© 2017 by Man Ki Yoon.

SECURE AND DEPENDABLE CYBER-PHYSICAL SYSTEM ARCHITECTURES

BY

MAN KI YOON

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2017

Urbana, Illinois

Doctoral Committee:

Professor Lui Sha, Chair/Director of Research
Professor Carl Gunter
Professor Tarek Abdelzaher
Research Assistant Professor Sibin Mohan
Dr. Mihai Christodorescu, Qualcomm Research

Abstract

The increased computational power and connectivity in modern Cyber-Physical Systems (CPS) inevitably introduce more security vulnerabilities. The concern about CPS security is growing especially because a successful attack on safety-critical CPS (e.g., avionics, automobile, smart grid, etc.) can result in the safety of such systems being compromised, leading to disastrous effects, from loss of human life to damages to the environment as well as critical infrastructure. CPS poses unique security challenges due to its stringent design and implementation requirements. This dissertation explores the structural differences of CPS compared to the general-purpose systems and utilizes the intrinsic characteristics of CPS as an asymmetric advantage to thwart and detect security attacks to safety-critical CPS. The dissertation presents analytic techniques and system design principles to enhance the security and dependability of CPS, with particular focus on (a) modeling and reasoning about the logical and physical behaviors of CPS and (b) architectural and operating-system supports for trusted, efficient run-time monitoring as well as attack-resiliency.

To my father and mother who have waited for a long time with love and belief.

Acknowledgments

I would like to express my deepest gratitude to my advisor, Prof. Lui Sha, for giving me the opportunity to study under his guidance and instruction. He has supported me with endless patience, generosity, encouragement, and invaluable advice from his own inspiring experience. He has taught me not only how to do research but, more importantly, how to be a good researcher and teacher, which will guide me through my academic journey. I would also like to thank Sibin Mohan for introducing me to the area of real-time systems security and for helping me enjoy this exciting field. I am also grateful to my other committee members, Prof. Carl Gunter and Prof. Tarek Abdelzaher, for their valuable feedback and guidance that have improved the quality of this dissertation. I also thank Mihai Christodorescu for mentoring me through the Qualcomm Innovation Fellowship and my internship at Qualcomm Research. I am also grateful to Jaesik Choi for helping me get to know the power of machine learning. I would like to thank Prof. Chang-Gun Lee who guided me to pursue research in the field of real-time systems in the first place. I would also like to thank my internship mentors - Gabriela Ciocarlie at SRI International who gave me the opportunity to have my first Silicon Valley experience, Negin Salajegheh and Yin Chen at Qualcomm Research who helped me go through a challenging yet exciting project, and Michael LeMay at Intel Labs who gave me an opportunity to work on a unique project that I would never have experienced elsewhere. I am also very grateful to Bo Liu who has provided me with great technical supports for the drone project and helped me get to like the field of robotics. I would like to thank Min Young Nam who sincerely helped me settle down in this new place and also in school life.

I would like to acknowledge the National Science Foundation, the Office of Naval Research, Rockwell Collins, and Lockheed Martin for the funding that have made my research possible. I am also grateful to Qualcomm Research and Intel for awarding me the fellowships that not only financially supported my research but also encouraged me to pursue my goals with confidence.

Finally, no single page of this dissertation would have been written without Jung-Eun's supports both at and away from school. I am truly grateful to her for having always been with me, tasting the sweets and bitters of life together. I could not have endured this long journey without her.

Table of Contents

Chapter 1 Introduction **1**
1.1 Overview of Research 1
1.1.1 Research Goal and Challenges 1
1.1.2 Summary of Solutions 3
1.2 Background and Related Work 5
1.2.1 Security Attacks to CPS 5
1.2.2 Simplex Architecture 6
1.2.3 Behavior-based Intrusion Detection 7
1.2.4 Hardware-based Security Measures 9
1.2.5 Virtual Machine Introspection 10

Chapter 2 SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems **11**
2.1 Introduction 11
2.1.1 Assumptions 12
2.1.2 Motivation 13
2.2 SecureCore Architecture 14
2.2.1 High-Level Architecture 14
2.2.2 Design Considerations 15
2.2.3 Timing Trace Module (TTM) 17
2.3 Gaussian Kernel Density Estimation for Execution Time-Based Intrusion Detection 19
2.3.1 Overview 20
2.3.2 Trace Tree 21
2.3.3 Profiling Block Execution Time Using Gaussian Kernel Density Estimation 22
2.3.4 Intrusion Detection Using Execution Time Profiles 23
2.4 Implementation 24
2.4.1 System Implementation 25
2.4.2 Application Model 26
2.5 Result and Discussion 28
2.5.1 Early Detection of an Intrusion 28
2.5.2 Intrusion Detection Accuracy 29
2.5.3 Limitations and Possible Improvements 30
2.6 Conclusion 31

Chapter 3 Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior **32**
3.1 Introduction 32
3.2 The Memory Heat Map 34
3.2.1 Monitoring Kernel Memory Space 35
3.2.2 Overall Process 35
3.2.3 Assumptions 36
3.3 Monitoring Memory Heat Maps 36

3.3.1 Memometer . . . 36
3.4 Learning Memory Heat Maps . . . 39
3.4.1 Definitions and Overall Learning Process . . . 39
3.4.2 Eigenmemory . . . 40
3.4.3 Finding MHM Patterns . . . 43
3.5 Evaluation . . . 44
3.5.1 Prototype Implementation . . . 44
3.5.2 Training . . . 45
3.5.3 Anomaly Detection . . . 46
3.5.4 Analysis Time . . . 52
3.5.5 Limitation . . . 52
3.6 Conclusion . . . 53

Chapter 4 Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded Systems . . . 54
4.1 Introduction . . . 54
4.2 Overview . . . 55
4.2.1 Attacks against Sequence-based Approach . . . 56
4.2.2 Adversary Model . . . 57
4.2.3 Assumptions . . . 58
4.3 Anomaly Detection Using Execution Contexts Learned from System Call Distributions . . . 58
4.3.1 Definitions . . . 58
4.3.2 Learning Single Execution Context . . . 59
4.3.3 Learning Multiple Execution Contexts . . . 62
4.3.4 Reduced SCFD . . . 64
4.4 Evaluation Framework . . . 65
4.4.1 Target Application . . . 65
4.4.2 System Implementation . . . 66
4.4.3 Attack Scenarios . . . 67
4.5 Evaluation Results . . . 67
4.5.1 Sequence-based Security Analysis . . . 68
4.5.2 SCFD Training . . . 72
4.5.3 Accuracy . . . 75
4.5.4 Time Complexity . . . 78
4.5.5 Limitations and Discussion . . . 78
4.6 Conclusion . . . 79

Chapter 5 The DragonBeam Framework: Hardware-Protected Security Modules for In-Place Intrusion Detection . . . 80
5.1 Introduction . . . 80
5.1.1 Threat Model and Assumptions . . . 82
5.2 Overview . . . 82
5.2.1 High-level Architecture . . . 82
5.2.2 Sample Use Cases . . . 85
5.2.3 Requirements and Challenges . . . 85
5.3 Detailed Architecture . . . 85
5.3.1 DragonBeam Framework Operations . . . 85
5.3.2 SKM Registration . . . 86
5.3.3 Secure Memory and Access Control . . . 88
5.3.4 Heartbeat and SKM Integrity Check . . . 90
5.3.5 Secure Memory for SecMan and Secure Stack . . . 90
5.4 Security Guarantees of the DragonBeam Framework . . . 91
5.5 Implementation . . . 92
5.5.1 System Configuration . . . 92

5.5.2 Secure Memory Implementation 93
5.5.3 Software Configuration 94
5.6 Evaluation 94
5.6.1 Implementation of Detection Mechanisms 94
5.6.2 Performance Evaluation 96
5.6.3 Hardware Costs 98
5.6.4 Extension to Multiple Monitored Cores 98
5.6.5 Limitations 100
5.7 Conclusion 100

Chapter 6 VirtualDrone: Virtual Sensing, Actuation, and Communication for Attack-Resilient Unmanned Aerial Systems 101
6.1 Introduction 101
6.2 VirtualDrone Framework 103
6.2.1 High-level Framework 103
6.2.2 Assumptions and Adversary Model 105
6.2.3 Virtual Sensing, Actuation, and Communication 106
6.2.4 Security and Safety Monitoring 110
6.3 Implementation 111
6.3.1 Quadcopter Control 111
6.3.2 System Implementation 112
6.3.3 Autopilot 112
6.3.4 Virtualization 113
6.4 Experiments 115
6.4.1 Case Study 115
6.4.2 Discussion 125
6.5 Conclusion 127

Chapter 7 Conclusion and Future Work 128

References 130

Chapter 1

Introduction

Cyber-Physical Systems (CPS) such as advanced avionics and automotive systems as well as industrial automation systems have traditionally been considered to be invulnerable against security breaches. This was particularly the case since, in general, such systems are physically isolated from the external world and also used specialized protocols. However, increased computational power and connectivity in modern CPS platforms inevitably introduce more security vulnerabilities. Accordingly, threats to such systems are growing, both in number as well as sophistication, as demonstrated by recent attacks [145, 91, 131, 142, 88, 29, 31, 30]. Attackers can not only steal critical information but also destabilize such systems. A successful attack on such safety-critical systems can have disastrous effects, from loss of human life to damages to the environment as well as critical infrastructures. Despite tremendous advances in security technologies for general-purpose systems, these systems still rely on rather primitive, outdated countermeasures. Hence, there is a need to develop effective mechanisms that foil attacks on such systems.

CPS poses unique security challenges, as such systems are required to meet stringent requirements in design and implementation as well as strong safety requirements. Furthermore, the limited resources in these systems, namely, computational power, storage, energy, etc., prevent security mechanisms that have primarily targeted on general-purpose systems from being effective. On the other hand, CPS provides defenders with a unique opportunity to take advantage of the design and implementation constraints and the tight coupling of cyber and physical components to deter attackers. This dissertation investigates the *software and hardware architectures* as well as *analytic methods* to enhance the security and dependability of CPS, utilizing the inherent characteristics of CPS.

1.1 Overview of Research

In this section, we overview the research goal and challenges as well as the solutions proposed in this dissertation.

1.1.1 Research Goal and Challenges

It is infeasible to completely secure modern CPS since they have many entry points that are vulnerable to potential attacks and also security attack mechanisms keep evolving. Hence, instead of attempting to prevent every possible

security breach, this research focuses on (a) *detecting attacks as soon as they happen* and, more importantly, (b) *ensuring that the underlying physical system remains safe*. Such security mechanisms work in CPS due to the *regularity* of their behavior – designers ensure that the systems have narrow operational ranges and fixed modes of execution for safety guarantees. Hence, the parameters for legitimate behavior are *limited by design*. This behavioral predictability can enhance the security of such systems by enforcing a strong invariant on their execution behavior. This is because any form of unwanted malicious activity consumes finite resource such as time, memory, I/O, etc., to carry out, and thus it would inevitably alter the run-time behavioral signature from expected baseline behavior.

The behavioral properties should be not only fairly deterministic in normal cases, but also sensitive to abnormal deviations during the execution of the application and the system under monitoring. Depending on the types of the target application and system, the most effective behavior signal and thus the corresponding detection method can vary. The challenge in profiling the normal behaviors lies in the fact that mundane reasons such as system effects (e.g., cache pollution, task preemption, etc.) can also result in *variations of execution behavior*.

Another key challenge towards the realization of this behavior monitoring is *how to monitor*, since if the monitoring component itself is compromised, whatever we monitor is no longer trustworthy and informative. Thus, a secure separation of the monitoring entity from the monitored entity is needed while still guaranteeing a high observability of the monitor, i.e., the ability to instrument the behavior of target application or system at a certain level of granularity.

A successful security attack on CPS, especially safety-critical systems, can lead to failure or malfunction which may result in a safety hazard. Hence, one of the key requirements for secure and dependable CPS is the *attack-resiliency*. That is, the system should be able to maintain safety, for example, even in the presence of zero-day attacks or in the event that the root privileges are taken over by the attacker.

Hence, the goal and key challenges of this research are:

- **Behavior-based Intrusion Detection:** Identifying various types of behavioral properties that can accurately differentiate abnormal behaviors from normal ones and validate the cause of any variations, by utilizing the intrinsic properties of CPS.

- **Trusted Monitoring:** Developing system design principles that can achieve high observability on the system behavior while guaranteeing the integrity and availability of the monitoring and detection mechanisms.

- **Attack-resiliency:** Providing robust recovery mechanisms that can rebound from security attacks and keep the system controllable with minimal disruption.

