

However, under the attack, the attackers can trigger `read` command to disrupt the ongoing measurement and destabilize the drone. However, this disturbance is transient, and the drone recovers in the next control loop iteration. To crash the drone, the attacker needs to continually trigger read commands. Figure 10 shows the result of triggering unsolicited operations using Direct Comms commands [8]. To defend against such an attack, we can infer the protocol between ArduPilot and `ms5611`. Figure 11 shows the inferred model. This model can be loaded in M2MON to enforce correct operations.

*Post-Detection Response:* If the flight control software deviates from this behavior M2MON would have to continue operation without the sensor value. This is similar to Ardupilot's EKF fail-safe. In the case of detection, we utilize this fail-safe as described in Section 5.4.

*Rule Circumvention:* To launch this attack, the attacker needs to repeatedly trigger the read command to disrupt the normal operation. However, since the barometer driver exhibits a deterministic pattern, any additional operation will break the pattern. Hence, an M2MON-aware attacker can only take a

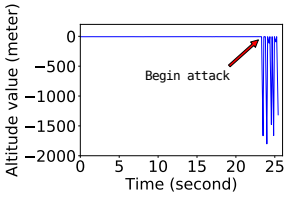
[4] The specification of the GPS module mentions that the spoofing detection cannot detect all types of attacks.



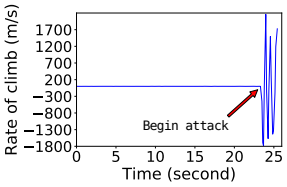

**(a)** Pressure values under the attack.




**(b)** Temperature values under the attack.




**(c)** Altitude values under the attack.




**(d)** Rate of climb values under the attack.

**Figure 10:** The changed sensor values under the barometer sensor attack (*i.e.,* case 7).

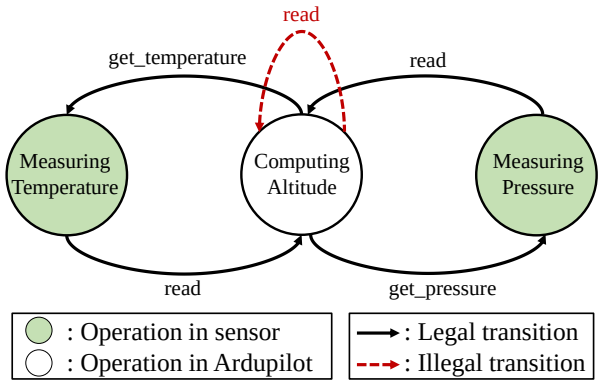



**Figure 11:** State diagram describing how the `ms5611` barometer sensor operates. To obtain the barometer values, it periodically conducts the state transitions. The solid black arrows indicate legitimate state transitions and the red dotted arrow represents an illegal self state transition under the barometer sensor attack.

legal transition to evade detection, thus eliminating the attack.

**Case Study: Flash Patch Attack (case 4).** After compromising the `ESP8266` Wi-Fi module, attackers can execute arbitrary code on a UAV using the Flash Patch and Breakpoint (FPB) [10] unit supported since ARMv7-M. Since the FPB can replace instructions during the CPU execution, it is often used to patch the firmware on the fly. To launch the attack, adversaries use NuttShell (NSH) [42] over the compromised channel to execute memory read and write commands (`mb`, `mh`, and `mw`) remotely.

Attackers can replace a function call with an infinite loop by using the FPB. This can lead to disrupting the operation, even crashing the drone. This attack does not yield any memory access violation because the NSH has permission to access the FPB unit. Accordingly, previous defense methods [16,17,29] cannot detect or defend it. Using M2MON, we can restrict the access to the FPB by blocking the access to registers related to the FPB (*e.g.,* `FP_CTRL` ).

*Post-Detection Response:* Since the defense avoids the attack, we don't need to trigger any post-detection response.

*Rule Circumvention:* In this case, we block access to I/O address essential to the attack, so even an attacker who is aware of M2MON defense will not be able to conduct this attack.

**Case Study: Radio Controller Replay Attack (case 3).** This attack records and replays commands over the radio channel with malicious intent. Our RC replay attack uses `HackRF One` [20] with `GNU radio` to record and replay the control signals against the FrySky receiver. We conducted replay on throttle update, arming, and disarming commands.

