VirtualDrone: Virtual Sensing, Actuation, and Communication for Attack-Resilient Unmanned Aerial Systems

Man-Ki Yoon
University of Illinois at Urbana-Champaign
mkyoon@illinois.edu

Bo Liu
University of Illinois at Urbana-Champaign
boliu1@illinois.edu

Naira Hovakimyan
University of Illinois at Urbana-Champaign
nhovakim@illinois.edu

Lui Sha
University of Illinois at Urbana-Champaign
lrs@illinois.edu

ABSTRACT

As modern unmanned aerial systems (UAS) continue to expand the frontiers of automation, new challenges to security and thus its safety are emerging. It is now difficult to completely secure modern UAS platforms due to their openness and increasing complexity. We present the *VirtualDrone Framework*, a software architecture that enables an attack-resilient control of modern UAS. It allows the system to operate with potentially untrustworthy software environment by virtualizing the sensors, actuators, and communication channels. The framework provides mechanisms to monitor physical and logical system behaviors and to detect security and safety violations. Upon detection of such an event, the framework switches to a trusted control mode in order to override malicious system state and to prevent potential safety violations. We built a prototype quadcoper running an embedded multicore processor that features a hardware-assisted virtualization technology. We present extensive experimental study and implementation details, and demonstrate how the framework can ensure the robustness of the UAS in the presence of security breaches.

CCS CONCEPTS

•Security and privacy →Systems security; •Computer systems organization →Embedded and cyber-physical systems; *Robotics;*

KEYWORDS

Unmanned Aerial Systems, Security, Virtualization

ACM Reference format:
Man-Ki Yoon, Bo Liu, Naira Hovakimyan, and Lui Sha. 2017. VirtualDrone: Virtual Sensing, Actuation, and Communication for Attack-Resilient Unmanned Aerial Systems. In *Proceedings of the 8th ACM/IEEE International Conference on Cyber-Physical Systems, Pittsburgh, PA USA, April 2017 (ICCPS),* 12 pages.
DOI: http://dx.doi.org/10.1145/3055004.3055010

This work was carried out in part in the Intelligent Robotics Laboratory, University of Illinois. This work is supported in part by grants from NSF CNS 13-02563 and Navy N00014-14-1-0717. Any opinions, findings, and conclusions or recommendations expressed here are those of the authors and do not necessarily reflect the views of sponsors.

ICCPS, Pittsburgh, PA USA

DOI: http://dx.doi.org/10.1145/3055004.3055010

1 INTRODUCTION

The booming *Unmanned Aerial Systems* (UAS) industry holds tremendous potential to boost productivity and the economy. In addition to military use cases, UAS platforms have already been experimented in many civil uses, such as delivery, surveillance, transportation, and journalism, to name a few. With flight intelligence and autonomy enabled by modern computing and communication technologies, UAS are ubiquitously networked as an important component for Internet of Things.

The concern about UAS security is growing with the increasing demand on advanced functionalities and thus their increasing capabilities. The increased computational power and connectivity in modern UAS, and open-software environment (both operating system and applications) expose hitherto unknown security vulnerabilities. Threats to such systems are growing both in number as well as sophistication, as demonstrated by recent attacks [8, 9, 16]. A successful attack on safety-critical systems like UAS can result in the safety of such systems being compromised, leading to disastrous effects, from loss of human life to damages to the environment.

Hence, in order to fully integrate UAS into the current airspace, we need an *attack-resilient* UAS platform to assure the safety of modern UAS and the environment. In this paper, we propose VirtualDrone, a software framework to tackle security challenges and achieve assured autonomy in modern UAS platforms. The framework aims to achieve cyber attack-resilient control of UAS even in the event of a security violation. For this, it provides two separate control environments – the *normal control environment* that allows the user to fully control the UAS with advanced functionalities, and the *secure control environment* that provides only a minimal set of capabilities for a safe control in order to minimize the attack surface. In normal circumstances, a UAS operates in the normal control environment, utilizing advanced but potentially untrusted applications. A security and safety monitoring module, which runs in the secure environment, continuously monitors the physical and logical states of the UAS in order to detect safety and security violations. Upon detection of such an event, the secure control environment takes the control of the UAS, limiting unreliable, untrustworthy functionalities. Then, the trusted controller drives the control of the UAS while a corrective action takes place.

For a clean separation between the two control environments, we take advantage of modern embedded processor that features hardware-assisted *virtualization* technology. We sandbox the normal control environment in a virtual machine to isolate potential security breaches. The secure control environment runs directly on the host machine, acting as if it is a hardware security module

but with a higher flexibility due to software control. The virtualization of sensor, actuator, and communication is the key element of the VirtualDrone framework. We virtualize these devices to protect them from potential threats on the integrity and availability by abstracting away low-level details and by controlling accesses. The virtual communication also enables an authorized operator to override suspicious behaviors of the normal control environment, by providing a hidden communication channel. The virtualization also allows for the use of a rich set of existing security techniques such as virtual machine introspection [14]. Furthermore, multicore processors, which become more prevalent in modern embedded computing systems, enable concurrent execution of the normal and secure control environments and also run-time safety and security monitoring efficiently on the same chip.

Our paper makes the following contributions:

- We introduce a novel framework, VirtualDrone, a software architecture that enables a cyber attack-resilient UAS platform by safeguarding critical system resources using a virtualization technique on a multicore processor.
- We implemented the framework on a prototype quadcopter using an off-the-shelf embedded computing board that runs a quad-core processor with hardware-assisted virtualization. Our implementation aims to use existing open-source software stacks without any modifications to the host and guest operating systems as well as the virtual machine monitor.
- We present case studies to illustrate the effectiveness and versatility of the framework. Through experimental validations we demonstrate how the framework provides an integrated platform to defend against various types of attacks.

2 BACKGROUND

Modern UAS Computing Platform: General-purpose operating systems, especially Linux-based OS variants, are becoming the leading OS for intelligent vehicles [7, 10], enabling the industry to promote more intelligent applications. Thanks to manufacturing advancement, embedded systems can run on light-weight computing platforms, such as Raspberry Pi [5], Qualcomm Snapdragon flight control board [10], and Intel's Aero board [12]. Many computation-intensive applications such as computer vision and complex navigation programs can now run onboard to unleash advanced capabilities of modern UAS computing platforms. Applications developed for general-purpose systems can also be ported to these platforms with minimal migration efforts. Moreover, these systems provide convenient networking interface such as WiFi and cellular network which increases connectivity and usability of the platforms. Meanwhile, many UAS applications are community-supported open-source applications [1]. They allow for adding new features, tuning the performance, etc. However, the open nature of the software environment, in conjunction with the increased capabilities and complexities of the modern UAS platforms, inevitably introduce more security vulnerabilities to UAS.

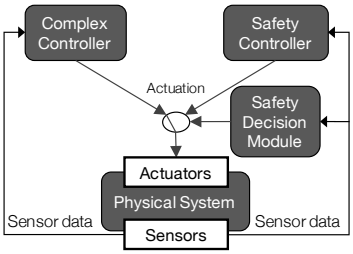

Figure 1: Simplex architecture.

Simplex Architecture: The Simplex architecture [24, 27], shown in Figure 1, is a software solution that enables the use of a high-performance (with the purpose of system performance optimization), potentially unverifiable controller (due to complex software structure) in a safe manner; a high-assurance control is guaranteed even when the *complex controller* fails due to, for example, software bugs or unreliable logic. This is achieved by running a *safety controller*, which has a limited level of performance but is robust, in parallel. Sensor data from the physical system is fed to both controllers, each of which individually computes actuation commands using their own control logic. Under normal circumstances, the physical plant is driven by the complex controller. The *safety decision module* plays a critical role in assuring the safety of the system; it continuously monitors physical states of the plant and checks safety violations, determined by a *safety envelope*. If such a violation is detected, the control is transferred to the safety controller to guarantee a continuous and robust control of the system.

3 VIRTUALDRONE FRAMEWORK

The increasing security challenges posed on the modern UAS platforms make it infeasible to completely secure them because there exist many entry points that are vulnerable to potential security attacks. The main idea that we propose in this paper is that of a software framework to achieve an *attack-resiliency*. The framework isolates a normal but untrustworthy execution environment from a trusted one that (i) manages real I/O operations, (ii) continuously monitors for detection of security and safety violations by the former and (iii) takes back the control of the physical system in such an event. We achieve this by taking advantage of modern embedded processor that features virtualization technology and increased computing power due to multiple cores.