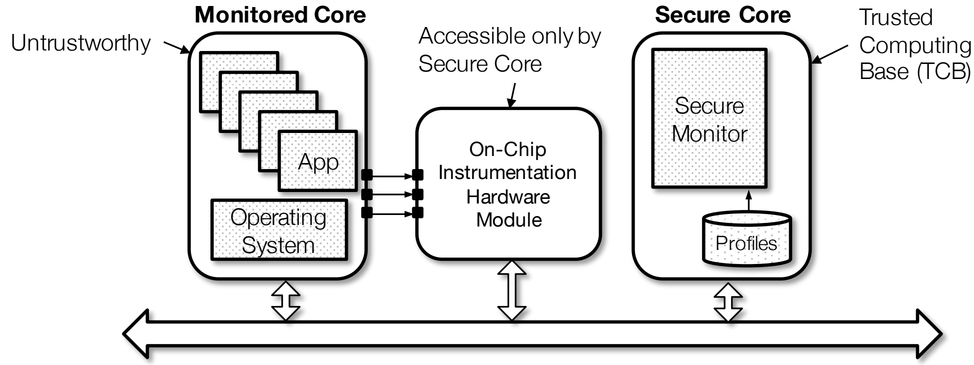


Figure 1.1: Overview of the SecureCore Architecture.

1.1.2 Summary of Solutions

This dissertation presents cross-layer approaches that employ (a) light-weight yet accurate learning and monitoring of the deterministic behavioral properties of CPS to look for anomalous behavior in spite of normal variations of execution behaviors and (b) architectural and operating-system supports to facilitate behavior monitoring, increase trust, and provide attack-resiliency.

Specifically, we propose an array of analytic techniques that take advantage of the behavioral predictability of CPS and yet can be realized with their constraints. These techniques include *statistical learning-based methods* to build models of timing behavior (Chapter 2), memory access patterns (Chapter 3), and operating system-level resource usage (Chapter 4). The major strength of such learning mechanisms is that any variability due to changes in inputs, code complexity, system effects, etc. can be captured in the resulting execution profile. With such mechanisms, attackers would not only need to know the profile of legitimate behavior but would also need to be able to implement an attack without disturbing it. This significantly raises the bar against would-be attackers.

We also present an architectural solution called *SecureCore* architecture, shown in Figure 1.1, a multicore-based architectural supports for run-time monitoring by protected monitoring components (i.e., the trusted computing base, or TCB). SecureCore takes advantage of the redundancy in multicore processors and includes the installation of an on-chip hardware module residing between the cores for a high-resolution but non-intrusive instrumentation. Using the on-chip monitoring module, the trusted entity, i.e., secure core, continuously monitors the run-time behavior of the untrustworthy entities, i.e., monitored core(s). The SecureCore architecture has the following two distinguishing requirements:

- **One-Way Observability:** The monitoring activity of the secure core must be invisible to the untrustworthy entities, i.e., the monitored cores. The secure core should be able to observe the state of the physical system under control, the processor state of the monitored cores, and the I/O data to and from them.

- **One-Way Controllability:** The secure core must be able to intercept and to control data exchange between the monitored cores and the external world for inspection and integrity. When a malicious activity is detected, the secure core should be able to stop the malicious activity from affecting the system.

Chapter 2 provides further details about the SecureCore Architecture. We also show how the architecture can facilitate the process of monitoring timing and memory behavior of real-time embedded systems in Chapters 2 and 3, respectively. In Chapter 5, we present an extension of the SecureCore architecture for monitoring and detecting high-level activities in general-purpose systems, and an implementation on an FPGA softcore processor.

One of the key design principles of the SecureCore architecture is that it should be able to recover the system from security attacks by taking an evasive or corrective action as soon as an abnormal behavior is detected, and thus guarantee that the physical system does not come to harm. For this, the SecureCore architecture is built upon the concept of the *Simplex* architecture [128]. In Chapter 2, we describe an architectural support for the Simplex mechanism and show how it, coupled with behavior-based intrusion detection, can provide a loss-less control for a real-time control system even when it is compromised. Extending the concept of SecureCore architecture, Chapter 6 presents an attack-resilient *software-based* architecture for unmanned aerial systems (UAS) that takes advantage of modern embedded computing technologies including multicore processor and virtualization. The architecture aims to enable advanced flight applications to run in a *potentially untrustworthy* software environment. Upon detection of a security or safety violation, it overrides the malicious system state to keep the UAS operational. We present a prototype quadcopter that runs open-source software stack on a commercial-off-the-shelf embedded computing board, and demonstrate how the framework can provide the attack-resiliency in the presence of various types of security attacks.

This dissertation includes some material previously published in peer-reviewed conferences [162, 163, 161, 156, 159]:

- Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Jung-Eun Kim, Lui Sha, "SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems," in Proceedings of the 19th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2013), Apr. 2013.

- Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Lui Sha, "Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior," in Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC 2015), Jun. 2015.

- Man-Ki Yoon, Mihai Christodorescu, Lui Sha, Sibin Mohan, "The DragonBeam Framework: Hardware-Protected Security Modules for In-Place Intrusion Detection," in Proceedings of the 9th ACM International Systems and Storage Conference (SYSTOR 2016), Jun. 2016.

- Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, Lui Sha, "Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System," in Proceedings of the 2nd ACM/IEEE International Conference on Internet-of-Things Design and Implementation (IoTDI 2017), Apr. 2017.
- Man-Ki Yoon, Bo Liu, Naira Hovakimyan, Lui Sha, "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 (ICCPS 2017), Apr. 2017

1.2 Background and Related Work

In this section, we present the background and relevant work related to the research problem and solutions proposed in this dissertation.

1.2.1 Security Attacks to CPS

The W32.Stuxnet worm [145] was able to successfully subvert operator workstations and gain control of Iran's nuclear power plants through sophisticated attacks including the first known PLC rootkits and use of multiple zero-day vulnerabilities. The worm was able to intrude into the control system by first gaining access and then downloading attack code from a remote site. The malware then gradually inflicted damage to the physical plant by substituting infected actuation commands for legitimate ones over a period of time. Despite the employment of several protection and monitoring mechanisms, the control system could not detect the intrusion and the attack until the physical damage to the plant was significant.

Koscher et al. [91] demonstrated a malicious code injection into the telematics units of modern automobiles. They replaced the engine control module by overwriting the firmware of Electronic Control Unit (ECU), while the vehicle is running. The injected malicious code gained access to the inputs of the internal network, Controller Area Network (CAN). Miller and Valasek [31] showed that attackers can take control of modern automobiles remotely through the Internet. They were able to hack into the central entertainment system of Jeep Cherokee through Wi-Fi and also a cellular network, and to reach the car's CAN bus. Once they had full access to the car's internal system, they were able to take control of physical components such as locks, transmission, brake, etc.

Current unmanned vehicle systems are also vulnerable to cyber-threats as demonstrated by recent attacks. Maldrone [29] is a software virus that can compromise drones based on ARM Linux systems. It was demonstrated that the malware can open a backdoor in the Parrot AR Drones software, infect on-board software and take over the control. Pleban et al. [122] presented analysis details on the insecure WiFi network and OS user management of the Parrot

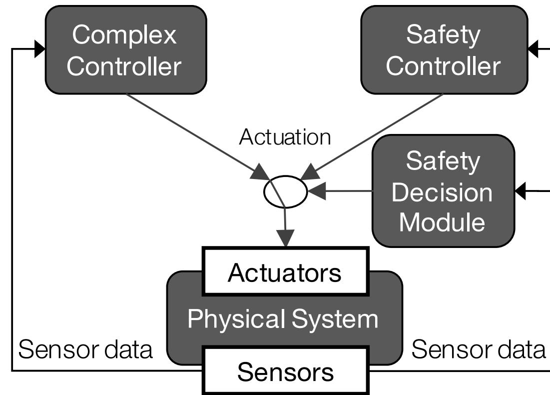


Figure 1.2: Simplex architecture for high-assurance control.

AR Drones. Also, researchers were able to demonstrate that they can hijack DJI consumer drones by emulating fake GPS signals using low-cost software defined radio tools [33]. It is also possible to inject MAVLink message into a radio link by modifying the radio firmware, and hijack a flying drone [30]. Javaid et al. [80] 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 [27] 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. Teso [142] was able to break into the on-board computer system and manipulate the steering of an airplane in the autopilot mode, using Aircraft Communications Addressing and Reporting System (ACARS). It is also demonstrated that the airplane could even be led to a collision.

1.2.2 Simplex Architecture

The Simplex architecture [128, 148, 83], shown in Figure 1.2, 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 and formally verifiable, 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 plant is actuated by commands from 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 maintain loss-less actuation of the physical plant and thus guarantee robust control of the system.

1.2.3 Behavior-based Intrusion Detection

Instead of attempting to prevent every possible security breaches, monitoring the behavior of systems and applications has drawn a great deal of attention, especially in general-purpose systems, due to its ability to detect zero-day attacks [56]. Behavior-based intrusion detection systems (IDS) rely on the *reference* behavior model that is profiled when the target system was in a known good state (i.e., no security violations). The run-time behavior monitored by a specific monitoring/detection technique is compared against the normal profile, and any significant deviation is considered to be abnormal/suspicious. To obtain the normal behavior profile, machine learning techniques are often used. Because the behavior-based IDS treat the *unknowns* abnormal, they can detect zero-day attacks well. However, for the same reason, an inaccurate model can cause lots of false positives.

Behavior-based IDS have been used in various contexts. The use of system calls has been extensively studied in behavior-based anomaly detection for general-purpose systems since many malicious activities often use system calls to execute privileged operations on system resources. Forrest et al. [63] opened up the direction of system call monitoring. They build a database of look-ahead pairs of system calls; for each system call type, what is the next i^{th} system call for $i = 1, 2$, up to N. Then, given a longer sequence of length $L > N$, the percentage of mismatches is used as the metric to determine abnormality. Hofmeyr et al. [76] extends the method by profiling unique sequences of *fixed* length N, called *N-gram*, to reduce the database size. The legitimacy test for a given sequence of length N is carried out by calculating the smallest Hamming distance between it and all the sequences in the database. This fundamental model evolved in various aspects in follow-up work. For example, the N-gram model requires a prior assumption on suitable N because it affects the accuracy as well as the database size. Marceau [100] proposes a finite state machine (FSM) based prediction model to relax these requirements; it can predict the next system call given a sequence consisting of the last N system calls, where N varies with the sequence. Eskin et al. [61] further improves by modeling legitimate sequences as a mixture of prediction trees, in which a *wild-card* is employed to represent sequences in a compact and flexible way. Other prediction models such as Hidden Markov Model (HMM) [150], Markov chains [81], and variable-order Markov chain [140] have also been explored. Chandola et al. [47] provides an extensive survey on various anomaly detection techniques for discrete sequences, considering among different applications, sequences of system calls or user commands.

Burguera et al. [46] find malicious apps using the system call counts of Android applications (traced by a software tool called *strace*). Using a crowdsourcing, the approach collects the system call counts of a particular application from multiple users and applies k-means (with Euclidean distance metric) to divide them into *two* clusters. A smaller cluster is considered to be malicious based on the assumption that benign apps are the majority. There has also been work on system call arguments monitoring. Mutz et al. [104] introduce several techniques to test anomalies in argument lengths, character distribution, argument grammar, etc. Maggi et al. [97] use a clustering algorithm to group

system call invocations that have similar arguments.

Network activity is another type of behavior signal used to detect abnormality. Sinclair et al. [133] used decision trees along with genetic algorithms to generate rules for differentiating malicious traffic from normal network activities. Similar work used robust support vector machines (RSVMs) to detect network activity anomalies [77]. Barford et al. [40] detect abnormal traffic in terms of volume that deviates from the expected baseline traffic by using wavelet analysis. Gu et al. [68] proposed a method to monitor packet class distributions (packet types and port numbers). Yoon and Ciocarlie [157] considered a problem of detecting malicious traffic on Industrial Control Systems (ICS) network by a probabilistic sequence model of legitimate command/data traffic. Sommer et al. [135] provides a good summary of network security using machine learning techniques. There also exists work that looks at function call sequences [114] and unexpected changes to system files [116].