The RC receiver and Pixhawk 1 board communicate via a GPIO port. During the attack, we observed that the frequency of I/O accesses to the GPIO port increased, as shown in Figure 1c. For a window of 10 accesses, the maximum moving average of the GPIO access interval was 222.1 ms under normal conditions and 122 ms under the attack. We infer higher I/O activity results because both GCS and the attacker are sending messages at the same time. Furthermore, we note that the RC transmitter continuously sends control signals (such as roll, pitch, throttle) to the UAV regardless of user activity.

Since the activity on the radio channel is independent of the user activity, we use the moving access frequency of the GPIO Output Data register to detect this attack. If the average goes below 200 ms, we conclude an attack is in progress.

*Post-Detection Response:* Once the radio channel is detected to be under attack, we cannot reliably continue its usage. To continue operations, we can return to the initial position without relying on RC commands. This situation is similar to ArduPilot's radio fail-safe. Consequently, we utilize this fail-safe as described in Section 5.4.

*Rule Circumvention:* M2MON-aware attackers can try to conduct this attack while staying over the threshold of 200 ms. We observed that the moving average for a shorter duration of RC replay attacks deviates less from the expected value. More concretely, attacks under two seconds do not exhibit detectable variations. Hence, the attackers might freely change the attitude of the controller during those two seconds. Although we could not 100% defend against the RC replay attack, M2MON severely limits the replay attack time. M2MON was successfully able to detect RC replay attacks longer than three seconds.

## 6.2 Performance Evaluation

**Real-Time Benchmarks** To verify that M2MON is able to satisfy real-time constraints, we use the ArduPilot test suite for UAV flight controllers. This test suite contains a set of tasks, together with their soft deadlines and periods. For instance, the `throttle_loop` task, responsible for controlling

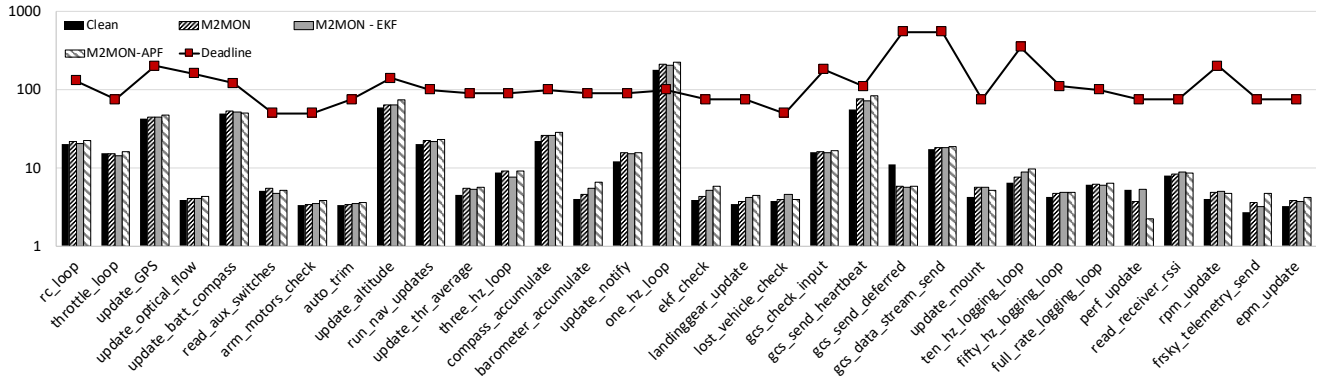



**Figure 12:** Log-based average execution time of soft real-time tasks w/ and w/o M2MON over 100 runs.

the throttle of motors, has a deadline of 8$\mu s$ and a period of 75$\mu s$. Each task runs once after a period and should finish executing within the deadline to satisfy the real-time constraints. The deadlines of these tasks vary from 50$\mu s$ to 550$\mu s$. To get a precise measurement, we disable the preemption of real-time tasks. Figure 12 shows the log-based average runtime of different tasks, in comparison with the soft deadlines, and w/ and w/o M2MON. Compared to the baseline, M2MON introduces 8.85% overhead in average and, it does not violate any soft deadlines except for `one_hz_loop`, which misses its deadline in all cases including the baseline. In other words, we found that this test fails to meet its deadline even in an unmodified system. We investigated this issue and found that this task was sensitive to the UAV's configuration. For our benchmarking, we used the default configuration provided by ArduPilot. Implementing EKF using M2MON incurs an overhead of 10.25%, whereas access pattern-based filter resulted in 16.59% overhead.