3.1 High-level Framework

The VirtualDrone framework runs two separate control environments that provide different levels of functionalities and capabilities and thus require different degrees of protection.

Normal Control Environment (NCE): It corresponds to the complex controller in the Simplex architecture (see Figure 1) that runs software components for any normal function of UAS. This includes advanced flight and mission controls, and supplementary software such as image processing and networking applications. These typically require external networking (which could be insecure) for status reporting, data transfer, administration, etc. Also, often they are complex, third party-developed, and/or subject to frequent updates/upgrades, which hinders pre-verification or certification on them. Hence, these types of software components running in the NCE are more susceptible to security threats.

Secure Control Environment (SCE): It runs a minimal set of software components that are critically required to control the UAS even when the normal environment is completely taken over by an adversary and does not function. A *security and safety monitoring module* in the SCE thus not only monitors the physical state of the system but also implements a set of security monitors to detect potential security violations. The software components running in the SCE are static since they are designed for safety purpose

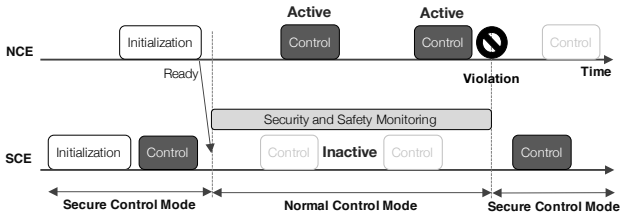

Figure 2: Switching between the SCE and the NCE.

and thus require simple software structure and that a significant amount of analysis is carried out post-design/implementation. Also, they require no or infrequent updates once deployed.

We call the system is in *virtual control mode* if the system is driven by the controller running in the NCE (i.e., virtual machine). By contrast, the system is in *host control mode* if the system is driven by the trusted controller in the SCE (i.e., on the host). In normal circumstances, the system is in the virtual control mode (i.e., NCE), as illustrated in Figure 2, providing the user with the full functionality. Upon detection of the event of a security or safety violation, the SCE takes back the control of the system in order to override the malicious behavior and to maintain the system in a controllable state.

Figure 3 presents the high-level overview of the VirtualDrone framework. The key components in the framework are (i) the virtual machine that sandboxes the full-featured but untrusted operations including the control and other applications (e.g., mission, imaging, networking), (ii) the virtualized sensors (e.g., inertial-measurement unit), actuators (e.g., motors), communication channel (e.g., telemetry with the ground station) for use by the normal environment, (iii) the interface between the real I/O devices and the virtualized ones, (iv) the security and safety monitor that continuously monitors on the logical and physical behaviors of the system driven by the normal environment, and lastly (v) the trusted controller for a robust backup and recovery. It should be noted that the controller running in the NCE could have been fully verified. However, it is deemed untrustworthy because of potential direct and indirect threats by other software components residing together.

The VirtualDrone framework benefits from a multicore processor by being able to run the normal and secure environments in parallel. The latter can continuously perform safety and security checks, and real I/O operations while the former is carrying out its normal operations, which are not possible on a single core processor.

3.2 Assumptions and Adversary Model

- Since we utilize a virtualization technique for the isolation of potentially untrustworthy software components, the virtual machine monitor is our trusted computing base.
- The SCE does not use the external network (i.e., Internet) because, as explained above, the software components in the SCE are static. If needed, any updates to the SCE require physical accesses to the hardware; over-the-air management is not allowed.
- Inside the virtual machine, an adversary may breach any part of the software stack (the OS kernel, run-time libraries, file system, user applications) and can even have root-level access (in the VM) and thus have full control of any software running in the VM. This would enable the attacker to create a backdoor and even replace applications with maliciously modified ones by exploiting vulnerabilities or social-engineering techniques.

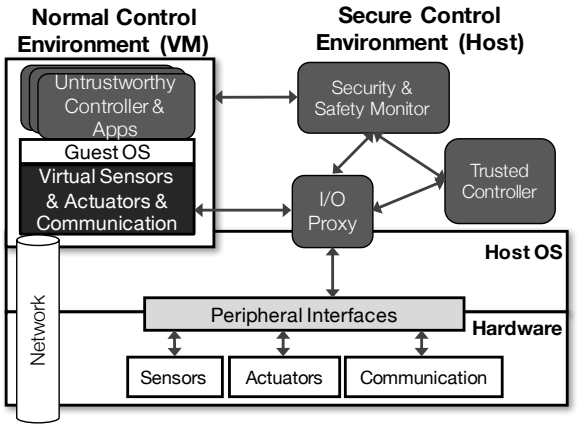

Figure 3: Overview of the VirtualDrone Framework.

- We do not consider physical attacks [16, 26], such as GPS spoofing, against the hardware. Such attacks can be handled by control-theoretic approaches such as [18, 19].

3.3 Virtual Sensing, Actuation, and Communication

The key requirement for sound functioning of the VirtualDrone framework is a clear separation between the normal and secure control environments. Otherwise, an attacker may take over the trusted controller or the monitoring module so that they could not function properly when needed. It is also crucial to protect the sensors and actuators from the untrustworthy components in the NCE as the attacker may degrade their availability or even corrupt them so that the entire system operates with incorrect information on the physical state. Hence, we use a virtualization technique to isolate the NCE from what we need to protect, i.e., the SCE. The NCE, which runs potentially untrustworthy components, is *sandboxed* in a virtual machine. They see a controlled environment configured by the secure side that has a direct control on the system resources.

Virtual Sensing and Actuation: One of the key functions of the SCE is to protect the *sensors* and *actuators*. Our framework does not allow the NCE to directly access such devices (e.g., passthrough I/O [17]) because of potential security risks. Instead, as typically done in virtualization, the I/O operations to/from the devices are emulated. Most sensors and actuators that we can find from UAS are typically interfaced through certain on-board peripheral communication protocols such as SPI (Serial Peripheral Interface) and I^2C (Inter-Integrated Circuit). Hence, one possible way is to implement back-end drivers for such common protocols at the virtual machine monitor (VMM) layer so that the applications running in the NCE can transparently use the (front-end) drivers already provided by the guest OS. However, this approach poses significant challenges to security and complexity for the following reasons:

- An attacker running in the NCE might reconfigure some sensor devices (e.g., changing the sampling rate or gain) in use. Hence, we would need a proper way to filter out impermissible I/O transactions from the NCE. This requires the emulation interface to have a comprehensive map of device registers that specifies write and read permissions for the NCE. Furthermore, some permission assignments may need to change dynamically depending

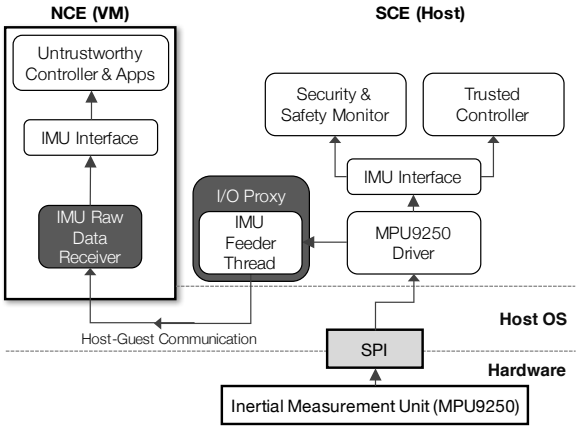

Figure 4: Sensor virtualization in the VirtualDrone framework.

on the current high-level context of each I/O operation, which is hidden in the low-level I/O emulation.

- The attacker may even attempt denial-of-service attacks on the shared devices by simply keeping them busy.
- Allowing low-level I/O transactions also requires a tight synchronization between the NCE and the SCE because some I/O operations are *stateful*. For instance, a compass sensor used in our prototype performs a register read in two stages – the device driver initializes a read by writing the read address, waiting for 10 ms, and then collecting the data. GPS parsing is also done with a state machine. Without a proper synchronization, the state of the device could be lost due to interleaving transactions from the NCE and the SCE. An attacker in the NCE could use this very property to hinder a timely, correct use of sensor data by the SCE (e.g., attacker may reinitialize a sensor while the SCE is waiting for a sample). If a synchronization is to be enforced, the availability could significantly degrade because the device should be locked for a long time to serve one at a time.