Bigham et al. [43] analyzed n-grams of power readings in an electricity network to detect anomalous events. They also proposed a technique that learns relationships between different power readings to model the normal behavior of the system. Düssel et al. [58] evaluated different combinations of feature extraction and similarity measurement techniques for SCADA (Supervisory Control and Data Acquisition) network payload. These techniques were used to build the center of mass of the normal data, performing n-gram analysis on payloads. Abnormalities were determined by computing the distance from the center. Cheung et al. [50] demonstrated the regularity in communications among SCADA network devices and developed a communication-pattern-based detector that triggers an alert when the availability of (Modbus) server or service is changed due to, for example, a denial-of-service attack. Valdes et al. [146] proposed a flow-based anomaly detection approach, which keeps a library of flows and, using simple statistics, such as mean and variance, detects flows that are unexpected or exhibit significant change in parameters such as packet inter-arrival time, volume, etc. Hadziosmanovic et al. [70] analyzed network- and host-based data traces from a real-world industrial control system network. The traces included communication topology and patterns, message type and content, and also host-level information such as system log and memory traces. The authors evaluated each of these data-trace types in terms of various criteria including threat scope, approach validation, analysis granularity and so on. Good summaries of SCADA-specific intrusion/anomaly detection are provided by Zhu and Sastry [166] and Garitano et al. [66].

In the context of real-time systems, timing information has been used as a behavior signal. The Secure System Simplex Architecture (S3A) proposed by Mohan et al. [103] uses the execution time and the period of real-time control application as a side-channel monitored by a trusted hardware. An earlier work was proposed by Zimmer et al. [168], in which the absolute worst-case execution time (WCET) was used as a security invariant. Zadeh et al. [164] proposed a signal processing technique that analyzes event traces (including time, processor specific identifier, system call type) obtained from periodic execution of embedded system applications. Salem et al. [124] modeled embedded system

behaviors using inter-arrival curves of recurrent system operations in a discrete event trace.

1.2.4 Hardware-based Security Measures

Hardware-based security measures can improve the overall security of system by cutting off potential vulnerabilities in software modules. ARM TrustZone [153] 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. [39] 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. Santos et al. [125] use the TrustZone to build a trusted language runtime (TLR) for running secure applications on mobile devices. Zhou et al. [165] use ARM processors memory domain support for software fault isolation. They isolate untrusted modules in sandboxes by assigning different memory domain IDs to different sandboxes and hence block out-of-domain memory access by controlling control access registers.

There exists work in which a multicore processor (or a coprocessor) is employed as security measure in different aspects. Shi et al. [132] proposed INDRA, an Integrated Framework for Dependable and Revivable Architecture, in which logs of application executions on monitored cores are verified by a monitoring core through buffering of logs on a special on-chip memory. While this is similar to the work presented in this dissertation, the difficulties arise due to the real-time nature of the systems we consider. Also, the security measure in their work is functional behavior such as function calls and return (e.g., the monitoring core verifies if each function always returns to the right address). Similar work can be found in [48] by Chen et al. The work also employs a logging hardware that captures the execution information, e.g., program counter, input and output operands and memory access addresses of any instruction that the monitored application executes. The captured traces are delivered through a cache to another core for inspection for the detection of memory leaks and access to unallocated memory. The work was extended where a hardware accelerator was proposed to reduce high overheads in instruction-grain monitoring [49]. There have also been coprocessor-based approaches. Kannan et al. [87] addressed the high overheads in the multicore-based Dynamic Information Flow Tracking (DIFT) by proposing the DIFT Co-processor, in which application instructions and memory access addresses, etc. are checked with a pre-defined security policy. A similar approach was taken by Deng et al. [55] where reconfigurable logic attached to the main CPU checks for software error as well as DIFT from execution traces. Mohan et al. [103] use an FPGA-based trusted hardware component to monitor the real-time execution behavior of a real-time control application running on an untrustworthy main system.

The common theme in these systems is that they dedicate hardware resources to specific security tasks. These hardware-based approaches provide better security than virtualizations can do because the latter relies on the security and correctness of the virtual machine monitor which is susceptible to software attacks [154, 45, 115].

1.2.5 Virtual Machine Introspection

Advances in virtualization technologies have enabled Virtual Machine Introspection (VMI) [65], in which a trusted virtual machine (VM) or the virtual machine monitor (VMM) monitors and analyzes the state of applications or operating system running inside untrusted VMs. By placing the detection mechanism outside of the VMs it monitors (i.e., the untrusted VMs), VMI overcomes the vulnerability of the traditional host-based IDS. VMI also has advantages in that no hardware modification is needed.

VMIs have been applied to process execution monitoring [86, 138], kernel control-flow integrity check [119], virtual memory and disk monitoring [112], dynamic information flow tracking [75, 155], system call tracing [120], etc. Although the *out-of-box* approach can improve the security of IDS due to the separation, it misses out on a detailed view of the untrusted VMs, and OS/applications running on it, that leads to *semantic gap* problems [82, 57]. Hence, a line of work closes the gap by placing security *hooks* inside the untrusted VMs. Lares [113] closes the gap by placing hooks inside kernel APIs from which relevant information are passed to the security VM through the VMM. They modified the VMM to protect the hooks by making their memory regions read-only. The switching between the VMs, however, cause a significant performance overhead. Sharif et al. [130] proposed Secure In-VM Monitoring (SIM) in which the hook handler runs in the same VM as if it resides in a separate VM. The VMM protects the in-VM handlers by providing a separated address space to them; this is done by manipulating the shadow page tables by the VMM, which is also used by Wang et al. in [149] for in-VM hook memory protection.

As briefly mentioned above, virtualization is susceptible to security attacks because of the increased complexity in the virtualization software. For example, Pék et al. [115] demonstrated a number of device-related attacks that make it possible for an attacker to control or reconfigure devices attached to other virtual machines, or to modify unauthorized memory regions. They were also able to demonstrate interrupt attacks that can launch Denial-of-Service attacks against other VMs or the VMM, and even can execute arbitrary code in them. There have also been techniques to improve the security of virtualization. SecVisor [127] and TrustVisor [101] are special-purpose hypervisors (i.e., VMMs) that reduces the attack surfaces by minimizing the amount of code within the VMMs. Szefer et al. [141] eliminates VMM attack surface by removing the need for VMs to constantly interact with the VMM. CertiKOS [67], a certified operating system that can serve as a VMM, reduces vulnerabilities in VMMs by incorporating formal logic and verification techniques.

Chapter 2

SecureCore: A Multicore-based Intrusion Detection Architecture for Real-Time Embedded Systems

Security attacks are becoming more common in safety-critical real-time embedded systems, an area that was considered to be invulnerable against software security breaches in the past. A failure to protect such systems from malicious entities could result in significant harm to both humans as well as the environment. Many recent successful security attacks on such systems call for a rethink of the security of safety-critical embedded systems. This chapter presents the *SecureCore* architecture that, coupled with novel monitoring techniques, is able to improve the security of real-time embedded systems. We aim to detect malicious activities by analyzing and observing the inherent properties of the such systems using statistical analyses of their *temporal behavior profiles*. With careful analysis based on these profiles, we are able to detect malicious code execution as soon as it happens and also ensure that the physical system remains safe.

2.1 Introduction

Multicore processors are finding wide use in a variety of domains and embedded systems are no exception. The increase in performance, reduction in power consumption, and reduced sizes of systems using multicore processors (a single board instead of multiple boards) makes them very attractive for use in safety-critical embedded systems. A problem with the use of multicore processors in such systems is that of *shared resources* – components such as caches, buses, memory, etc., are shared across the multiple cores and could result in security vulnerabilities [109]. For example, malicious entities could snoop on privileged information used/generated by critical code running on alternate cores, high-priority tasks could be prevented from executing by a denial-of-service attack on the shared resources (e.g., keeping the bus occupied by large DMA transfers could prevent the high priority task from obtaining the memory reads it requested), etc. Hence, there is a need for a comprehensive solution where multicore processors could be used in safety-critical systems in a *safe* and *secure* manner. In fact, the very nature of such processors – the parallel cores and the convenience they provide – could be used to improve the overall security of the system.

In this chapter, we present *SecureCore*, a secure and reliable multicore architecture solution to tackle security vulnerabilities in real-time embedded systems. We specifically pursue an approach to entrusting certain CPU cores in

a multicore processor with the role of monitoring and intrusion detection. The use of multicore processors has inherent advantages over off-chip security devices: (i) a CPU core is able to more closely monitor the execution behavior of software running on the other (potentially) unsecured core(s); (ii) the mechanisms cannot be tampered with easily or reverse-engineered. Section 2.2 provides further details about the SecureCore Architecture.

We also introduce novel techniques to observe inherent properties of the real-time code executing on the monitored core in Section 2.3 – properties such as execution time for instance. These properties tend to be fairly deterministic in such real-time systems and hence can be used as a way of detecting anomalous behavior that is indicative of malicious activity. These observations, in conjunction with the capabilities of the SecureCore architecture, significantly increase the security of the overall system by enhancing the abilities to detect intrusions. The key idea behind the proposed architecture and intrusion detection mechanism is that since real-time embedded control applications generally have regular *timing behavior*, an attack would inevitably alter its run-time timing signature from expected values [103].

Our architecture proposes a design so that a trusted entity, a *secure core*, can continuously monitor the run-time execution behavior of a real-time control application on an untrustworthy entity (henceforth referred to as the *monitored core*), in a *non-intrusive manner*. In case malicious behavior is detected, a reliable backup control application residing on the secure core takes control away from the infected core in order to guarantee stability and loss-less control for a physical system [128]. Since there will be some inherent variability in these properties – for instance due to changes in inputs, interference on shared resources, etc., we use a statistical learning-based mechanism for profiling the *correct* execution behavior of a sanitized system.

In summary, this work implements the following: (a) a novel architecture based on a multicore platform that provides security mechanisms for use in embedded real-time systems; (b) execution time-based intrusion detection mechanisms using a statistical learning; and (c) Simplex [128] architecture-based reliability. All of these combined provides non-intrusive, invisible monitoring capabilities and reliable, seamless control for safety-critical real-time systems.

2.1.1 Assumptions

In this work, the following assumptions are made without loss of generality:

- We consider a CPU-based real-time control application – i.e., a system consisting of periodic sensing and control tasks.
- We assume the application runs on a single monitored core. The proposed intrusion detection method does not work with multiple monitored entities in the current form.
- We assume that the size of the input set (to the control application under consideration) is small. This can be

justified by the fact that most real-time control applications have a small footprint for input data (velocity, angle, etc.) within fairly narrow ranges.

- We assume that the execution time of the application is not unbounded. For example, the upper bounds for loops is known a priori. However, this assumption is not strictly required. It is sufficient to assume that (almost) all possible loop bounds are profiled.
- Similarly, we assume there is no hidden execution flow path in the application – all paths are present when being profiled.

2.1.2 Motivation

Threat Model

Instead of trying to prevent and/or detect intrusions at every vulnerable component, we intend to monitor and detect intrusions at the *most critical* component: in real-time control systems, the primary concern is the safety of the physical plant under control. Thus, we focus on detecting an intrusion that directly targets the real-time control application. We assume that regular security process was in place to ensure the security during the application design and development phases, i.e., the application is trustworthy initially. The execution timing model is obtained via profiling prior to system deployment. Also, the timing information is obtained by re-profiling the system after any updates and is supplied with the modified application (if any). We assume that the application could be compromised after the profiling stage but the stored timing profile cannot be tampered with during the updating process. We consider malicious code that can be secretly embedded in the application, either by remote attacks or during upgrades. The malicious code activates itself at some point after the system initialization and then gradually tries to change, damage, or even snoop on the physical state of the plant under control. We are not directly concerned with *how* the malicious code gained entry, but more concerned with what happens after that.

Use of Multicore Processor in Real-Time Control Systems

Multicore processors are receiving wide attention from industries due to their ability to support generic and high-end real-time applications that traditional control hardware, e.g., programmable logic controllers (PLC), are unable to provide. This trend is especially strong for instance in automotive industries [5] where CPU-based real-time control applications have a significant presence, e.g., engine control, anti-lock braking systems (ABS), etc. It was shown that automotive control applications are increasingly vulnerable to security attacks as they are more equipped with high-end and complex technologies [91]. Although we do not specifically consider automotive control applications, the use of the mechanisms presented in this chapter will naturally fit into the future development processes of safety-critical

components as the industries are more moving toward employing more multicore-based real-time control systems. Also, the use of one or more cores for improving the security (and overall safety) of such systems is a big plus. Even though some of the resources (cores in this case) are being used up, the increase in security that is provided as a result definitely offsets any losses in performance. Hence, the use of multicore processors in secure real-time embedded systems will be beneficial to the community.

2.2 SecureCore Architecture

In this section, we present the *SecureCore* Architecture, a secure and reliable multicore architecture that aids in the detection of intrusions in embedded real-time systems and guarantees a seamless control to the physical system. We first introduce the overall structure of the architecture and then discuss the design consideration of each component in detail.