Some tasks, such as `gcs_send_deffered` result in a lower runtime with M2MON, because they consume work generated by other tasks which are slowed down because of M2MON. We suspect that this slow down results in fewer messages generated for such consumer tasks, which in turn complete their job faster.

**Micro-Benchmarks** The core of M2MON runtime is I/O handling. As mentioned in Section 4.2, MMIO accesses and external bus accesses are handled differently inside M2MON. A single MMIO transaction incurs a latency of 28$\mu s$, out of which the hashing incurs a latency of 24$\mu s$. Similarly, one transaction over an external bus incurs an overhead of 13$\mu s$. External bus access is faster due to the fact that there are fewer distinct device IDs on buses, unlike distinct MMIO addresses. Thus instead of hashing, we used jump tables to dispatch any rules on external buses based on the device ID. Even though user-supplied rules are not part of M2MON, based on our security experiments, the barometer access chain checking incurred an overhead of 6$\mu s$ and the access list checking incurs an overhead of 4$\mu s$.

**Storage Overhead** All of M2MON data is kept inside a custom ELF section, named *.mon*. M2MON incurs a storage overhead of 2,560 bytes. M2MON-APF incurs an overhead of 18,976 bytes. M2MON-EKF incurs an overhead of around 3,584 bytes. M2MON-APF incurs the highest overhead, which was mainly due to M2MON-APF's hash map. For this reason, we suggest keeping the hash map as small as possible without collisions. For our evaluation, we used a 4,096 element hash map, where each element requires 4 bytes. In all configurations, the shadow stack is also stored in the *.mon* section.

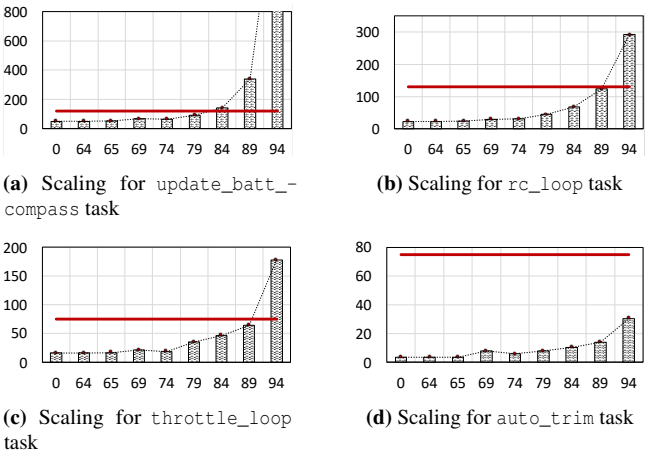
| TCB | LoC |
|---|---|
| NuttX RTOS | 3,114,206 |
| M2MON | 3,422 |
| M2MON-APF | 3,775 (M2MON + 353) |
| M2MON-EKF | 4,027 (M2MON + 605) |
| M2MON-PDR | 4,069 (M2MON + 647) |

**Table 3:** LoC Comparison between the NuttX RTOS, M2MON microkernel, M2MON-APF, M2MON-EKF and M2MON-PDR (Post-Detection Response).

**Instrumentation Overhead** M2MON instrumentation incurs an overall overhead of 0.04% increase in code size. This is because interrupts handlers are a very small part of the RTOS. Considering only the interrupt handlers, we increase their size of 30.14%. These numbers show that running SFI on interrupt handlers yields in trivial overhead.

**TCB Reduction** Due to the simple nature of the M2MON microkernel, we were able to drastically reduce the TCB running in our test device. Specifically, the 3DR IRIS+ UAV used in our experiments originally run the entire NuttX RTOS (3,114,206 LoC) in the privileged mode. On the contrary, using our approach it only needs to run the M2MON microkernel in the privileged mode, which is composed of 3,775 LoC, as shown in Table 3. The M2MON-based firewall and Extended Kalman filter implementations add an extra 353 and 605 LoC, respectively. Implementing post-detection response requires additional 647 LoC. We do not include the board support package in our line count.