To solve the challenges illustrated above, we take a more *passive* approach to sensor and actuator virtualization. The framework runs an *I/O proxy* for each device in the SCE that *feeds* sensor data to the NCE. This is a suitable mechanism because of the periodic nature of their operations and the small data sizes that we can find from typical sensor and actuator devices on UAS platforms (see Table 1 in Section 4). Since it is known and fixed how often each sensor data should be sampled, the I/O proxy only needs to make sure that the SCE feeds the sensor data in time. It feeds raw sensor data instead of processed ones so that the applications in the NCE can process them as needed. From the perspective of the NCE, this sensor feeding looks as if the data is sampled from invisible, inaccessible devices (i.e., virtual sensors). Hence, the SCE can also remove certain side-channels (e.g., radio signal strength indicator) present in sensor information by not providing it to the NCE especially if this would not degrade the normal functionality of the NCE.

Figure 4 illustrates how an IMU (Inertial Measurement Unit) sensor is virtualized in our prototype based on an open-source autopilot suite [1]. A user-level device driver (MPU9250 Driver) uses the SPI interface to fetch raw IMU data. The IMU I/O proxy runs a feeder thread which sends the current sample to the *raw data receiver* running inside the NCE through a host-to-guest communication interface. The receiver behaves as a device driver from which the higher-level interface (i.e., IMU interface) can fetch sensor data, without needing to know the particular model of the real device. Hence, no modifications are needed in the higher layers. Actuators (e.g., motors) are virtualized in a similar way as explained in Appendix A.

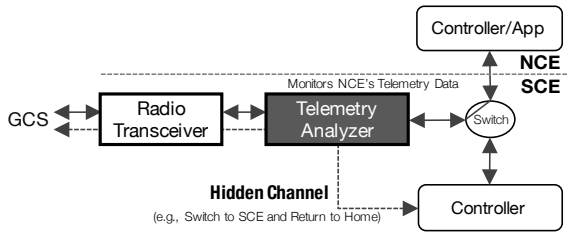

Figure 5: Telemetry virtualization creates a hidden communication channel between the SCE and the GCS.

Virtual Communication: Applications running in the NCE require network communication with the external world for data transfer, remote management, update, etc. Hence, as shown in Figure 3, the NCE allows users to directly access into the NCE through, for example, a port forwarding mechanism provided by the VMM. However, there exist some communication channels that need to be managed by the SCE – the ground control and the remote controller.

The ground control application utilizes a radio communication channel to establish a telemetry transfer between the ground control station (GCS) and a vehicle. The telemetry data, such as the GPS location and sensor measurement, can be monitored at the GCS. It can also use telemetry to dynamically control the vehicle by sending commands for setting new waypoints, landing, arming, etc. However, this communication channel can be exploited by an adversary who can simply disconnect the channel by disabling the radio driver. Hence, we virtualize the telemetry communication channel as done for the sensor and actuator.

The key benefit of this mechanism is that it can create a *hidden* communication channel for the SCE, as illustrated in Figure 5. For example, the GCS can send a special command to the SCE, e.g., switching to the SCE and returning to the home, which is filtered by the telemetry analyzer (part of the telemetry I/O proxy) that inspects every incoming message. These hidden messages are not relayed to the NCE. We take advantage of this mechanism as a solution to drone hijacking scenario presented in Section 5.1; upon a detection of a hijacking attempt, the GCS commands the SCE to switch to the host control mode and to return to where it is launched. We can use the same mechanism to reboot the virtual machine (after the control is switched to the SCE) from the GCS when a suspicious behavior is observed. Note that an attacker can send these special commands. Hence, such commands should be chosen carefully in such a way that a successful attempt cannot lead to a safety hazard.

A hand-held remote controller (see Figure 11 in Section 5) is used by a human pilot to wire-lessly fly a vehicle. The communication is also carried via a radio link (e.g., 2.4 GHz). It is virtualized in the same fashion as the telemetry radio, and we can also create a hidden channel. For instance, in our prototype implementation, we bind one of the switches of the remote controller with the function that tells the VirtualDrone to switch to the SCE. The pilot uses this function to manually control the vehicle when the NCE-driven vehicle shows abnormal behavior.

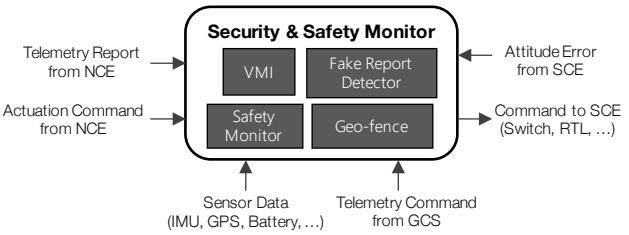

Figure 6: Example configuration of the security and safety monitor.

3.4 Security and Safety Monitoring

Figure 6 shows an example configuration of the security and safety monitor and data flow, based on the prototype implementation presented in Section 4. Notice from the figure (and also from Figure 4) that the monitor receives the sensor data for analysis. Because the data are fetched from the trusted environment, the monitor is guaranteed to use the true measurement for a safety analysis. Hence, we can detect attacks that, for example, try to put the vehicle in an open-loop state or to set wrong control parameters. In our prototype quadcopter, we analyze the attitude (i.e., roll, pitch, and yaw) errors, which are bounded in normal conditions, to detect an unsafe physical state. One can also implement a control-theoretic analysis [27]. The monitor also analyzes the actuation outputs from the NCE, as shown in Figure 16 in Appendix A, to prevent potential safety violations. For example, the monitor can upper-bound on the motor outputs to prevent motor failure due to the attacker's attempt to apply abrupt voltage changes. The monitor can also check if it receives actuation commands from the NCE at the defined frequency – abnormal patterns could be an indicator of a potential security breach. In order to handle physical attacks to the sensors, one can also implement a sensor attack detection technique [18, 19] using the true measurements available in the SCE.

The SCE also inspects communications between the NCE and the external world. The telemetry analyzer, shown in Figure 5, intercepts radio telemetry messages to and from the NCE and provides them to the monitoring module for analysis. Using this mechanism, one can detect an attacker that, for example, sends out fabricated messages (e.g., flight path report replayed or generated by a software-in-the-loop simulation) in an attempt to misinform the ground control station about the true physical and logical states.

The monitoring module can also host critical safety measures that otherwise would run at the same layer as untrustworthy applications. For instance, geo-fencing typically runs as a part of an autopilot software. An attacker can simply disable it or modify the configuration to fly the vehicle into a no-fly-zone. Since this type of function does not require external interaction, it can run in the SCE using the true sensor measurements (e.g., GPS location).

Virtualization also enables the use of virtualization-based security measures, for instance, virtual machine introspection (VMI) [14]. The monitoring module in the SCE can implement various VMI techniques to monitor the behavior of the applications and the guest OS running in the NCE. For example, through interfaces provided by the VMM, the module can continuously check the integrity of the kernel code and/or its critical data structures (e.g., interrupt vector table, process and module lists) for detection of rootkits, and inspect network connections and packets for detection of backdoors, and so on. Upon a detection, the framework may switch to the SCE as a preventive action, from which moment comprehensive

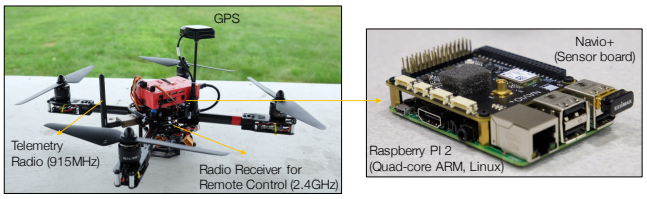

Figure 7: Quadcopter prototype implementation with Raspberry Pi 2 and Navio+ sensor boards.

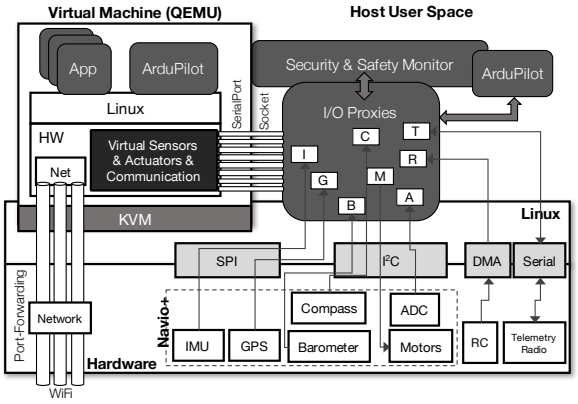

Figure 8: VirtualDrone implementation on the prototype.

security analyses (e.g., deep memory scan, rebooting VMs) can take place without losing control of the vehicle.

4 IMPLEMENTATION

In this section, we present the implementation details for our prototype of the VirtualDrone framework on a quadcopter drone, shown in Figure 7, running an open-source autopilot on an embedded computing board.