There exist several challenges in both hardware as well as software, before these techniques could be implemented in a satisfactory manner. First, a protection mechanism must be provided to the secure core so that it is tamper-resistant (especially from malicious activity on the unsecured/monitored cores). Second, the secure core should be able to closely monitor the state of the other core in real time. However, the monitoring activity should be invisible as far as the observer is concerned – this is mainly so that an attacker should not be able to deceive the intrusion detection mechanisms by means of replay attacks (i.e., replicating previously recorded execution behavior of an application in its correct state). Third, in a multicore environment, an application will inevitably experience a considerable variation in its execution time due to the interference caused from inter-core resource contentions [126, 167, 158]. Thus, the security invariant, i.e., the execution time profile, should be accurate enough so that the intrusion detection method can effectively validate the cause of any such variations. Similarly, the method should be able to take into account execution time variation caused by legitimate application contexts such as differences in input sets and execution flow. Finally, the secure core should be able to guarantee loss-less control to the physical system that it manages even if the main, monitored, or control application is compromised. How we solved these problems is elaborated in the following sections.

2.2.1 High-Level Architecture

Figure 2.1 shows the high-level structure of the SecureCore architecture. The system is composed of four major components – (a) the secure core, (b) the monitored core, (c) the on-chip Timing Trace Module (TTM) and (d) the hypervisor. The system is built upon the concept of the Simplex architecture [128]: a safety controller and a decision module rest on the secure core while a *complex controller* (essentially the controller that manages the physical system)

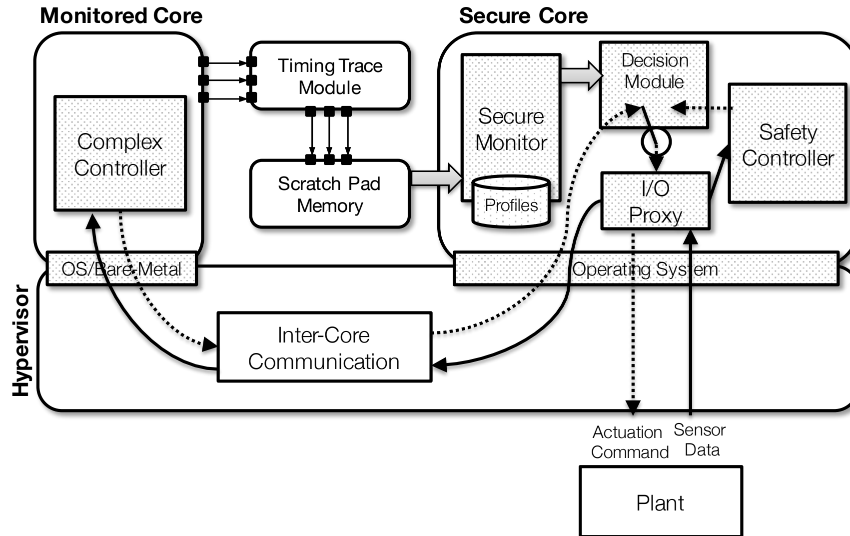


Figure 2.1: SecureCore Architecture.

runs on the monitored core. Sensor data from the physical plant is fed to the both controllers, each of which computes actuation commands using their own internal control logic. The decision module on the secure core then forwards the appropriate command to the plant depending on a pre-computed *safety envelope* for the physical system. In normal circumstances, the plant is actuated by commands from the complex controller. However, when an abnormal operation of the complex controller is detected (e.g., due to unreliable/erroneous logic and faults), control is transferred to the safety controller in order to maintain loss-less actuation of the physical plant. With this mechanism, the stability of the control actions can be guaranteed by the decision module and the safety controller (that can be formally verified), provided however that all the entities are trustworthy. It is possible for the decision module or safety controller to be compromised by a security attack. Furthermore, the complex controller may deceive the decision module by providing a legitimate actuation value while, for example, collecting critical system information that could be exploited during a future attack. Thus, it is important to ensure a high security level for the system. In the following sections, we describe how the security as well as the reliability of this basic Simplex mechanism can be enhanced by use of the SecureCore architecture.

2.2.2 Design Considerations

Our solution includes a hypervisor that provides a *virtualization* of hardware resources on our proposed SecureCore architecture through *partitioning* and *consolidation*. In order to protect the secure core from malicious alteration by a compromised complex controller, the hypervisor provides a clean separation of memory spaces by programming the memory management unit (MMU). Also, the hypervisor itself runs in its own protected memory space. Thus, any attempts at memory access across the partitions is blocked by the hypervisor.

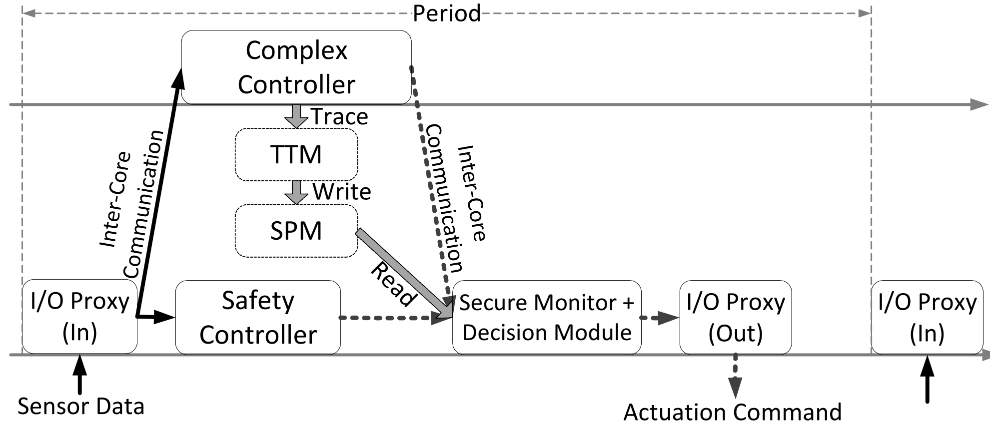


Figure 2.2: Execution flow of the SecureCore components.

With the help of this memory protection, we design an I/O channel between the processor and the plant. The channel is managed by an *I/O proxy* process that runs on the secure core. The I/O proxy manages all I/O to and from the physical plant. This is to prevent I/O data obfuscation that could be caused by malicious code on the monitored core. Furthermore, if the I/O channel device is directly accessible by both cores then a compromised application on the monitored core may attack the secure core indirectly by, for example, causing a denial-of-service attack on the I/O channel – this will prevent the safety controller from taking over from the complex controller. This I/O device consolidation capability is also provided by the hypervisor through the I/O MMU. The system is configured such that the device cannot be seen from the monitored core.

Since the memory space is partitioned and the I/O device is consolidated to the secure core, data to and from the monitored core is relayed via the *inter-core communication* channel on the hypervisor level. Transferring data through a shared memory region is strictly prohibited because of a potential vulnerability [109]. As shown in Figure 2.2, the I/O proxy first retrieves sensor data from the plant and then transfers it to the two controllers. For the complex controller, the I/O proxy places the data on a dedicated channel between the memory space of the secure core and that of the hypervisor. The data is then copied to the buffer at the monitored core's side. The complex controller retrieves the data by either polling or an interrupt-driven method. For the opposite direction, however, i.e., if the complex controller wishes to send out actuation commands and when the decision module wishes to retrieve such a command, the decision module polls the buffer on the inter-core communication channel. The decision module also sets a watch-dog timer for this process. When the timer expires and the decision module has still not received data from the complex controller, the safety controller takes over the control. This polling-based data passing is to prevent the secure core from being unboundedly interrupted by a compromised complex controller – a vulnerability that can be exploited using an interrupt-driven method.

The main component that enforces the security invariant in the architecture is the *secure monitor* – a process that

continuously monitors the execution behavior of the complex controller. The secure monitor works in conjunction with an on-chip hardware unit called the *Timing Trace Module* (TTM). The details of secure monitor and TTM are discussed in Section 2.2.3. The key role of the monitor is to detect if the run-time execution time signature has deviated from what is expected/has been profiled. If any unexpected deviations are observed then the secure monitor informs the decision module and the control is immediately switched over to the safety controller in the secure core. At the same time, the hypervisor is told to reset the monitored core and then reload a clean copy of the complex controller binary from a secure memory region. Once the reset and reload is complete, the monitored core could, potentially, resume operation and take control back. Of course, this could depend on the policy for recovery that is implemented on the actual system. It could also happen that the monitored core is completely shut down and not restarted until an engineer analyzes the issue. This will prevent smart attackers from triggering constant back and forth switches between the complex and simple controllers – events, that could themselves, cause harm to the physical system if timed correctly.

As described above, the architecture relies heavily on the hypervisor. Thus, the entire secure mechanism can collapse if the hypervisor itself is compromised in the first place. Hence, it is assumed in this chapter that the hypervisor forms part of the trusted base; no malicious code is embedded in it. We note that while a hardware-enforced memory protection mechanism would further enhance the security of the hypervisor [153], we do not address this issue in this work.

One may argue why the monitored core does not receive the same protection as the secure core. The secure core needs to exist for two reasons: (i) fault tolerance when the main controller fails either due to a fault or a security violation and (ii) security; the monitored core might need to have software updates/interact with external sensors, perform I/O, etc., while the data/code in the secure core will not often change. Hence, it makes sense to *harden* the secure core and not always the monitored core. Of course, even if the monitored core is hardened, it can still be susceptible to smart attackers. Another issue is that the way the secure core is hardened is by providing it with private memory and other hardware mechanisms. Doing this for all other cores would be prohibitive, especially when we want to increase the observation to multiple cores.

2.2.3 Timing Trace Module (TTM)

The Timing Trace Module (TTM) is a special on-chip hardware unit that traces the run-time timing information of the monitored core. The module is located between the monitored and secure cores and directly attached to the former as seen in Figure 2.3. When a certain event is triggered (the execution of a special instruction; explained shortly), a part of the processor state is read by the TTM. The processors state includes the values of the *timestamp counter*, the *program counter* (PC) and the *process ID* (PID) of the current task. The trace information is then written to the scratch

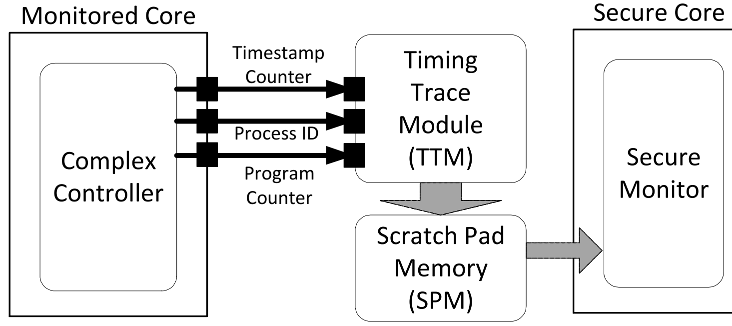


Figure 2.3: Timing Trace Module.

pad memory (SPM) that can be seen/accessed only by the secure core. The SPM is mapped to a range of the secure core's address space. A sequence of traces is collected during one single run of the complex controller (Figure 2.2). The secure monitor verifies the legitimacy of the execution profile obtained from the trace by comparing it with one that has been collected during implementation time when the system was in a known *good state*.

We now present how TTM traces the required information from a running application. A trace operation is carried out by executing a special trace instruction in the monitored application, as described in Figure 2.4 (a). The special instruction also has a mode when it can register the PID of the monitored application with the TTM. Once a PID is registered, only a process that matches the PID can execute other trace instructions; this is to prevent traces from being forged by another process that might be compromised. The PID value is written at the top of the SPM and the PC value at that point is registered as the Base Address (BA) as shown in Figure 2.4 (b). When the trace instruction is executed while the tracing is enabled, the timestamp and the instruction address at that point execution are written at the address specified by the value of $\mathrm{Addr}^{\mathrm{Head}}$. Here, the address being written is a *relative* address from BA, i.e., $\mathrm{PC_i} - \mathrm{BA}$, that can contain positive or negative values. The reason for storing a relative address is to capture the *exact* signature of each trace, since the real addresses can change between executions – two sequences of traces may not match although they are produced at the identical places.[1]

Note that the TTM is used at *two* different points in the whole process: (i) during the development/testing phase, it is used to collect profiling information about the application tasks, i.e., the real-time execution time profile described later in this chapter and (ii) when the system is actually deployed in the field, the TTM is a conduit for flow of the monitoring information from the monitored core to the secure core. Meanwhile, the trace instructions are manually inserted into the code.

The SPM is a circular buffer of traces. When a single run of the complex controller completes, the secure monitor consumes a sequence of traces specified by $\mathrm{Addr}^{\mathrm{Head}}$ and $\mathrm{Addr}^{\mathrm{Tail}}$. While it is possible for SPM buffer to overflow during execution, we note that only a small number of traces would be enough in a real-world control application due

[1] We assume that no dynamically loaded libraries exist in the system and even if they do, we do not trace them.