**Scalability Test** M2MON targets low-end embedded sys-




**(a)** Scaling for `update_batt_compass` task

**(b)** Scaling for `rc_loop` task

**(c)** Scaling for `throttle_loop` task

**(d)** Scaling for `auto_trim` task

**Figure 13:** Runtime for various tasks under stress testing. We observe different scalability for different tasks, based on the usage of I/O and slack time available. The x-axis is number of rules, while y-axis is time in milliseconds.

tems. Low processing power limits the functionality of M2MON. We use M2MON-APF to demonstrate the scalability of M2MON, with access control rules as our target rules. Theoretically speaking, we can incur as much overhead as the slack time available in the system. Slack time is the difference between the deadline and worst-case execution time (WCET) of a task. Since ArduPilot is a soft real-time system, we use the average execution time in place of WCET.

Figure 13 shows the execution times of different tasks against a different number of rules. Due to hashing, we see a constant overhead for up to 64 rules. Afterward, in general, we see an exponential increase in overhead. The variability in the execution times is due to asynchronous interrupt handlers. Each task misses its deadline based on two factors 1) the number of I/O accesses and 2) slack time available. For example, `update_batt_compass` start missing its deadline near 80 rules, whereas `auto_trim` task does not miss its deadline even after more than 90 rules.

In conclusion, M2MON can process a reasonable amount of work within the system constraints. For our evaluation, we had fewer than 20 rules for all of the case studies, covering all known attacks. We might also need more rules to defend against other attacks and even future ones. However, M2MON cannot execute overly complicated rules, such as statistical machine learning methods, without violating the deadlines. Multiple solutions can be adapted to tackle this limitation. For example, users can add a co-processor on board to process these rules. Another option could be processing rules asynchronously, such that we can process the data in later epochs. Lastly, we can log this data to non-volatile storage for post-mortem analysis.

# 7 Discussion

**Rule Circumvention:** Attackers aware of the enforced policies might try to mimic the regular pattern to evade detection. These attacks are also known as mimicry attacks. For a mimicry attacker, we have two types of enforced policies: 1) deterministic (case 1, 2, 4, 6, and 7 in Table 2), e.g., access control and access patterns, and 2) statistical (case 3, 5, and 8 in Table 2), e.g., access frequency and Kalman filtering. For deterministic policies, attacks have to perform forbidden MMIO patterns. Hence, they cannot evade detection. For statistical policies, like any statistical method, we cannot guarantee that attackers cannot circumvent the enforced policies. However, M2MON severely limits the freedom of the attacker (RC attack, GPS Spoofing attack) or even eliminating the attack (FPB attack, IRQ Override attack), as shown in Section 6.1.

**Normal Operations vs. Exceptions** Currently, we profile I/O accesses on normal/benign runs of the UV program for the firewall. This comes with the inherent limitations of dynamic analysis, *i.e.* it is hard to guarantee full coverage of source code. However, due to the deterministic behavior of control programs, we are able to retrieve information about the normal operations of UV which is sufficient for most functionality of UV. For instance, we triggered all flight modes and fail-safe modes[5] in ArduPilot. As a result, except for the MMIO associated with the RC channel (*i.e.,* GPIO), the MMIO access patterns were the same as the patterns under the normal operation. This is because the fail-safe modes just change the flight mode. In addition, the flight modes do not change any operation of peripheral MCUs. However, when the drone is disarmed (not flying), it temporarily masks GPIO interrupts. In terms of exceptional behavior, the firewall would regard it as malicious behavior, since it has not seen it in the benign run. Currently, the solution to this problem is to use the return-to-home feature in IRIS 3DR to cater for exceptions. We can complement our dynamic analysis to facilitate programmable exception handling.

**M2MON Adaptability:** M2MON's design is generic and runs on any system providing privilege separation and MPU. Both of these features are readily available on most MCUs. However, adapting M2MON to a new platform requires some engineering effort. This effort depends on 1) the RTOS and 2) the Flight Control Software.