System Implementation: We implemented the VirtualDrone framework on a Raspberry PI 2 Model B (RPI2) board [5], as depicted in Figure 8. It has a quad-core ARM Cortex-A7 CPU, each core of which runs at 900 MHz, and has a main memory of 1 GB. The processor features the ARM Architecture Virtualization Extension. It enables running VMs with unmodified guest OS using KVM (Kernel-based Virtual Machine).

On the host, we run Linux 3.18. No modification was made to the kernel, except that we enabled virtualization with KVM in the configuration. On top of the host OS, we run unmodified QEMU v2.3 [4], an open-source virtual machine monitor. As explained in Section 3.2, QEMU, i.e., the VMM, is our trusted computing base (TCB). We created one VM that emulates an ARM Versatile Express A15 board (which runs a Cortex-A15 CPU, the same ARMv7 architecture as Cortex-A7) and assigned one of the four cores of the processor exclusively to the virtual machine. In the virtual machine, we run unmodified Linux 4.3.

We use QEMU's port forwarding mechanism to open an SSH (Secure Shell) port on the virtual machine, through which a user logs in to launch, update, manage services and applications. More ports can be open to the virtual machine using the port forwarding mechanism. In this paper, we assume the QEMU's port forwarding mechanism is secure.

In our prototype, we chose to use Linux as the host OS. However, it is desirable to use a formally verified OS on the host instead of

Table 1: The rate and amount of data transfer between the secure and normal control environments.

Component	Direction	Rate	Size
IMU	Host → VM	200 Hz	14 bytes
Barometer	Host → VM	25 Hz	4 byte
Compass	Host → VM	50 Hz	24 bytes
ADC	Host → VM	1.67 Hz	5 bytes
GPS	Host → VM	5 Hz	Max 1K bytes
Motor Output	VM → Host	200 Hz	16 bytes
RC Input	Host → VM	55 Hz	16 bytes
Telemetry	Host ↔ VM	Vary	Vary

such a general-purpose, monolithic kernel. Our choice of Linux is to minimize engineering efforts to port VMM, autopilot, and sensor and actuator device drivers to a new OS. Our implementation did not require any modifications to the host OS, VMM, and guest OS.

Autopilot: We stack Navio+ sensor board [2] on top of the Raspberry PI 2 (as shown in Figures 7 and 8) to provide various sensor data for flight control. The NCE (i.e., the virtual machine) runs the open-source ArduPilot (a.k.a. APM) [1] autopilot suite as the flight control software for our quadcopter drone. APM combines sensor data and RC flight maneuver commands to compute correct inputs for the four motor-prop units, which are sent by the actuator ports of the Navio+ board to drive the actuators.

As shown in Figure 8, we run one instance of APM in the SCE (i.e., the host). The Simplex architecture suggests the use of a robust controller for the safety controller in order to handle and recover from physical failures. For demonstration purposes, we chose to use the APM's default PID controller as the trusted controller. In a production implementation, however, a high-assurance controller would be used.

Virtualization: The I/O proxy, shown in Figures 4 and 8, runs one feeder thread for each sensor. Each sensor retrieves a new sample at a fixed frequency. Each feeder thread then sends the newly available data to the virtual machine using a host-guest communication channel. Appendix B explains how the sensors and the actuators are virtualized in our prototype implementation, which are also summarized in Table 1.

Host-Guest Communication: We use QEMU's `virtio-serial` for data transfer between the host (i.e., I/O proxy threads) and guest systems. It creates *virtual serial ports* in the guest, each of which is mapped to a character device such as Unix domain socket, pipe, TCP/UDP port, etc. in the host side. We created eight virtual serial ports for the components listed in Table 1. The I/O proxy threads in the SCE use eight Unix domain sockets to communicate with the virtual machine. One may use other types of host-guest communication mechanisms such as shared memory. In our implementation, we aimed to utilize an existing infrastructure that does not require any modification or insertion to the stock QEMU. The `virtio-serial` is adequate enough to handle the low-speed, low-volume data transfer for sensor/actuator/communication virtualization. We assume `virtio-serial` is trustworthy as it is part of QEMU, the TCB.

Virtual Telemetry: The APM uses a serial port (UART) for telemetry radio transceiver.[1] Hence, the NCE-side APM does not need any addition/modification for the virtual telemetry. The SCE-side APM reads/sends telemetry data from/to the real UART port, performs a filtering (as described in Figure 5), and relays to/from the virtual machine using the virtual-serial port mechanism explained above. The APM uses MAVLink protocol (Micro Air Vehicle Communication Protocol).[2] MAVLink contains all interface functions to control the vehicle, monitor states, change parameters, etc. Each MAVLink message size and frequencies vary depending on the message type. The messages typically have small size (the maximum is 263 bytes) and low frequency (a few Hz).

[1] Our prototype copter also supports WiFi, and APM's telemetry can be transferred through UDP or TCP as well. The APM abstracts these away by treating them as UART communication.

5 EXPERIMENTS

We now present case studies that demonstrate how the VirtualDrone framework can detect and prevent various types of security and safety violations.

5.1 Case Study

To demonstrate the effectiveness and versatility of our framework, we consider the following five scenarios. Note that we are not proposing new detection/mitigation solutions for the use cases presented here. Each scenario could be handled in many different ways. For instance, hijacking can be easily defeated by a simple authentication of the communication. We instead intend to demonstrate how an integrated platform, i.e., the VirtualDrone framework, can defend against various attack scenarios that would otherwise have been handled separately with potentially conflicting requirements. We also note that these attacks can occur in a number of different forms. In this paper, we consider a pessimistic scenario; the attacker can know the HW/SW configuration and even gain a root access to the NCE.

Attacks on Safety: An adversary can launch an attack on the safety of a vehicle by, for example, degrading the availability of critical sensors (e.g., IMU) or actuators, or the control performance (e.g., by changing PID gains). The worst-case scenario, from the vehicle's safety perspective, is when the attacker disables the flight controller while the vehicle is flying. This immediately leads the system to an *open-loop* state, which will cause the vehicle to crash. To demonstrate this type of attack, we consider an extreme scenario in which the flight controller is killed by an attacker. The attacker can launch this attack by, for example, entering through a backdoor, replacing the control program with one that self-crashes, or installing a rootkit. We do not assume specific scenario of how it is launched. We implemented a Linux kernel module that 1) finds the APM autopilot process from the kernel's process list and then 2) kills the process. The attacker could achieve the same goal by causing the virtual machine to crash.

There could be several ways to detect this type of attack. One may use a heartbeat mechanism, which however can be circumvented by an attacker that impersonates the flight controller. Instead, we take advantage of the SCE's ability to monitor the *true physical state* of the vehicle. We use the attitude control performance measured at the SCE. At each control loop, the SCE-side controller calculates the errors of the rate control on the pitch, roll, and yaw of the copter. During a stable flight, the rate errors are bounded, as shown in Figure 17 in Appendix C, because the flight controller is active to minimize the rate errors. In case of an open-loop state, the flight controller cannot work to minimize the rate errors. Due to the fact that a multirotor system is naturally an unstable flight platform,

[2] http://qgroundcontrol.org/mavlink/start

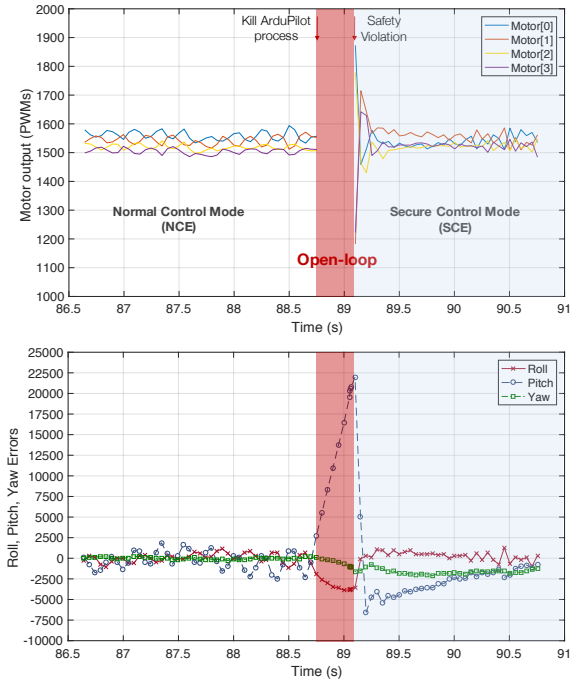

Figure 9: The motor outputs (top) and the roll, pitch, yaw errors (bottom) measured at the SCE. The attacker in NCE kills the flight controller, which leads the copter to an open-loop state. The SCE takes the control when the errors grow beyond the thresholds, after which the copter is stabilized.

the rate errors will increase quickly beyond the normal bound. Therefore, a properly chosen threshold of rate errors can be used in detection of flight safety violations. Hence, the security and safety monitoring module in the SCE continuously checks if the errors grow beyond the threshold. Upon a safety violation, the framework switches to the secure control mode. Appendix C explains how the APM's attitude control works and how we obtained the threshold of rate errors.