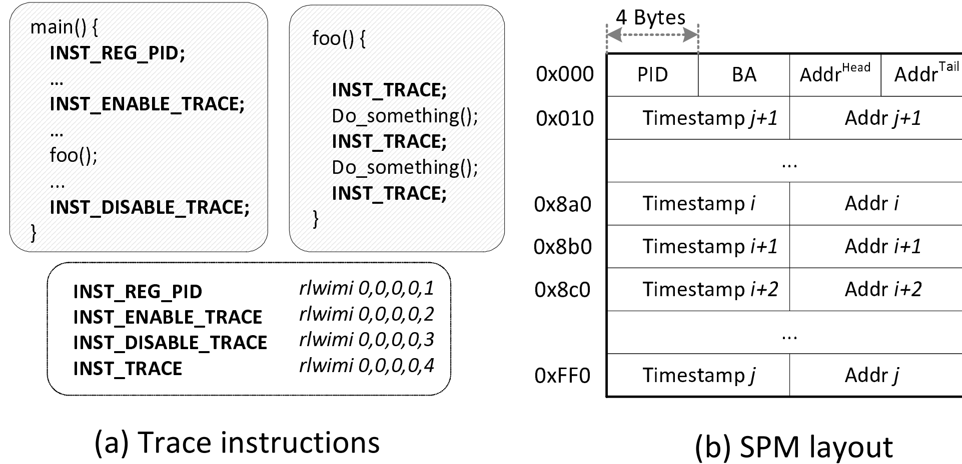


Figure 2.4: Trace instructions and the memory layout of SPM.

to a short span of the execution times. Also, we chose to use an SPM instead of shared memory as the buffer for the traces because an SPM has a lower access latency. The shared memory communication can also open up potential security breaches [109].

2.3 Gaussian Kernel Density Estimation for Execution Time-Based Intrusion Detection

The intrusion detection method presented in this chapter utilizes the deterministic timing properties of real-time control applications. Since any form of unwanted malicious activity consume finite time to execute, a deviation from expected regularity would likely point towards an intrusion. However, as explained in Section 2.1, the execution time of an application can also include variations due to other, more mundane, reasons such as system effects. On a multicore processor, the sharing of hardware resources such as caches, buses, memory, etc., can result in variability in the execution times. Also, an application's own context such as different input sets and execution flows can cause deviations in timing. The main difficulty in profiling and estimating execution time comes from the fact that it is often non-parametric; e.g., monitoring only the mean, minimum, or maximum values is often not accurate enough for our purposes. Thus, in this section, we present a *statistical learning-based execution time profile and intrusion detection method* that can effectively validate the causes of any observed perturbations in execution time and account for their causes.

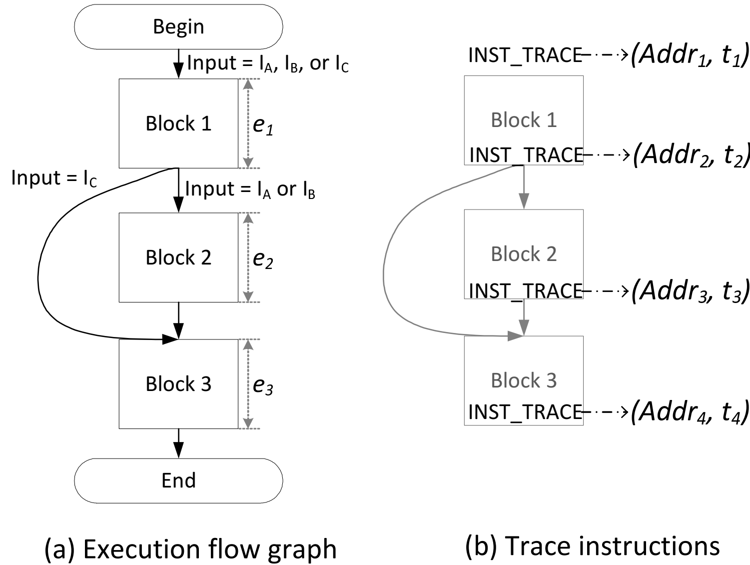


Figure 2.5: Trace instructions inserted to an example application.

2.3.1 Overview

Let us first consider a simple example application consisting of three blocks of code (Figure 2.5 (a)).[2] The blocks are sequentially executed but depending on the input value Block 2 may be skipped. Here, we do not assume a specific form of inputs – the input can be a single value, a range, or even multiple ranges of values. However, it should be assumed that the execution flow does not show deviations when presented with the same input.

The execution time profiling method (explained in Section 2.3.2) profiles the execution times of each *block* (measured in CPU cycles) and generates an estimation on it. During run-time monitoring, each measured execution time of block i, e_i, is compared with the estimation, $\hat{e}_i$, to check how close it is to the legitimate behavior. The reason we do not profile aggregated execution time is to improve the detection accuracy by narrowing the estimation domain. That is, each variation at every block gets accumulated along the execution path and this would obscure potential malicious code execution inside. For example, an attack code could redirect the execution (say using buffer overflows) during the execution of Block 2 and then return to the right address in a short amount of time. In such cases, the time taken by the extra code may fall within the interval of allowed deviations of aggregated execution time. Moreover, with block level monitoring, each block boundary can be used as a check point – the monitor can detect malicious execution along a path where a block is either skipped or never exited. Thus, an attacker would need to not only keep within fixed paths, but also complete execution in a very short amount of time – both of which significantly raise the bar against would be attackers.

[2] A *Block* could refer to a sequence of instructions of arbitrary size and does not necessarily mean *Super Block* [78].

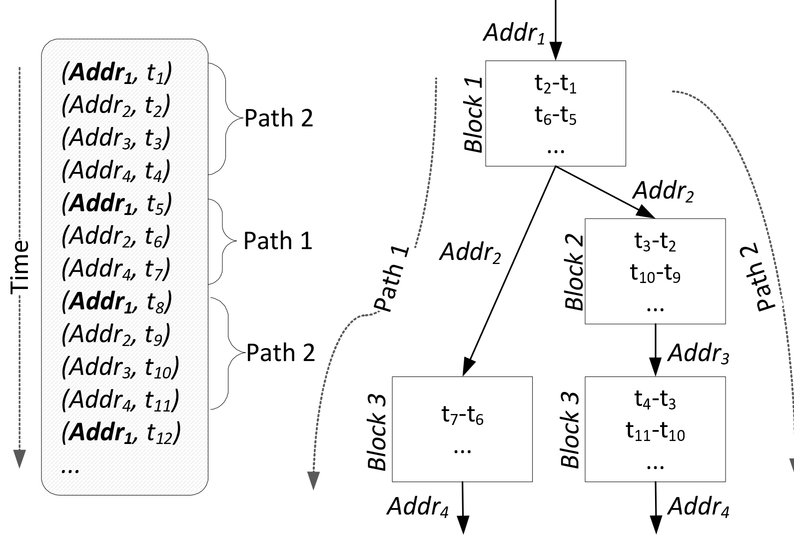


Figure 2.6: Trace tree generated from a sequence of traces.

2.3.2 Trace Tree

We now explain how traces generated by the TTM can be used to profile block execution times. Consider the execution flow graph in Figure 2.5 (a). Suppose we are interested in monitoring blocks between `Begin` and `End`. Then, we add an `INST_TRACE` instruction at the end of each block and at the top of the flow as shown in Figure 2.5 (b).[3] Every time the instruction executes, a pair $(Addr_i, t_i)$ is added as a trace (see Section 2.2.3). This results in a sequence of traces for a single execution of the application – e.g., $(Addr_1, t_1)$, $(Addr_2, t_2)$, $(Addr_4, t_4)$, etc., for one input of I_C. Assuming each run of the application begins at the same entry point, we can construct a *trace tree* from a collection of such sequences as shown in Figure 2.6. In the tree, each edge corresponds to the address (relative from a base address) at which each `INST_TRACE` is executed. Thus, a block in the original execution flow graph can be defined as a pair $(Addr_p, Addr_c)$, where $Addr_p$ is the address of the last trace instruction that is executed before and $Addr_c$ is the address of the instruction that is executed right after the block. Accordingly, each node in the tree is a set of *time differences* between the two addresses that then are the *samples* of the block execution time.

Note, however, that the same block may have different $Addr_p$ values depending on an execution flow, for instance Block 3. Observe that such a block appears in multiple *trace paths*. Here, we define trace path P_i as a sequence of addresses $(Addr_{i,1}, Addr_{i,2}, \ldots, Addr_{i,n})$, where n is the number of blocks along the execution path. Thus, two trace paths, P_i and P_j, are distinguishable if there exists a k such that $Addr_{i,k} \neq Addr_{j,k}$ (e.g., $Addr_{1,3} = Addr_4$ and $Addr_{2,3} = Addr_3$). Thus, the two Block 3's can be distinguished by the trace paths taken. Note that we extracted the trace paths from the tree without prior knowledge of input values. The tree is constructed only from a given collection of trace sequences. A higher accuracy in profiling and monitoring would be achieved by including input information

[3]It should be noted that no `INST_TRACE` instruction must be placed *inside* a recursive function.

when constructing the trace trees.

Now the trace tree gives us the information of how the application needs to behave in order to be considered as *legitimate execution* – i.e., in what order the traces have to be generated. In the next step, we estimate each block's execution time with samples at each node. The obtained profile will strengthen the *invariant* by enforcing what ranges of execution time each block has to fall within. The trace tree will also infer what each block's execution time should be, for individual path. However, one issue remains: a block's execution time can also vary for different inputs even along the same path (e.g., Block 3 at the right subtree in Figure 2.6). In what follows we address the problem of block execution time estimation in the face of varying control flow and inputs.

2.3.3 Profiling Block Execution Time Using Gaussian Kernel Density Estimation

Suppose we are given a set of samples of block execution times from a trace tree node. In this section, we show how to find a good estimation on the samples that can effectively classify the differences between legitimate and malicious execution behaviors. As previously explained, although a real-time control application has regularity in timing, noise (system effects, resource contentions, etc.), control flow variations and even input sets can cause variance in execution times. Thus, instead of trying to obtain accurate (or tight) ranges of execution times we calculate the likelihood of legitimate executions by taking into account the effects of such perturbations. For this purpose, we estimate the *probability density function* (pdf) of execution times, $f(e)$, from a set of samples, $(e^{(1)}, e^{(2)}, \ldots, e^{(m)})$, by using the Kernel Density Estimation (KDE) [111, 85, 51] method. KDE is a non-parametric pdf estimation method that estimates an unknown pdf directly from sample data as follows:

$$\hat{f}_h(e|e^{(1)}, \ldots, e^{(m)}) = \frac{1}{m} \sum_{i=1}^{m} K_h(e - e^{(i)}),$$

where K_h is a *Kernel* function and h is *Bandwidth* (also known as the *smoothing constant*). Hereafter, we simplify $\hat{f}_h(e|e^{(1)}, \ldots, e^{(m)})$ as $\hat{f}_h(e)$.

There exist several kernel functions such as Epanechnikov [60], triangular, uniform, etc. However, in this work, we use the Gaussian kernel, $K_h(x) = \frac{1}{\sqrt{2\pi}h} e^{-x^2/2h^2}$, where $-\infty < x < \infty$.[4] The key idea of the Gaussian KDE is to first draw a scaled Gaussian distribution (parameterized by the bandwidth h) at each sample point along the x-axis (i.e., e-axis) and then to sum up the Gaussian values at each e that results in the probability density estimate at e, i.e., $\hat{f}_h(e)$. Thus the more samples that are observed near e, the higher the density estimate $\hat{f}_h(e)$ will be. Figure 2.7 shows the probability density estimation derived by Gaussian KDE from a set of 6708 samples of an example block (used in the prototype implementation in Section 2.4). As can be seen from the figure the estimated pdf is in a non-regular shape

[4] We do not address the problem of choosing the kernels and the optimal bandwidth. Interested readers can refer to [111, 60, 85].