For 1), we have to run the RTOS in userspace. Most RTOS can run in both single privilege execution and privilege separation execution, based on configuration. Users can leverage such support to achieve this task. Furthermore, with wider adoption, this effort would diminish. For instance, even for M2MON, we were able to use the existing implementation of userspace NuttX from Minion [29]. Secondly, to run code from outside the TCB in privilege mode, we provide a custom compiler attribute. Annotating functions with this attribute

[5] ArduCopter 3.3 supports 15 flight modes and 4 fail-safe modes [7].

instruments the code with required checks.

For 2), we have to get the I/O patterns from the software. To this end, we provide scripts to extract the access patterns from MMIO access logs. Furthermore, we provide high-level APIs to bind rules to a specific MMIO address. This helps bridge the semantic gap and ease the adaptability for M2MON.

**Allowlist vs. Blocklist.** We use a blocklist-based approach in the firewall to list the MMIO addresses that attacks often exploit and trigger certain access control, defending all known attacks. We could also implement an allowlist for MMIO addresses e.g., only allowing access to the addresses within the list and rejecting others by default. We could even leverage the MMIO profiles to extract the access pattern for a given address and register a rule/policy for this address to enforce access control. While the blocklist-based approach saves us the policy storage overhead provided that only a few MMIO addresses are the attacking target (and it is usually true), the allowlist method could defend against unanticipated attacks in the price of both runtime and storage overhead. This trade-off depends on the threat model and the specific UV environment.

**Why not TrustZone?** ARM TrustZone is used to partition a system into secure and non-secure worlds. The extension provides support for programming the bus dynamically using TrustZone controllers [11], which helps configure system memory space into secure and non-secure memory at runtime. In our survey, however, we found that most vendors do not support extensions for TrustZone at the bus level [43, 45, 46]. Using such design would break the **G1** *Complete Mediation* of M2MON. Even if the support does exist at the bus-fabric level, they might not use the appropriate controllers to enable dynamic programming of the system memory map. Therefore, users are stuck with the vendor-supplied configuration, which often consists of only a couple of secure devices and leaves most resources for the non-secure world. Furthermore, TrustZone for ARM microcontroller profiles is relatively new and not pervasive.

**Static Analysis Limitations.** M2MON uses static analysis for software fault isolation. However current implementation is unable to find MMIO accesses not local to the translation unit. This is because our pass runs on a single translation unit, and symbols not internal to the file cannot be determined at compile-time, hence a user can escape the sandbox using MMIO pointers with external linkage. However, this can be solved by using a Link Time Optimization Pass (LTO) [19]. During our implementation, we found that such a pattern is not used to define MMIO pointers in ArduPilot.

## 8 Related work

**Malicious Peripherals & Defenses in Traditional Domains:** Different peripheral attacks have been demonstrated in past. Google Project Zero showed how full mac Wi-Fi chips can be compromised, which led to eventually resulting in compromising Android and iOS [55]. Existing work has shown how USB devices can be used to attack file systems [64], exploit direct memory access (DMA) [51], eavesdrop [40] and masquerading [22]. BleedingBit [24] enables unauthenticated devices to attack host CPUs by exploiting vulnerabilities in Bluetooth chip-sets. To cater to this problem several solutions have been proposed. LBM [56] provides provisions for eBPF filter firewall in different device stacks to monitor/filter data transmitted on those devices. USBFilter [57] and USBFirewall [25] are solutions available for Linux and FreeBSD to defend against malicious USB devices. SeCloak [34] and Ditio [38] leverage ARM TrustZone to provide peripheral control securely. Unfortunately, none of these defenses could be applied to UV directly due to its unique settings and challenges.

**Attacks & Defenses in Embedded Systems.** Besides all the attacks impacting different peripheral MCUs mentioned in Table 1, attackers can also trick sensors to provide malicious values to the control loop. Targeted Electro-Magnetic Interference (EMI) has been shown to confuse sensors to provide wrong values [48]. Sound waves can disrupt gyroscope sensors [53]. To defend against these attacks, both compiler-based and system-based solutions have been proposed. Epoxy [17] automatically identifies all sensitive instruction and increases software privilege level to enforce hardware security mechanisms. However, this scheme is not feasible for real-time systems. MINION [29] has tried to partition the memory into as many clusters as available MPU regions, but it suffers from the limited number of regions available on MCUs. ACES [16] creates sandbox based on static and dynamic behavior to restrict memory access and code flow according to the least privilege policy. But the moderate overhead imposes hurdles for real-time constraints. Compared to these defenses, M2MON achieves balances between both security and performance.