Figure 9 shows the results of this experiment. While the drone was in the virtual control mode, the attacker activated the rootkit mentioned above at time around 88.8 sec, at which moment the APM process running in the VM is killed. The top plot in Figure 9 shows the motor outputs (4 channels) from the motor driver in the SCE. As we can see, the drone was in an open-loop state for about 300 ms. The bottom plot shows the attitude errors also measured at the SCE. The drone becomes unstable (i.e., the errors are far away from zero) for a moment because no actuation is applied to the motors during the open-loop period. Upon the detection of the violation on the errors (at time around 89.1 sec), the control is switched to the SCE from which moment the control loop is closed and the drone returns to a stable state.

We also tested a scenario when the control parameters are modified during flight. Figure 10 shows the attitude control errors when this happens. At time around 164.3 sec, a MAVLink message was sent via radio to change the proportional gain (180 times bigger than the original value) of the attitude controller. As can be seen, the drone became unstable immediately. The safety module detected the large roll errors, after which the SCE took the control.

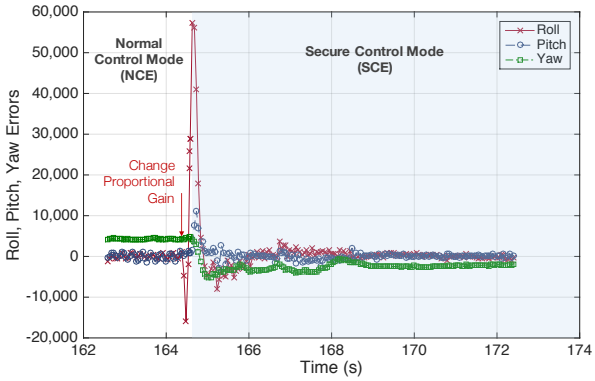

Figure 10: The roll, pitch, yaw errors when the proportional gain of the attitude controller is set to an abnormally high value. The SCE immediately stabilizes the drone upon the detection of the unstable physical state.

Hijacking: It has been demonstrated that it is possible to hijack a drone by exploiting the MAVLink protocol [9]. The idea is to send a command that sets a new flight plan through the telemetry channel. The telemetry radio pair of the drone and the ground control station (GCS) distinguish themselves from others by their unique NetID and the frequency band. Hence, by using the same NetID and radio band and running the same firmware (which decodes the MAVLink packets) as the target vehicle, the attacker can send any MAVLink messages to the target.

To demonstrate this attack scenario, we used three telemetry radio (i.e., for the GCS, the drone, and the attacker) that utilize the same frequency band (915MHz) and same default configuration. Since the out-of-the-box default NetIDs are same, the attacker's telemetry radio did not need any firmware modification.[3] Figure 11 describes the hijacking scenario. The drone takes off at the launching point (marked as 'T') and communicates with the GCS located at 'G'. The pilot flies the drone along the normal path. The attacker, located at 'A', then launches its ground control application. Then, it sends a new waypoint plan to the drone, which will send it to the location marked as 'W'. The attacker did not need to modify the

[3]It is possible to find out the NetID used between the GCS and the drone by modifying the firmware [9].

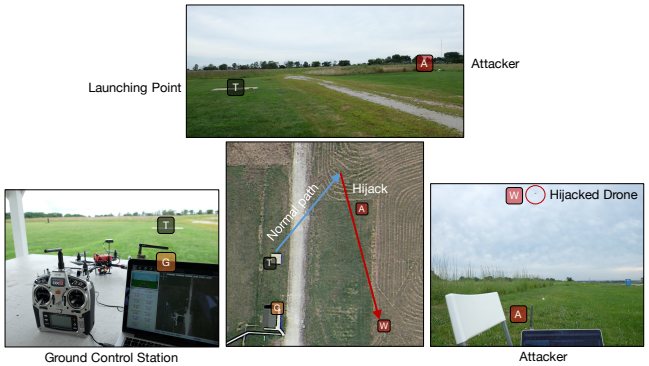

Figure 11: The attacker ('A') tries to hijack the drone flying along the normal path and to send it to a new waypoint ('W'). The attacker uses the same telemetry radio as the GCS and the drone, and unmodified APM Planner GCS.

(a) Without VirtualDrone (b) With VirtualDrone

Figure 12: (a) The flight trajectory of the drone hijacked by the attacker. It is sent to new location 'W' set by the attacker. (b) The attacker's attempt to hijack the drone is detected by the GCS. Upon the detection, the GCS sends a special command to the SCE, which switches the control mode to the SCE and directs the drone to where it was launched.

GCS program (the stock APM Planner 2.0[4] with default settings). Figure 12(a) shows the flight trajectory of the drone when it was successfully hijacked by the attacker.

While the drone itself cannot detect such a hijacking attempt, the GCS can detect it because of *unexpected message exchange* initiated by the attacker, as detailed in Appendix D. Hence, we added the functionality that detects such unexpected messages to our legitimate GCS. Upon a detection, the GCS commands the drone to return to where it was launched, as shown in Figure 12(b). This takes advantage of the VirtualDrone's *virtual telemetry* explained in Section 3.3; the SCE's telemetry proxy enables a hidden communication channel between the GCS and the SCE, through which the former sends the drone a pre-defined set of special commands. In this scenario, the command from the GCS overrides the NCE's abnormal operation by switching the drone to the secure control mode. Note that the attacker might be able to send the same special command. However, what it can do at worst is to send the drone back to the home.

As explained in Section 3.4, the virtual telemetry also enables the SCE to detect if the NCE is trying to deceive the ground station. For example, after a successful hijacking, the attacker may report fake GPS location (by replaying or by generating from SW simulator) to the GCS so that it looks as if it is flying on the planned path. The telemetry analyzer, however, can detect such attempts by analyzing each outgoing MAVLink packet and comparing against the true location retrieved from the GPS receiver, which is also implemented on our prototype.

Corrupting Safety Functions: Autopilot programs also support critical failsafe mechanisms that are activated under certain conditions such as losing radio communication signal or low-battery. It is in fact easy to corrupt such a safety-critical measure because typically it can be enabled/disabled remotely through a telemetry command (hence, an attacker can send a command via radio, as done in the hijacking scenario). Moreover, such a mechanism is often implemented as a part of autopilot that runs at the user-level. Hence, an attacker who has gained a root access can easily corrupt it by modifying the configuration file or the memory.

[4]http://ardupilot.org/planner2/index.html

Figure 13: Attacker disables the geo-fence and modifies Waypoint 3 so that the drone flies into a no-fly zone. Such a safety-critical function can be protected by running in the SCE.

In this case study, the attacker corrupts the *geo-fence* mechanism. Geo-fence employs positioning data such as GPS signal or local radio-frequency identifiers to set up a virtual boundary to prevent the vehicle entering a prohibited zone. The vehicle position is monitored at all times such that before the vehicle enters a no-fly zone, the system could act accordingly to prevent a geo-fence violation. Corruption of the geo-fence can result in a catastrophic result; an attacker may disable the geo-fencing and induce the vehicle to fly into a congested airspace such as takeoff pathways for mid-air collision.

To demonstrate this type of attack, we developed a rootkit that disables the geo-fence of APM during a flight, and sends it into a no-fly zone. The rootkit executes this attack by modifying the APM's *memory*, as detailed in Appendix E. Our drone was planned to fly through a path (0-1-2-3) in an autonomous mode as shown in Figure 13. During the flight, the attacker logged into the system through WiFi and launched the rootkit when the drone was flying toward Waypoint 2. The new Waypoint 3 set by the rootkit is located inside a no-fly zone, and because the rootkit has already disabled the geo-fence, the drone consequently flied into the no-fly zone, deviating from the original path, as shown in Figure 13.

The SCE of VirtualDrone provides a protected layer at which safety-critical functions like geo-fence can be placed. We implemented a simple geo-fence module in the security and safety monitoring module in the SCE. It continuously monitors the current GPS coordinate and checks it against the list of no-fly zones also stored at the SCE layer. Upon a violation, a pre-defined action is taken. In our implementation, the SCE takes back the control from the NCE and returns to where it was launched, as done for the hijacking scenario explained earlier.