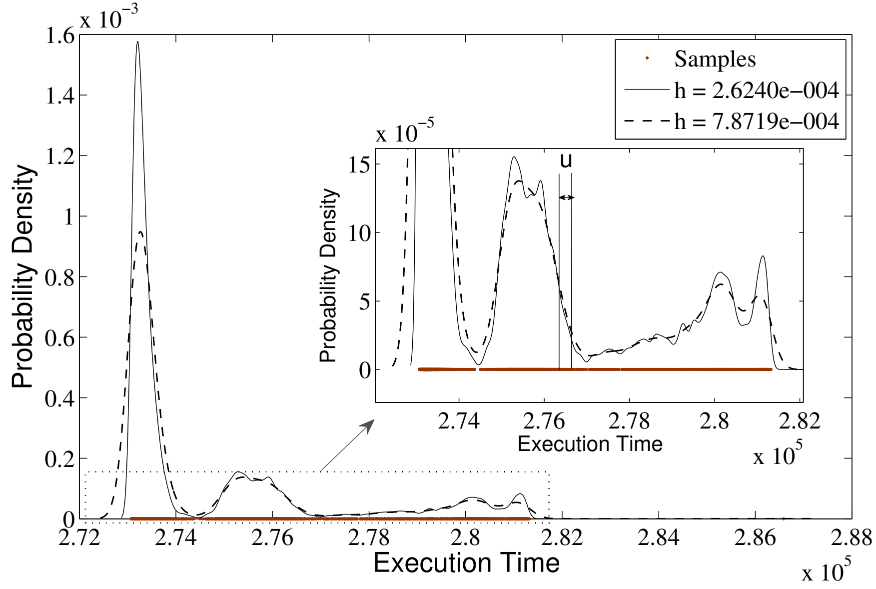


Figure 2.7: Probability density estimation of an example execution block.

compared to what could have been obtained by a parametric distribution such as Gaussian. Also, as the bandwidth becomes wider, the resulting pdf is further smoothed out. Given this pdf, one can expect that a newly observed e^* would highly likely fall within the ranges close to 2.73×10^5, 2.75×10^5 cycles, etc.

2.3.4 Intrusion Detection Using Execution Time Profiles

To deal with the timing variations during the execution of the code, we use the idea of probability density estimations for monitoring and detecting intrusions. We now show how this information is used to detect intrusion at run-time. In what follows we limit ourselves to a single trace node (i.e., a block). The same method is applied to all other nodes that form a part of the code. Suppose we are given the probability density estimation $\hat{f}^k$ of node k.[5] Let $P^k(a \le e \le b)$ be the probability that an arbitrary execution time e is observed between a and b with the given pdf. Note that the probability that e is included within a range $[a, b]$ is

$$P^k(a \le e \le b | e^{(1)}, \cdots, e^{(m)}) = \int_a^b \hat{f}^k(e | e^{(1)}, \cdots, e^{(m)})\, de.$$

Here, the obtained pdf may not be directly usable in the continuous domain, depending on the implementation. Thus, we derive the *discrete probability distributions* (or probability mass functions) instead. Let N be the number of uniformly distributed points on the e-axis that the Gaussian KDE evaluated on. Then, there are $N-1$ bins, each of which is characterized by $[e_{min} + i \cdot u,\ e_{min} + (i+1) \cdot u]$. Simply put: $[b^i_{min}, b^i_{max}]$, for $i = 0, \ldots, N-1$ and

[5] We drop the subscript h from $\hat{f}_h$ to simplify the expression.

$u = (e_{max} - e_{min})/(N-1)$, where e_{max} and e_{min} are the maximum and the minimum values among the observed samples, respectively. In this setting, $P^k(e^*)$, the probability of a specific execution time e^*, is approximated by

$$P^k(b^{i^*}_{min} \le e \le b^{i^*}_{max}) \approx \hat{f}^k(b^{i^*}_{min}) \cdot u,$$

where $i^* = \lfloor \frac{e^* - e_{min}}{u} \rfloor$, i.e., $e^* \in [b^{i^*}_{min}, b^{i^*}_{max}]$; u is the bin width and $\sum_{0 \le i \le N-1} P^k(e \in [b^i_{min}, b^i_{max}]) = 1$.

Note that $P^k(e^*)$ is the probability of being a legitimate execution instance assuming that the estimated pdf is the true distribution. Thus, in order to deal with errors resulting from the estimation we compare $P^k(e^*)$ with a predefined minimum required probability, θ (e.g., $\theta = 0.05$ or $\theta = 0.01$). If $P^k(e^*)$ is below θ we consider that the execution instance to be malicious.[6] Hence,

$$\begin{cases} \text{if } P(e^*) < \theta & \text{malicious,} \\ \text{if } P(e^*) \ge \theta & \text{safe.} \end{cases}$$

The value of θ affects the rate of misclassification. We define a *false positive* as a case where the secure monitor says something is malicious when it is not. Similarly, a *false negative* is defined as a case where the monitor could not detect a real attack. With a higher θ, the rate of false negatives would decrease. However, at the same time, the rate of false positives will also increase. Note that setting θ to 0 implies that any execution is considered to be legitimate.[7]

Lastly, suppose we obtained $\hat{f}^k$ for all nodes in a trace tree. For a given sequence of traces generated during a single execution of the monitored application, the secure monitor traverses the trace tree with the address values as explained in Section 2.3.2. At each node k, the secure monitor calculates $P^k(t_c - t_p)$ where t_p and t_c are the timestamps when two subsequent trace instructions are executed at $Addr_p$ and $Addr_c$. If there exists at least one k such that $P^k(t_c - t_p) < \theta$, the secure monitor considers that execution to be malicious. Since we use Gaussian KDE of execution time variations for detecting intrusions, we shall henceforth refer to this technique as the *Gaussian methods for Intrusion Detection using Timing profiles* (GaIT).

2.4 Implementation

In this section, we present the implementation details for a SecureCore prototype. We first describe the hardware-level implementation and setup and then explain the software components. The latter includes a real-time control application and embedded malicious code.

[6] The proposed model is related to outlier detection algorithms [89]. More specifically, the null hypothesis - e^* is legitimate - is that the sample has at least θ percent of all other points having a distance to the sample less than u, the bin width. One may regard that we reject the null hypothesis when $P^k(e^*) < \theta$.

[7] One may perform a non-parametric hypothesis test such as Wilcoxon Signed-Ranks Test [152] or Anderson-Darling Test [37]. It should be noted, however, that the intrusion detection process needs to be performed for a set of execution time samples instead of a single sample in such

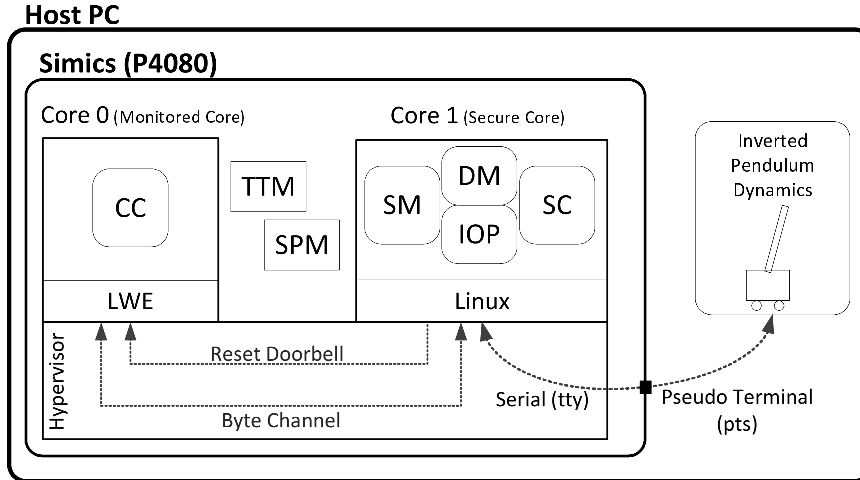


Figure 2.8: SecureCore prototype implemented on Simics P4080 model.

2.4.1 System Implementation

We implemented SecureCore on Simics [98], a full-system simulator that can run a hardware platform including real firmware, device drivers as well as an unmodified OS and hypervisor and also allow processor architecture modifications. Figure 2.8 shows the system implementation overview (see Table 2.1 for the implementation parameters). We used the Freescale QorIQ P4080 Processor [11] platform that has eight e500mc cores [8]. Only two out of the eight cores were enabled – i.e., cores 0 and 1 were used as the monitored and secure cores respectively. The secure core side runs Linux kernel 2.6.34. The monitored core runs on the Freescale Light Weight Executive (LWE) [12]. The choice of LWE is specific to this implementation but we used it for the support of rapid reset and reload of a trusted binary it provides. The LWE could easily be replaced by any commodity or real-time OS, depending on the system requirements.

The hypervisor is configured such that the memory spaces between the cores are cleanly separated and the monitored core is set to be a *managed partition* under the secure core (core 1 can reset core 0 via a unidirectional *reset doorbell*). A byte channel (16 bytes-wide) was established to be the inter-core communication channel between the cores. We set the clock speed of each e500mc core to be 1000Mhz. In addition, we attached caches to the cores (not shown in the figure) for a more realistic environment. Each core has L1 instruction and data caches, each of size 16KB. The cores share a unified L2 cache of size 128KB. We note that without the caches, every instruction execution and data fetch would take 1 cycle on Simics.

The Timing Trace Module (TTM) was implemented by extending the Simics `sample-user-decoder` that is attached to core 0. When the decoder encounters the trace instructions, the relevant information is written to the

non-parametric tests. Thus, the process of detection could be delayed depending on how many samples are used to perform the test.

Table 2.1: Implementation and experimental parameters.

Component	Description
Clock speed	1000MHz
L1 Instruction and Data cache	16KB, 8 ways, latency: 2 cycles
L2 Unified cache	128KB, 32 ways, latency: 10 cycles
Exec. time of complex controller	$[0.85^6, 1.2^6]$ cycles
Exec. times of malicious loops	440, 720, 1000 cycles (1,2,3 loops, resp.)
Min. required probability θ	0.01 or 0.05

SPM. We modified the instruction set architecture (ISA) of the e500mc core [8] so that the execution of the `rlwimi` instruction will trigger an event to the TTM.[8] As shown at the bottom of Figure 2.4, there are four types of trace events differentiated by the last parameter:

- `INST_REG_PID` registers the process ID of the calling application with the TTM,
- `INST_ENABLE/DISABLE_TRACE` enables/disables the trace operations of TTM and
- `INST_TRACE` writes a trace to the SPM.

As mentioned in Section 2.2.3, when tracing is turned on and the `INST_TRACE` instruction is executed, the timestamp and the instruction address at the point of the execution are written at the address specified by the value of $\mathrm{Addr}^{\mathrm{Head}}$. The SPM has a size of 4 KB and is mapped to a region of core 1's address space by the hypervisor.

Lastly, all processes including secure monitor (SM), decision module (DM), I/O proxy (IOP) and the safety controller (SC) run in user-space. Sending/receiving data through the byte channel is done via a kernel module that requests a hypervisor call. The processes (Figure 2.2) execute with a period of 10 ms.

2.4.2 Application Model

As our physical control system, we used an Inverted Pendulum (IP). However, since the simulation speed of Simics is slower by an order of magnitude than the dynamics of a real IP, we used control code and related dynamics generated from a Simulink [22] IP model. These were then encapsulated into software processes. The dynamics process runs on the host PC and is synchronized with the control system managing it in Simics through a pseudo terminal. The physical state of IP is defined as the cart position and the rod's angle (perpendicular to the ground). This state is sent to the controllers executing on Simics and they, in turn, compute an actuation command that is then sent back to the dynamics process that then emulates the action of the IP. For realistic dynamics, we embedded a *Gaussian noise generator* at the output of the rod angle in the dynamics.

[8] `rlwimi` is the *Rotate Left Word Immediate Then Mask Insert* instruction. Execution of `rlwimi 0,0,0,0,i` for $0 \leq i \leq 31$ is equivalent to nop.