**I/O Kernels:** Different microkernels for the sake of I/O have been created in past. Wimpy kernels [71] utilized virtualization extensions to move device drivers outside the operating system. Unlike driver domains [68], bus-related code is kept in the OS, while any services required by the bus driver are probed and verified by the wimpy kernel. VIPER [35] provides a firmware authentication protocol to mitigate proxy attacks during firmware attestation. NoHype [28] removes the virtualization layer by statically assigning devices to virtual machines, removing the monitor from the virtualization stack. Even though there is existing work on I/O Kernels, nearly all such kernels target server systems resulting in high overheads not suitable for UVs.

## 9 Conclusion

In this paper, we survey different UV attacks and observe their unique I/O activities at the MMIO level. We design and implement a security reference monitor namely M2MON, a microkernel with less than 4K LoC mediating every MMIO

access within the UV. We further implement an MMIO access pattern-based firewall and Kalman filter using M2MON, and demonstrate its effectiveness against a number of UV attacks while introducing minimal overhead. M2MON is the first step towards building a trusted and practical security reference monitor for UV.

## Acknowledgments

We thank the anonymous reviewers for their valuable comments. This work was supported in part by ONR under Grants N00014-20-1-2128 and 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] 3DR. 3dr iris+. http://3dr.com/support/articles/207358106/iris.

[2] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Transactions on Information and System Security (TISSEC)*, 13(1):1–40, 2009.

[3] Chuadhry Mujeeb Ahmed, Martin Ochoa, Jianying Zhou, Aditya P Mathur, Rizwan Qadeer, Carlos Murguia, and Justin Ruths. Noiseprint: Attack detection using sensor and process noise fingerprint in cyber physical systems. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 483–497, 2018.

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

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. Compilers, principles, techniques. *Addison wesley*, 7(8):9, 1986.

[6] James P Anderson. Computer security technology planning study. Technical report, Anderson (James P) and Co Fort Washington PA, 1972.

[7] ArduPilot. Ardupilot. http://ardupilot.org/.

[8] ArduPilot. Direct comms module. https://tinyurl.com/735giodv.

[9] ARM. Arm trusted firmware. https://github.com/ARM-software/arm-trusted-firmware.

[10] ARM. Cortex-m3 revision r2p0. https://tinyurl.com/1hrh4s6t.

[11] ARM. Trustzone® address space controller (tzc-380) revision: r0p0. https://tinyurl.com/2abykb3y.

[12] Brian C Barker, John W Betz, John E Clark, Jeffrey T Correia, James T Gillis, Steven Lazar, Kaysi A Rehborn, and John R Straton III. Overview of the gps m code signal. Technical report, MITRE CORP BEDFORD MA, 2006.

[13] BBC. Amazon prime air. https://tinyurl.com/2x933hgx.

[14] Kyong-Tak Cho and Kang G. Shin. Fingerprinting electronic control units for vehicle intrusion detection. In *25th USENIX Security Symposium*, pages 911–927, 2016.

[15] Kyong-Tak Cho and Kang G Shin. Viden: Attacker identification on in-vehicle networks. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1109–1123. ACM, 2017.

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

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

[18] TE Connectivity. Ms5611-01ba03 datasheet. https://tinyurl.com/1ohhie02.

[19] GNU. Linker plugins. https://tinyurl.com/yaf0yx0k.

[20] greatscottgadgets. Hackrf one. https://greatscottgadgets.com/hackrf.

[21] Dina Hadžiosmanović, Robin Sommer, Emmanuele Zambon, and Pieter H Hartel. Through the eye of the plc: semantic security monitoring for industrial processes. In *Proceedings of the 30th Annual Computer Security Applications Conference*, pages 126–135. ACM, 2014.

[22] Hak5. Usb rubber ducky. https://shop.hak5.org/products/usb-rubber-ducky-deluxe.

[23] Vinay M Igure, Sean A Laughter, and Ronald D Williams. Security issues in scada networks. *computers & security*, 25(7):498–506, 2006.

[24] Armis Inc. Bleeding bit. https://armis.com/bleedingbit/.