Side-channel: The virtualization of sensor, actuator, and communication allows for hiding certain types of information from the NCE. One of the examples is the RSSI (Received Signal Strength Indication) that indicates the link quality between a pair of radio transmitter and receiver. We especially consider a scenario in which an attacker tries to estimate the location of the GCS by observing the RSSI measured between the vehicle and the GCS. Due to signal attenuation, the radio signal is stronger as the vehicle is closer to the GCS. Hence, one can correlate the RSSI with the GPS coordinate.

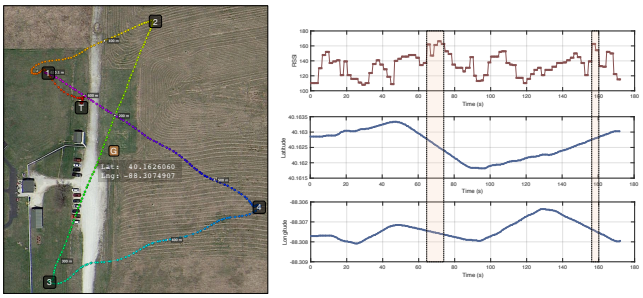

Figure 14: The drone flies through T-1-2-3-4-1. The graphs represent the RSSI (top), the latitude (middle), and the longitude (bottom). An attacker can estimate the location of the GCS by correlating the RSSI and the GPS information.

Figure 14 illustrates such a possibility of side-channel. The graphs in the figure represents the RSSI (between the drone and the GCS) and the GPS coordinate (latitude and longitude) measured while the drone flies through a path (T-1-2-3-4-1). If the attacker is able to obtain these RSSI information, he/she can estimate the location (or a region) of the GCS by finding when the RSSI is high. The results show quite accurate estimation of the true GCS location (shown in the map). A complex algorithm will allow for further narrowing down the location. The VirtualDrone framework eliminates this possibility by not providing the RSSI information to the NCE. Note that RSSI is needed only for the SCE (e.g., switches to the SCE and then performs a pre-defined operation such as return-to-home when RSSI is low due to the loss of telemetry link). Furthermore, because of the telemetry virtualization, the NCE cannot even know if the telemetry is communicated through a radio channel or a network (e.g., WiFi). In case of the network-based telemetry, the SCE can even hide the IP address of the GCS.

Virtual Machine Introspection: Many rootkits modify critical kernel data structures to intercept sensitive data, hide malicious processes or files, etc. For a demonstration, we implemented a security module that checks the integrity of the system call table of the guest OS. For this, we chose to utilize an existing interface provided by `QEMU`, namely the QEMU Machine Protocol (`QMP`). It allows host-side applications to communicate with or control a running `QEMU` VM. We created a `QMP` Unix socket to which our security module can connect. During the initialization of the VM, the module dumps the current system call table and stores this initial state in memory. From then on, the module regularly dumps the current table and compares it against the one obtained initially. Using this technique, we were able to detect a known rootkit, `modhide1`, that hijacks the open call to hide itself from the kernel module list.

Note that we are not proposing new rootkit detection methods here. We instead intend to demonstrate how the VirtualDrone framework can handle such a stealthy security violation. One can use a rich set of library for virtual machine introspection such as `LibVMI` [3], which we leave for future work.

5.2 Discussion

Sensor attack: The VirtualDrone framework cannot handle physical manipulations on the sensors such as GPS spoofing. These types of attacks are called *sensor attack* or *false data injection attack*, and typically tackled by control-theoretic approaches [18, 19]. The requirement, however, is that the methods themselves should be protected from cyber-attacks. Hence, one can implement such a technique on the SCE layer, specifically in the security and safety monitoring module, as it can see the true (but potentially physically manipulated) sensor measurements.

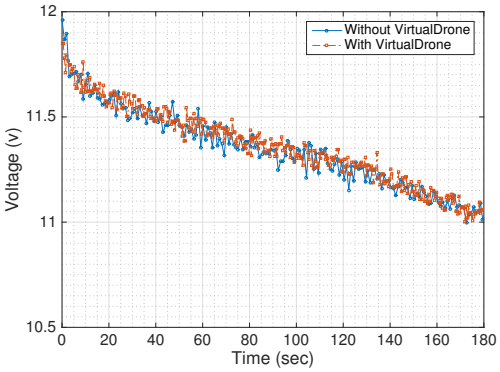

Figure 15: Voltage drop at the battery for 3-minutes of hovering with and without the VirtualDrone framework.

Power consumption: Since UAS typically runs on battery power, we compare the power consumption of the prototype drone with and without the VirtualDrone framework. In order to compare the power consumption in a controlled environment (e.g., eliminating varying disturbance due to wind), we flew the drone indoors, hovering it at a fixed position. We flew the drone for about 3 minutes with the same fully-charged battery and measured the voltage drops during the flight.

Figure 15 shows that the VirtualDrone does not impose overhead on the power consumption. This is because the majority of the power is consumed by *the motors to lift* the copter. The power consumption by RPI2 board and the Navio+ sensors cannot exceed 5 Watts which is the upper bound of the power supply by the power module. That is, the power consumption by any on-board computing cannot be more than 5 Watts. We calculated the average power of the 3-minutes of flights, which ranged between 114 and 118 Watts. Hence, the power consumption by any on-board computing can be ignored. Due to the fact that the upper bound of power consumption by computing is two orders of magnitude smaller than that of the motors, we conclude that the power consumption overhead of running the VirtualDrone is negligible.

6 RELATED WORK

Current unmanned vehicle systems are very vulnerable to cyber-attacks as demonstrated by recent attacks. Maldrone [8] is a software virus that can compromise drones based on ARM Linux systems. The malware can open a backdoor in the Parrot AR Drones, infect on-board software and take over the control. Pleban et al. [23] presented analysis details on the insecure WiFi network and OS user management of the Parrot AR Drones. Also, researchers demonstrated a hijacking of DJI consumer drones by emulating fake GPS signals using low-cost software defined radio tools [11]. It is also possible to inject MAVLink message into a radio link by modifying the radio firmware, and hijack a flying drone [9], which we reproduced for our case study. Javaid et al. [15] addressed some vulnerabilities of wireless communications channels in unmanned aerial vehicles. Commercial airplanes are also facing with such security problems. The Actel ProASIC chip used in early Boeing

787 had a backdoor [6] that could allow an attacker, via internet connection or as a passenger, to use entertainment system in the aircraft to take over the control of the aircraft.

Advances in virtualization technologies have enabled Virtual Machine Introspection (VMI) [14], in which a trusted VM or the VMM monitors and analyzes the state of applications or OS running inside untrusted VMs. By placing the detection mechanism outside of the VM it monitors (i.e., the untrusted VMs), VMI overcomes the vulnerability of the traditional host-based intrusion detection systems. VMIs have been applied to process execution monitoring [25], control-flow integrity check [22], virtual memory and disk monitoring [20], dynamic information flow tracking [29], etc.

Hardware-based approaches can also be used to isolate vulnerable components. ARM TrustZone [28] is an architectural extension that allows a coexistence of secure world and normal world using the same on-chip hardware. Access to certain secure components are blocked depending on the execution mode. Azab et al. [13] proposed TZ-RKP which protects the integrity of operating system kernel that runs on a normal world by running a security monitor in the secure world. Zhou et al. [33] use ARM processors memory domain support for software fault isolation. Yoon et al. [31] takes advantage of the redundancy of a multicore processor; a core is used to monitor the execution time of a real-time application running on a monitored core. It was extended to monitor memory behavior for system-wide anomaly detection [32] and to protect in-place security monitoring module [30]. These hardware-based approaches provide better security than virtualizations can do because the latter relies on the security and correctness of the VMM which is susceptible to attacks [21]. However, virtualization-based methods have advantages in that no hardware modification is needed and that they are more flexible due to the software control.

7 CONCLUSION

In this paper, we presented the VirtualDrone framework as a solution to increasing security threats to unmanned aerial systems. The use of this framework allows system designers (or users) to run advanced flight applications in an untrustworthy environment. Our prototype implementation requires a minimal effort to build the framework on an off-the-shelf platform with open-source software stack. Through case studies, we demonstrated that the framework provides an integrated way to handle various security threats that would otherwise have required potentially conflicting requirements. Nevertheless, there still remains challenges for practical use of the framework. We used a full-featured autopilot for both normal and safety controllers in the current implementation, simply for a demonstration purpose only. It is desirable to run a fully-verified control software in the SCE. Hence, we plan to adopt and implement a high-assurance controller.