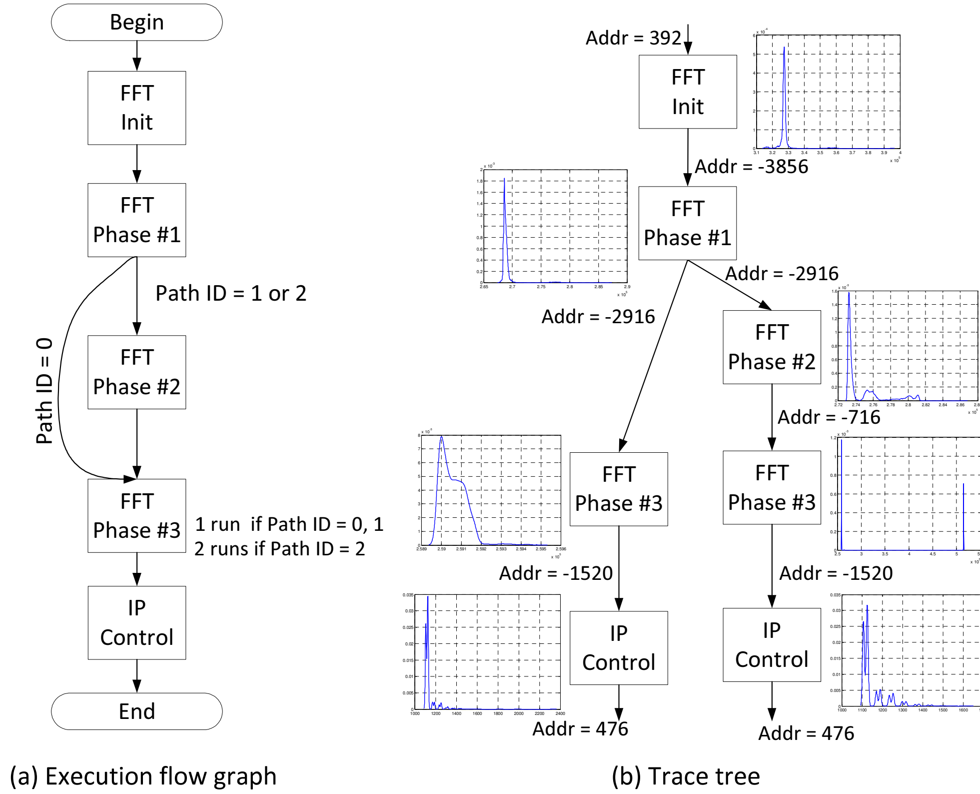


Figure 2.9: The execution flow and the corresponding trace tree of IP+FFT.

As mentioned above, the system runs on Simics and monitors the IP control application. We use the same control code for the complex and simple controllers for evaluation purpose only. However, since the code is too simple with very little variance in execution time, we inserted a fast fourier transform (FFT) benchmark from the EEMBC AutoBench suite [9], `aifft`, to the complex controller as shown in Figure 2.9 (a). The benchmark consists of three phases after initialization. We modified it so that after initialization it randomly selects a path ID. If the ID is '0', FFT Phase 2 is skipped, and Phase 3 is executed twice if the ID is '2'. From this structure, we wish to observe how well our detection methods can deal with execution time variances caused by inputs and flows. After the FFT phases complete, the IP control logic is executed. The logic controls the IP so that it is kept stabilized at position '$+1$' meter from the origin.

We inserted *malicious code* at the end of FFT Phase 3. It is a small loop in which some arrays used in previous FFT phases are copied. The average execution time of the malicious code is 440, 720 and 1000 cycles for 1, 3 and 5 loops respectively. The code becomes activated when the cart position of IP received from IOP becomes $+0.7$ meter. Thereafter the code is executed randomly and the complex controller discards the actuation command calculated by the IP logic and sends out one duplicated from the previous execution. This will result in two effects – variances in execution time that differ from expected values and also wrong actuation information to the control system. Both of

these effects should trigger our detection systems.

To profile the execution times of the complex controller, we inserted `INST_TRACE` instructions at the end of each block and one at the top of each flow (i.e., before FFT_init()) as explained in Section 2.3.2. We executed the system in a normal condition (i.e., no malicious code activation) for $10,000$ runs until we obtained a collection of traces. From these traces, a trace tree is constructed as shown in Figure 2.9 (b). We then used the `ksdensity` function in Matlab to derive the pdf estimation $\hat{f}^k$ of the samples at each block k.[9]

2.5 Result and Discussion

In this section, we evaluate our SecureCore architecture through experiments on the prototype presented in Section 2.4. We then discuss some limitations and possible improvements.

2.5.1 Early Detection of an Intrusion

We first evaluate our timing-based intrusion detection method by measuring how quickly it can detect malicious code execution compared to vanilla Simplex-only approach. As explained in Section 2.4.2, the malicious code embedded in the complex controller is activated when the cart passes through the point at $+0.7$ m. In this evaluation, we set the minimum required probability θ to 0.01 and the loop count of the malicious code to 3. The cart positions were traced from the IP dynamics process for the cases where (a) there is no attack, (b) attack + no protection, (c) attack + Simplex only and finally (d) attack + Simplex + GaIT (our detection method). Additionally, we set an event that is triggered when Simplex or our method detect anomalies. However, for evaluation purposes, we intentionally disabled our method until the cart passes over +0.5 meter; if we enable it from the beginning, a false positive would activate the safety controller before an attack takes place.

Figure 2.10 shows the different trajectories of the cart for the four cases. The cart is stabilized at the position near $+1$ m when there is no attack or if the control logic is protected (either by GaIT or vanilla Simplex). When there is no such protection mechanism, however, the cart becomes destabilized finally after time 25 seconds. When the protection mechanisms are active (SecureCore + GaIT) and the malicious code was activated (at around 6.9 seconds), it was almost instantly detected by GaIT. We can see this from the magnified section of the plot showing the trajectory of the cart along with the normal case (i.e., no attack). On the other hand, although it is not clear in the figure, the Simplex-only method detected the abnormal behavior of the complex controller at around 9.5 seconds. In this case, we see that the cart has deviated from its normal trajectory for a moment; it was later returned to the normal trajectory. Even though the experiment was performed with a restrictive setup for a simple application, the result shows that

[9] We set the number of bins, N, to 1000. The kernel smoothing bandwidth h is then automatically selected by the function.

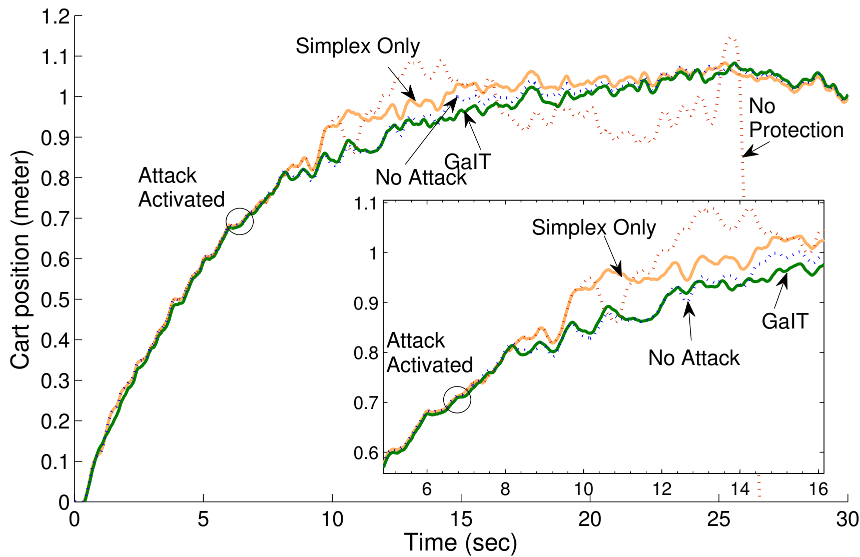


Figure 2.10: Trajectory of cart with different protection approaches.

our timing-based intrusion detection method (GaIT) can supplement Simplex through early detection. Even though vanilla Simplex can detect malicious activity, it does so much later than GaIT and only because the compromised controller tried to actuate the physical system into an unsafe state. Many times, attackers may not send wrong actuation commands – they may snoop on the operation of the system and collect privileged information. SecureCore and GaIT will be able to detect such activity almost instantaneously, as evidenced here, while Simplex will fail to detect it. Also, attackers could increase the wear and tear on the physical system under vanilla Simplex – by causing the system to operate, albeit briefly, in an unsafe state. This can also be avoided by use of our techniques.

2.5.2 Intrusion Detection Accuracy

The early detection capability, however, can be effective only when a higher classification accuracy is possible. Thus, we evaluate the accuracy of our intrusion detection method by measuring the false positive and false negative rates. In this experiment, we disabled the reset mechanism of secure core to correctly count the number of attacks and misclassifications. However, the functionality exists for future work. As mentioned before, a false positive occurs when the monitor classifies an execution to be malicious when it was not and a false negative is when a malicious attack goes undetected. The evaluation was performed with the minimum required probability θ set to 0.01 or 0.05. For each case, the loop count of the malicious code was set to 0, 1, 3 and 5. Then, we sampled decisions made by the secure monitor until we collected at least 1000 samples.

To measure the rate of false positives, we ran the system without activating the malicious code. For $\theta = 0.01$, only *one* false positive out of 1024 samples was found. With $\theta = 0.05$, the monitor classified 7 samples out of 1015

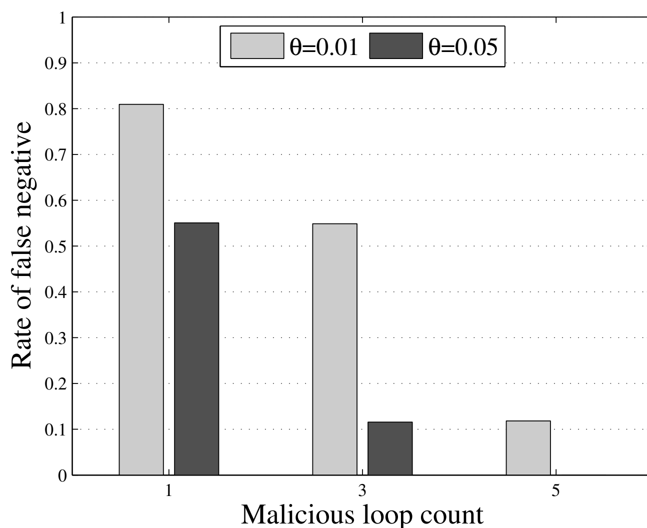


Figure 2.11: False negative rates for different θ and malicious loop counts.

Table 2.2: False negative rates (# attacks missed / # attacks tried).

	1 loop	3 loops	5 loops
θ =0.01	827/1022(81%)	574/1046(55%)	130/1098(12%)
θ =0.05	578/1050(55%)	117/1011(12%)	0/1045(0%)

legitimate executions as attacks. We then activated the malicious code to measure the false negative rates. Table 2.2 shows how many attacks the monitor missed for each θ and loop count. For example, for $\theta = 0.05$ and the loop count of 3, the monitor could not detect 117 out of a total 1011 malicious code executions. As can be seen from Figure 2.11, the false negative rate decreased when the malicious code executed for longer time frames. For the same execution, a higher value of θ also showed reductions in the rate of false negatives. However, as previously mentioned, there is a tradeoff between a higher θ and a lower one. That is, while setting θ higher can reduce the chances that the monitor will not miss malicious code execution, it can also increase the rate of false alarms. In such cases, the control would be frequently switched to the safety controller even if the complex controller is not compromised. This could degrade the overall control performance for the physical system. Thus, a balanced θ should be obtained, either through extensive analysis or through empirical methods.

2.5.3 Limitations and Possible Improvements

The main cause of misclassifications comes from noise during execution time profiling. A legitimate execution time might not appear in the samples but might be observed during the actual monitoring phase. Moreover, the malicious execution time might also fall within a legitimate interval. This is especially possible when an attacker exploits the system by using a short and steady malicious code execution such as an example using the Return-Oriented

Programming (ROP) attack [129]. An ROP-based attack is a sequence of short code blocks each of which would last for less than 100 cycles (or even 10 cycles) before returning to the original execution path. Thus, an attacker may deceive the proposed detection method by executing such short code because legitimate timing variations would likely last longer than the ones caused by such malicious code. As the result in the previous subsection shows, the proposed method would not perform well when such attacks are employed.

Thus, it is the key that we narrow the range of execution time variances as much as possible so that the above situations, i.e., the probability that a legitimate execution instance can fall within the range and that even a short length of malicious execution can deviate from the range is maximized. One way to achieve this is to run the final system on a real-time operating system that inherently has more deterministic execution times. Disabling interrupts during execution (if possible) [103] or locking frequently used data or instructions on cache [38] can help increase the predictability of such executions. In addition, using a real-time multicore processor [93, 107, 158] can further improve the accuracy by reducing or eliminating unpredictable variations in execution times caused from contentions on shared resources such as cache, bus, memory, etc.

2.6 Conclusion

In this chapter, we proposed SecureCore, a novel application of a multicore processor for creating a secure and reliable real-time control system. We used a statistical learning method for profiling and monitoring the execution behavior of a control application. Through the architectural and the theoretical support, our intrusion detection mechanism implemented could detect violations earlier than just a pure safety-driven method, Simplex. This helps in achieving reliable control for physical systems. The isolation achieved by SecureCore and the monitoring mechanisms also prevent attackers from causing harm to the physical systems, even if they gain total control of the main controller. Evaluation results showed that with careful analysis and design of certain parameters, one can achieve a low misclassification rate and higher intrusion detection rates.