[25] Peter C Johnson, Sergey Bratus, and Sean W Smith. Protecting against malicious bits on the wire: automatically generating a usb protocol parser for a production kernel. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, pages 528–541. ACM, 2017.

[26] Aleksandar Jovanovic, Cyril Botteron, and Pierre-Andre Fariné. Multi-test detection and protection algorithm against spoofing attacks on gnss receivers. In *2014 IEEE/ION Position, Location and Navigation Symposium-PLANS 2014*, pages 1258–1271. IEEE, 2014.

[27] Simon J Julier and Jeffrey K Uhlmann. Unscented filtering and nonlinear estimation. *Proceedings of the IEEE*, 92(3):401–422, 2004.

[28] Eric Keller, Jakub Szefer, Jennifer Rexford, and Ruby B Lee. Nohype: virtualized cloud infrastructure without the virtualization. In *Proceedings of the 37th annual international symposium on Computer architecture*, pages 350–361, 2010.

[29] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu. Securing real-time microcontroller systems through customized memory view switching. In *Network and Distributed Systems Security Symp.(NDSS)*, 2018.

[30] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. Rvfuzzer: finding input validation bugs in robotic vehicles through control-guided testing. In *28th USENIX Security Symposium*, pages 425–442, 2019.

[31] Amit Klein, Haya Shulman, and Michael Waidner. Internet-wide study of dns cache injections. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[32] Karl Koscher, Alexei Czeskis, Franziska Roesner, Shwetak Patel, Tadayoshi Kohno, Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, et al. Experimental security analysis of a modern automobile. In *2010 IEEE Symposium on Security and Privacy*, pages 447–462. IEEE, 2010.

[33] Ralph Langner. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy*, 9(3):49–51, 2011.

[34] Matthew Lentz, Rijurekha Sen, Peter Druschel, and Bobby Bhattacharjee. Secloak: Arm trustzone-based mobile peripheral control. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 1–13. ACM, 2018.

[35] Yanlin Li, Jonathan M McCune, and Adrian Perrig. Viper: verifying the integrity of peripherals' firmware. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 3–16, 2011.

[36] Mavlink. Mavlink version. https://mavlink.io/en/guide/mavlink_version.html.

[37] Mavlink. Message signing over rc. https://mavlink.io/en/guide/message_signing.html.

[38] Saeed Mirzamohammadi, Justin A Chen, Ardalan Amiri Sani, Sharad Mehrotra, and Gene Tsudik. Ditio: Trustworthy auditing of sensor activities in mobile & iot devices. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 28. ACM, 2017.

[39] Shoei Nashimoto, Daisuke Suzuki, Takeshi Sugawara, and Kazuo Sakiyama. Sensor con-fusion: Defeating kalman filter in signal injection attack. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 511–524, 2018.

[40] Matthias Neugschwandtner, Anton Beitler, and Anil Kurmus. A transparent defense against usb eavesdropping attacks. In *Proceedings of the 9th European Workshop on System Security*, page 6. ACM, 2016.

[41] Dennis K Nilsson and Ulf E Larson. Secure firmware updates over the air in intelligent vehicles. In *ICC Workshops-2008 IEEE International Conference on Communications Workshops*, pages 380–384. IEEE, 2008.

[42] NuttX. Nuutx kernel. https://nuttx.apache.org/.

[43] OP-TEE. Raspberry pi trustzone implementation. https://tinyurl.com/1bnf0vic.

[44] osqzss. Gps-sdr-sim project. https://github.com/osqzss/gps-sdr-sim.

[45] Paparazziuav. Trustzone implmentation in parrot bebop drone. https://wiki.paparazziuav.org/wiki/Bebop.

[46] Parrot. Parrot ar drone trustzone implmentation. https://tinyurl.com/1rw2f3dz.

[47] Raul Quinonez, Jairo Giraldo, Luis Salazar, Erick Bauman, Alvaro Cardenas, and Zhiqiang Lin. SAVIOR: Securing autonomous vehicles with robust physical invariants. In *29th USENIX Security Symposium*, 2020.

[48] William A Radasky, Carl E Baum, and Manuem W Wik. Introduction to the special issue on high-power electromagnetics (hpem) and intentional electromagnetic interference (iemi). *IEEE Transactions on Electromagnetic Compatibility*, 46(3):314–321, 2004.