REFERENCES

[1] ArduPilot Autopilot Suite. http://www.ardupilot.org
[2] Emlid NAVIO+. https://docs.emlid.com/navio/
[3] LibVMI. http://libvmi.com/
[4] QEMU. http://wiki.qemu.org
[5] Raspberry PI 2 Model B. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/
[6] Cyber-attack concerns raised over Boeing 787 chip's 'back door'. *The Guardian*. https://www.theguardian.com/technology/2012/may/29/cyber-attack-concerns-boeing-chip.
[7] 3DR's Solo Drone Boasts Dual Linux Computers Running Dronecode. https://www.linux.com/news/3drs-solo-drone-boasts-dual-linux-computers-running-dronecode
[8] A hacker developed Maldrone, the first malware for drones. *Security Affairs*. http://securityaffairs.co/wordpress/32767/hacking/maldrone-malware-for-drones.html.
[9] Hijacking drones with a MAVLink exploit. http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.
[10] Qualcomm Goes Ubuntu for Drone Reference Platform. https://www.linux.com/news/qualcomm-goes-ubuntu-drone-reference-platform
[11] Watch GPS Attacks That Can Kill DJI Drones Or Bypass White House Ban. *Forbes*. http://www.forbes.com/sites/thomasbrewster/2015/08/08/qihoo-hacks-drone-gps/#26431a2853fe.
[12] Intel Aero Compute Board. http://www.intel.com/content/www/us/en/technology-innovation/aerial-technology-overview.html
[13] Ahmed M. Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision Across Worlds: Real-time Kernel Protection from the ARM TrustZone Secure World. In *the ACM Conference on Computer and Communications Security*.
[14] Tal Garfinkel and Mendel Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *the Network and Distributed Systems Security Symposium*.
[15] Ahmad Y. Javaid, Weiqing Sun, Vijay K. Devabhaktuni, and Mansoor Alam. 2012. Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In *the IEEE Conference on Technologies for Homeland Security*.
[16] Andrew J. Kerns, Daniel P. Shepard, Jahshan A. Bhatti, , and Todd E. Humphreys. 2014. Unmanned Aircraft Capture and Control Via GPS Spoofing. *Journal of Field Robotics* 31 (2014).
[17] Jiuxing Liu, Wei Huang, Bulent Abali, and Dhabaleswar K. Panda. 2006. High Performance VMM-bypass I/O in Virtual Machines. In *the USENIX Annual Technical Conference*.
[18] Y. Mo and B. Sinopoli. 2015. Secure Estimation in the Presence of Integrity Attacks. *IEEE Trans. Automat. Control* 60, 4 (2015), 1145–1151.
[19] Miroslav Pajic, James Weimer, Nicola Bezzo, Paulo Tabuada, Oleg Sokolsky, Insup Lee, and George J. Pappas. 2014. Robustness of Attack-Resilient State Estimators. In *the ACM/IEEE International Conference on Cyber-Physical Systems*.
[20] Bryan D. Payne, Martim Carbone, and Wenke Lee. 2007. Secure and Flexible Monitoring of Virtual Machines. In *the Annual Computer Security Applications Conference*.
[21] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. 2014. On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment. In *the ACM Symposium on Information, Computer and Communications Security*.
[22] Nick L. Petroni, Jr. and Michael Hicks. 2007. Automated Detection of Persistent Kernel Control-flow Attacks. In *the ACM Conference on Computer and Communications Security*.
[23] Johann-Sebastian Pleban, Ricardo Band, and Reiner Creutzburg. 2014. Hacking and securing the AR.Drone 2.0 quadcopter: investigations for improving the security of a toy. In *the SPIE Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications*.
[24] Lui Sha. 2001. Using Simplicity to Control Complexity. *IEEE Softw.* 18, 4 (2001), 20–28. DOI: http://dx.doi.org/dx.doi.org/10.1109/MS.2001.936213
[25] Deepa Srinivasan, Zhi Wang, Xuxian Jiang, and Dongyan Xu. 2011. Process Outgrafting: An Efficient "out-of-VM" Approach for Fine-grained Process Execution Monitoring. In *the ACM Conference on Computer and Communications Security*.
[26] Nils Ole Tippenhauer, Christina Pöpper, Kasper Bonne Rasmussen, and Srdjan Capkun. 2011. On the Requirements for Successful GPS Spoofing Attacks. In *the ACM Conference on Computer and Communications Security*.
[27] X. Wang, N. Hovakimyan, and L. Sha. 2013. L1Simplex: Fault-tolerant control of cyber-physical systems. In *the ACM/IEEE International Conference on Cyber-Physical Systems*.
[28] Peter Wilson, Alexandre Frey, Tom Mihm, Danny Kershaw, and Tiago Alves. 2007. Implementing Embedded Security on Dual-Virtual-CPU Systems. *IEEE Des. Test* 24, 6 (Nov. 2007), 582–591.
[29] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. 2007. Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis. In *the ACM Conference on Computer and Communications Security*.
[30] Man-Ki Yoon, Mihai Christodorescu, Lui Sha, and Sibin Mohan. 2016. The DragonBeam Framework: Hardware-Protected Security Modules for In-Place Intrusion Detection. In *the ACM International Systems and Storage Conference*.
[31] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, and Lui Sha. 2013. SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems. In *the IEEE Real-Time Embedded Technology and Applications Symposium*.
[32] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, and Lui Sha. 2015. Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior. In *the ACM/EDAC/IEEE Design Automation Conference*.
[33] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-based Fault Isolation for ARM. In *the ACM Conference on Computer and Communications Security*.

A ACTUATOR VIRTUALIZATION

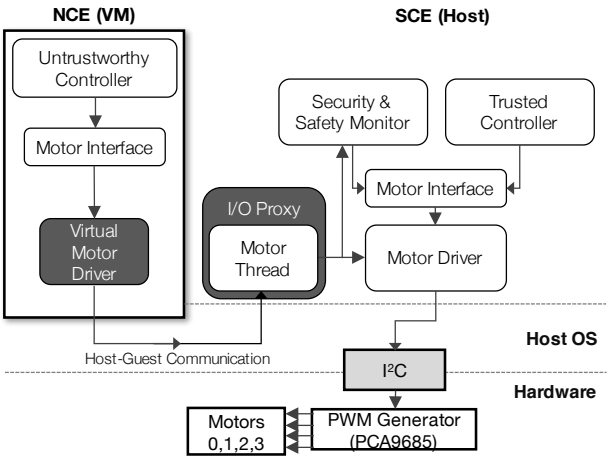

Figure 16: Actuator virtualization in the VirtualDrone framework.

Figure 16 shows how an actuator is virtualized in the VirtualDrone framework, which is similar to the sensor virtualization explained in Section 3.3. The controller running in the NCE computes a set of PWM (Pulse Width Modulation) output values to control the motors of the vehicle. The controller writes these values to the virtual motor driver through the motor interface. It transfers the data to an I/O proxy thread that relays the PWM values to the motor driver in the SCE. The motors are finally actuated by analog signals converted by the on-board speed controller.

B DETAILS ON VIRTUALIZATION

Sensors: The following describes how the sensors are virtualized in our prototype implementation.

- Inertial Measurement Unit (MPU9250): The device driver running on the host retrieves a raw IMU data (14 bytes) every 1 ms through SPI. However, every 5th sample is actually used for control (hence, the effective control frequency is 200 Hz). The data includes a 3-axis gyroscope and 3-axis accelerometer values.
- Barometer (MS5611): This sensor measures the barometric pressure and temperature. The driver runs a state machine that operates at a frequency of 25 Hz. A raw data (24 bits) retrieved from the sensor through I^2C is converted to either pressure or temperature value depending on the state (8 bits).
- Compass (AK8963): It measures terrestrial magnetism in the 3 axes at a frequency of 50 Hz using I^2C. Total 24 bytes (including per-axis calibration factor) of raw data are fed to the NCE.
- Analog-to-Digital Converter (ADS1115): It measures the voltages on 6 ADC channels. It reads each channel every 600 ms through I^2C. For each channel, a data of 40 bits (8 bits for channel ID and 32 bits for sampled data) is transferred to the VM.
- GPS (u-blox NEO-M8): It provides the current longitude, latitude, and altitude of the vehicle, the time information (current millisecond time of week), etc. The driver reads UBX protocol messages from the u-blox GPS receiver through SPI, parses each one, and then obtains the above information. The parsing is stateful, and a after-parsing data can be at most 1K bytes. It is fed to the NCE at the frequency of 5 Hz.