[49] Aanjhan Ranganathan, Hildur Ólafsdóttir, and Srdjan Capkun. Spree: a spoofing resistant gps receiver. In *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking*, pages 348–360. ACM, 2016.

[50] Paul H Riley. Failsafe electronic control systems, January 12 1988. US Patent 4,718,229.

[51] Russ Sevinsky. Funderbolt: Adventures in thunderbolt dma attacks. *Black Hat USA*, 2013.

[52] Yasser Shoukry, Paul Martin, Yair Yona, Suhas Diggavi, and Mani Srivastava. Pycra: Physical challenge-response authentication for active sensors under spoofing attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 1004–1015. ACM, 2015.

[53] Yunmok Son, Hocheol Shin, Dongkwan Kim, Youngseok Park, Juhwan Noh, Kibum Choi, Jungwoo Choi, and Yongdae Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *24th USENIX Security Symposium*, pages 881–896, 2015.

[54] Paul Syverson. A taxonomy of replay attacks. Technical report, NAVAL RESEARCH LAB WASHINGTON DC, 1994.

[55] Google Project Zero Team. Over the air: Exploiting broadcom's wi-fi stack. https://tinyurl.com/1bvwtgyv.

[56] Dave Jing Tian, Grant Hernandez, Joseph I Choi, Vanessa Frost, Peter C Johnson, and Kevin RB Butler. Lbm: A security framework for peripherals within the linux kernel. In *LBM: A Security Framework for Peripherals within the Linux Kernel*. IEEE, 2019.

[57] Dave (Jing) Tian, Nolen Scaife, Adam Bates, Kevin Butler, and Patrick Traynor. Making USB great again with USBFILTER. In *25th USENIX Security Symposium*, pages 415–430, 2016.

[58] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 75–86, 2011.

[59] ublox. Neo-7 u-blox 7 gnss modules ubx-13003830 r07. https://tinyurl.com/oub09a48.

[60] Anthony Van Herrewege, Dave Singelee, and Ingrid Verbauwhede. Canauth-a simple, backward compatible broadcast authentication protocol for can bus. In *ECRYPT Workshop on Lightweight Cryptography*, volume 2011, 2011.

[61] Mathy Vanhoef and Frank Piessens. Key reinstallation attacks: Forcing nonce reuse in wpa2. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, pages 1313–1328. ACM, 2017.

[62] Mathy Vanhoef and Frank Piessens. Release the kraken: New kracks in the 802.11 standard. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 299–314. ACM, 2018.

[63] Stack Shield Vendicator. A stack smashing technique protection tool for linux. *World Wide Web, http://www.angelfire. com/sk/stackshield/info. html*, 2000.

[64] Common Vulnerabilities and Exposures. Cve-2015-0096. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2015-0096.

[65] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM symposium on Operating systems principles*, pages 203–216, 1993.

[66] Kevin Warwick and Ming T Tham. *Failsafe Control Systems: Applications and Emergency Management*. Springer Science & Business Media, 2012.

[67] Hengqing Wen, Peter Yih-Ru Huang, John Dyer, Andy Archinal, and John Fagan. Countermeasures for gps signal spoofing. In *ION GNSS*, volume 5, pages 13–16, 2005.

[68] XEN. Driver domains in xen. https://wiki.xenproject.org/wiki/Driver_Domain.

[69] Yi Yang, K McLaughlin, T Littler, S Sezer, Eul Gyu Im, ZQ Yao, B Pranggono, and HF Wang. Man-in-the-middle attack test-bed investigating cyber-security vulnerabilities in smart grid scada systems. ., 2012.

[70] Kexiong Curtis Zeng, Shinan Liu, Yuanchao Shu, Dong Wang, Haoyu Li, Yanzhi Dou, Gang Wang, and Yaling Yang. All your gps are belong to us: Towards stealthy manipulation of road navigation systems. In *27th USENIX Security Symposium*, pages 1527–1544, 2018.

[71] Zongwei Zhou, Miao Yu, and Virgil D Gligor. Dancing with giants: Wimpy kernels for on-demand isolated i/o. In *2014 IEEE Symposium on Security and Privacy*, pages 308–323. IEEE, 2014.