Actuator: For the motor actuation, the virtual controller in the NCE sends the PWM values (for the four motors) to the host at the frequency of the main control loop (i.e., 200 Hz).

Radio Control Input: A remote control via radio link is used to manually control (arm/disarm, flight maneuver) the quadcopter. The raw input pulses are retrieved through DMA (Direct Memory Access) at 1,666 Hz. About every 18 ms (i.e., 55 Hz), a new set of PWM values representing the stick and switch movements (total 8 channels) becomes available. As done for the sensors mentioned above, the I/O proxy runs an RC feeding thread. However, instead of feeding the raw pulse data, we send the processed data, i.e., the PWMs, in order to avoid the overhead due to feeding the raw data at such a high frequency (1,666 Hz).

C ATTITUDE CONTROL AND ERROR BOUND

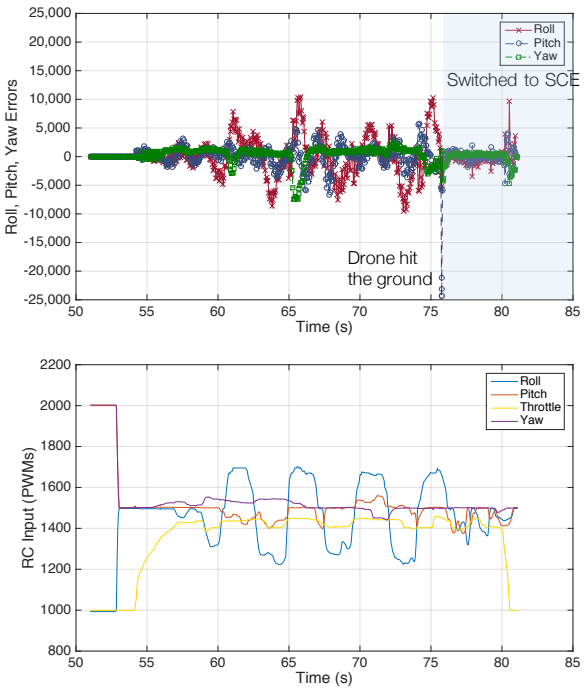

Figure 17: The roll, pitch, yaw errors (top) measured during the drone's extreme movements. The bottom plot shows the roll, pitch, throttle, and yaw targets set by the pilot using the remote controller.

The APM uses a double-loop PID/P control to stabilize the quadcopter. The *Angle loop*, a.k.a. the *outer* loop, controls the attitude of the vehicle. The Angle loop utilizes a P control to achieve the desired attitude by outputting a desired angular speed, i.e., the angular speed setpoint, to the *Rate loop*, a.k.a. the *inner* loop. The Angle loop's output, i.e., the angular speed setpoint, is proportional to the difference between the target angle value set by the pilot (or autopilot when in autonomous flight mode) and the measured angle value from the IMU sensor. The Rate loop controls the attitude rates of the aircraft. The Rate loop continuously calculates an error value as the difference between the angular speed setpoint and the measured angular speed from the IMU sensor. A PID control is

utilized in Rate loop to minimize the error over time by outputting PWM signals to control motors.

For the thresholds on the errors, we used ±20, 000 (rad/sec) for roll and pitch, and ±10, 000 (rad/sec) for yaw. The switching logic activates when the violation happens three times consecutively. We obtained these bounds on the errors by measuring from both normal and extreme movements of the prototype drone. The top plot in Figure 17 shows those errors measured during the drone's extreme movements and the bottom plot shows how the pilot created the movements. The result indicates that the errors are well bounded even when the drone experiences such movements. The pilot intentionally made the drone hit the ground at time between 75 sec and 76 sec, after which the control is switched to the SCE. We did not attempt to find optimal thresholds. The smaller the threshold is, the easier it is for the SCE to recover to a stable state. However, at the same time, it could create more false positives.

D DETECTION OF HIJACKING ATTEMPT

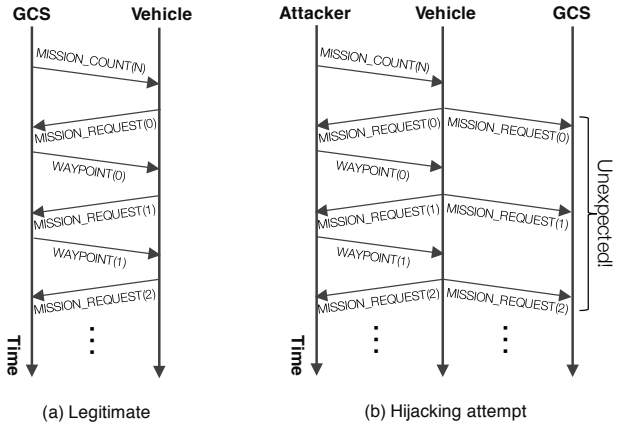

Figure 18: (a) MAVLink messages exchanged between the GCS and the vehicle for waypoint setup. (b) The legitimate GCS can detect the attacker's update on the vehicles route as the vehicle's responses are unexpected.

In order for a GCS to set waypoints for a vehicle, a series of MAVLink messages are exchanged. Figure 18(a) shows the MAVLink waypoint protocol. The GCS first sends `MISSION_COUNT(N)` to the vehicle where N represents the number of waypoints that it will set. The vehicle prepares for receiving the N waypoints, and then requests for each waypoint by sending `MISSION_REQUEST(i)` until all the waypoint locations are received.

Note that both the legitimate GCS and the attacker need to follow this protocol to set any waypoints. From the vehicle's perspective, the initialization requests (i.e., `MISSION_COUNT(N)`) from them are indistinguishable. That is, the vehicle itself cannot detect suspicious requests for a route change. However, the fact that the attacker can receive messages from the vehicle means that the legitimate GCS can also hear what the vehicle responds to the attacker's request. As Figure 18(b) shows the legitimate GCS can detect the attacker's update on the vehicle's waypoints when it receives *unexpected* `MISSION_REQUEST` messages as it did not initialize the message exchange. Due to such stateful communication, the MAVLink protocol enables detecting other types of suspicious attempts (e.g., changing control parameters) as well.

E GEO-FENCE ATTACK ROOTKIT

We developed a rootkit that disables the geo-fence and sends the vehicle into a no-fly zone. The rootkit finds the APM process, and then modifies its *memory* that stores (i) the flag that enables/disables the geo-fencing and (ii) the list of waypoints. These are loaded from the APM configuration file (`/var/APM/ArduCopter.stg`) on its startup and buffered in the memory.

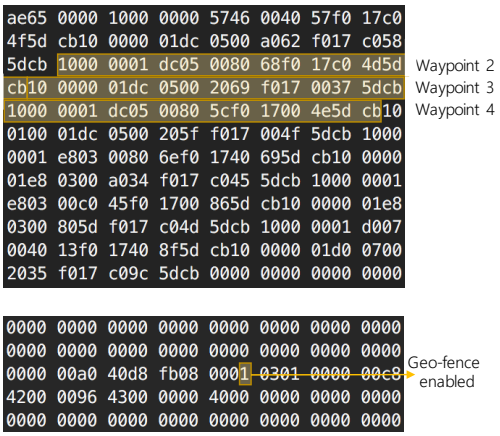

Figure 19: The waypoint list and geo-fence enable flag stored in the memory of APM process. An attacker who gained a root-level access can modify these memory values to disable the geo-fence and then send the drone to a new location.

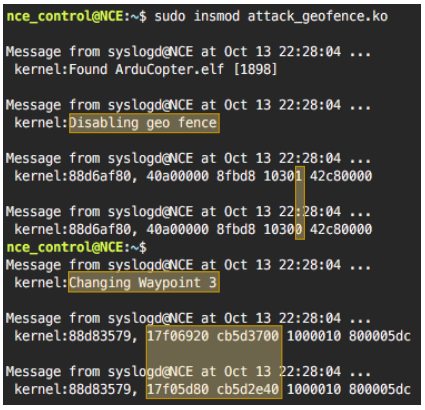

Figure 20: The rootkit that disables the geo-fence and modifies the flight plan by manipulating the APM's memory.

One can change these values remotely through radio, which however is easy to prevent since radio communication can be monitored easily. Modifying the buffered values in the memory is difficult to prevent especially if the attacker has gained a root access. The memory locations of these data can be easily found if the source code or the executable binary is available. Figure 19 shows the memory dump of the waypoint list and geo-fence enable flag. Figure 20 shows how these values are modified by the rootkit.