Chapter 3

Memory Heat Map: Anomaly Detection in Real-Time Embedded Systems Using Memory Behavior

In this chapter, we introduce *Memory Heat Map* (MHM) to characterize system-wide memory behavior of real-time embedded systems. Our machine learning algorithms automatically (a) summarize the information contained in the MHMs and then (b) detect deviations from the normal memory behavior patterns. These methods are implemented on top of the SecureCore architecture presented in Chapter 2 to aid in the process of memory behavior monitoring and detection. The techniques are evaluated using multiple attack scenarios including kernel rootkits and shellcode.

3.1 Introduction

In this work, we present techniques to detect *system-wide anomalies* in the execution of real-time embedded systems by monitoring *the behavior of memory accesses* for the operating system. Memory access is an important property since it is particularly hard to fake or hide for malicious tasks. We find that the memory profiles of many real-time applications have a predictable nature and use this property to detect malicious activity.

The behavior of memory access for a system can be defined in many ways. For instance, one could track the exact sequence of memory addresses that are accessed. While this could provide very precise information on memory usage, it requires a prohibitive amount of storage not to mention excessive computation times for lookup, etc. Hence, it will be hard to apply in real-time systems with limited resources. Another problem is that such profiles are very sensitive to even legitimate variations. On the other hand, we might still be able to use the memory behavior in a system but relying on coarse-grained information. For instance, consider the amount of memory traffic generated during each time interval. We can use these types of profiles to detect anomalies that significantly change the memory traffic. However, such profiles could abstract away from the detection of small, abnormal variations.

To avoid such problems we present the use of a novel method to profile memory behavior, viz. the *Memory Heat Map* (MHM). The MHM is a concise data structure that represents how many times a particular memory location was accessed (regardless of which component accessed it) during a fixed time interval. An 'access', in this case, is defined as a read/write of either an instruction or data. The key idea is that an MHM is a *composition* of different activities in a certain memory region. Each activity will contribute differently in each MHM. The periodic nature of real-time

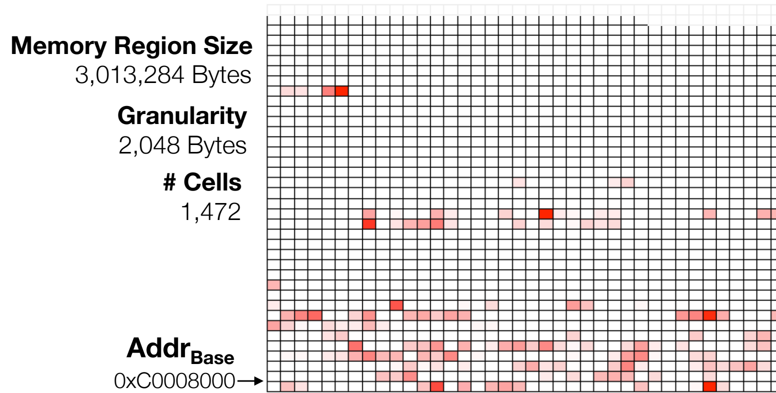


Figure 3.1: An example memory heat map of Linux kernel `.text` segment measured for an interval of 10 ms. The darker the red color for a cell, the more number of times it is accessed.

systems enables us to learn the patterns of usage in such MHMs especially when the system is behaving in a normal, expected fashion. Section 3.2 provides further insights into the construction of such MHMs while Figure 3.1 presents an example of one such heat map. We then apply techniques learned from image recognition algorithms [134, 144] to transform these memory profiles to a more efficient representation so that analysis becomes easier. Section 3.4 explains how we perform these transformations as well as the analysis that follows.

We demonstrate our techniques on a dual core architecture where one core captures the initial profiles and also performs the analysis (at run-time) for anomaly detection. The other core executes the operating system and main applications. This is an extension of the SecureCore architecture (Chapter 2) that we adapt for the process of monitoring memory behavior. The architecture is embellished with certain hardware modifications to ensure that (a) the information can be captured in an efficient fashion *without* affecting the main flow of the system and (b) the information that is obtained by the monitoring core can be trusted. Further details are provided in Section 3.3. We describe our evaluation framework and results in Section 3.5. Experiments on a prototype show that our approach catches various types of anomalies effectively in an efficient manner.

Hence, our main contributions of this work are:

- Novel monitoring model to characterize the memory behavior for real-time embedded systems – in the form of *memory heat maps*.
- The novel combination of the MHM and image recognition algorithms to provide efficient representation and analysis of MHMs that aids in the process of detecting anomalies.
- A multicore-based architecture to perform the profiling and run-time monitoring.

To the best of our knowledge, this is the first work that uses aggregated memory behavior, especially the concept of memory heat maps, for detecting system-wide anomalies.

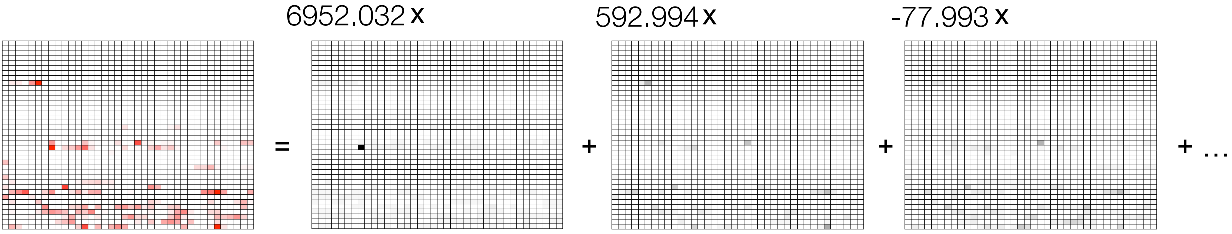


Figure 3.2: An MHM is a linear combination of primary activities on the target memory region.

3.2 The Memory Heat Map

A memory heat map (MHM) is defined by the following triple: the *base address*, $\mathsf{Addr}_{\mathsf{Base}}$, the *size*, S, and the *granularity*, δ. These are configurable parameters (explained more in Section 3.3) that determine where and at what detail we wish to monitor the memory behavior of the system. Figure 3.1 shows an example of an MHM that was profiled from an embedded Linux kernel's `.text` segment (between `0xC0008000` and `0xC02E7AA4`) for an interval of 10 ms. A memory region is divided into *cells*, each with a size, δ. In the example, the Linux kernel's `.text` segment (size S is about $2,943$ KB) is divided into $1,472$ cells of 2 KB each. The 2-D plots of MHMs presented throughout this chapter are for illustrative purposes only. An MHM is in reality a vector like the actual memory space is. The length of the vector is equal to the number of cells.

Each cell counts the number of accesses to a memory address for a specified time interval. One can even consider it to be the *temperature* of each cell (hence the phrase "memory heat map"). In the simplest case, it can be used to detect abnormal "temperatures" of specific cells since they are supposed to be fairly predictable. But, the "temperature" of each cell, on its own, does not reveal useful information; it might exhibit variations due to many factors. However, the state of the *entire map* may reveal important system activities. An MHM is composed by the counts of accesses from a variety of system activities due to applications as well as kernel executions). Figure 3.2 provides an overview of how an MHM is composed.

In fact, the term 'activity' should be understood to be a collection of activities by smaller system components (and applications) as well as the kernel. An MHM represents a composition of memory accesses from a variety of system activities due to applications and OS. Thus, an MHM can be represented by a weighted combination of the *primary activities*; where the weights represent their contributions to the MHM. The figure shows that an MHM is a linear combination of such activities; [1] A good analogy is that of a Fourier series. The key ideas are that (i) normal memory behavior can be grouped into a finite number of sets according to the weights of primary activities and (ii) abnormal behavior can be detected by just looking at these weights. The creation and use of MHMs can be quite efficient since

[1] Precisely speaking, the MHM is the sum of the *average* MHM (not shown in the figure) and the linear combination on the right-hand side.

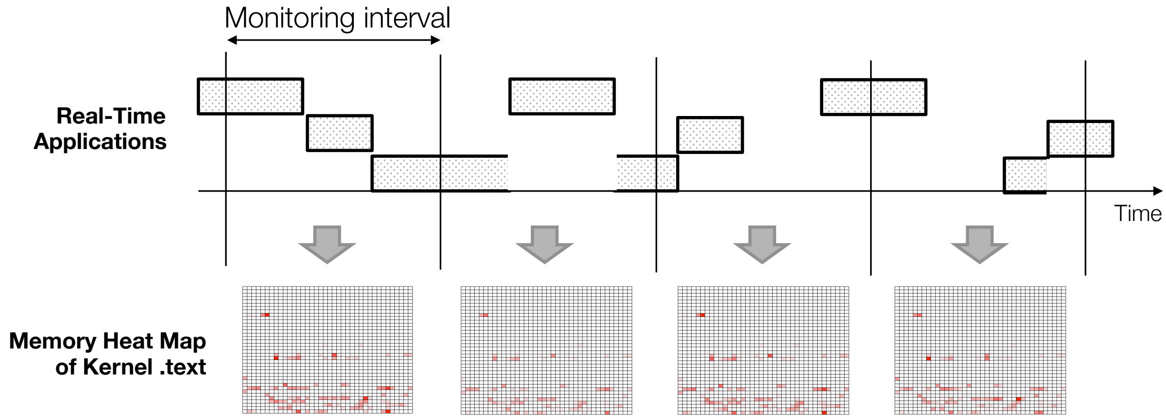


Figure 3.3: The overall memory heat map monitoring process.

it is just a vector of numerical counts. Also, the heat maps depend only on the size of the memory region that we observe and not on the complexity of the kernel and other applications. Both of these make the process of analyzing MHMs tractable and efficient.

3.2.1 Monitoring Kernel Memory Space

In this work, we focus on monitoring the memory space for *the operating system kernel*, though our memory heat map-based method can be applied to any region as long as the system's usage of these memory regions shows predictable behavior. However, monitoring the kernel memory space has the following advantages. First, observation of the kernel memory space provides a good indicator of system behavior since every application has to use kernel services (e.g., system calls) for privileged operations. From the kernel memory space, we can detect certain types of anomalies, e.g., unexpected application launch/kill or even suspicious use of kernel services. Furthermore, monitoring the kernel space makes the hardware design much simpler (when compared to monitoring user-level processes). This is because (i) the (base) kernel's location in the memory space is known and fixed and (ii) it is contiguous in both the virtual and physical memory spaces.[2] Hence, we do not need a complex hardware architecture to deal with the address translation and also memory paging issues.

3.2.2 Overall Process

As shown in Figure 3.3, our anomaly detection framework periodically monitors the MHM of the kernel's memory. At each interval, one MHM is created by the on-chip hardware module (Section 3.3). The anomaly detector analyzes the MHM at the end of the interval. At that point, we calculate the likelihood of this MHM being part of the normal execution. Section 3.4 presents the details of this analysis.

[2] As will be explained in Section 3.3, we monitor the 'base' kernel's `.text` segment. It is in the kernel's logical address space.

3.2.3 Assumptions

We make the following assumptions without loss of generality:

- The system runs a set of real-time applications that execute in a periodic fashion.
- Most of the possible execution contexts due to, e.g., different execution modes and/or inputs, can be profiled ahead of time for higher detection accuracy and lower false positive rate. This can be justified by the fact that real-time embedded applications have a limited set of execution modes and input data fall within fairly narrow ranges.
- The system is in its normal (trustworthy) state while being profiled; also, the profiling is done prior to system deployment.
- We consider certain types of anomalies that make changes in the memory regions that are being monitored; our detection mechanism cannot detect anomalies that access memory segments outside the region under monitoring.

3.3 Monitoring Memory Heat Maps

This original SecureCore architecture presented in Chapter 2 provided the means for a trusted on-chip entity, viz., a secure core, to monitor the run-time behavior of another component, the monitored core. We modified the SecureCore architecture to observe the memory behavior of the monitored core through an on-chip module, *Memometer*. The Memometer aids in the process of creating profiles for, and monitoring, memory heat maps (as explained in Section 3.2). Figure 3.4 shows this new architecture.

The Memometer hooks into the monitored core and continuously snoops upon the memory requests that are sent from it. Using this information, the Memometer periodically creates heat maps that represent the memory usage of the monitored core. The heat maps are then retrieved by the secure core that performs analyses on them as explained in Section 3.4; the analysis determines if the monitored core is behaving in a normal or abnormal fashion.

3.3.1 Memometer

The actual implementation of a Memometer depends on the specific processor architecture especially (a) the memory sub-system and (b) the type (instruction or data) of memory accesses it uses to build the heat maps. In this work, the Memometer snoops on the *address line* between the monitored core and L1 cache because otherwise we would lose memory access information due to cache hit. In addition, we consider an L1 cache is virtually addressed. Thus,

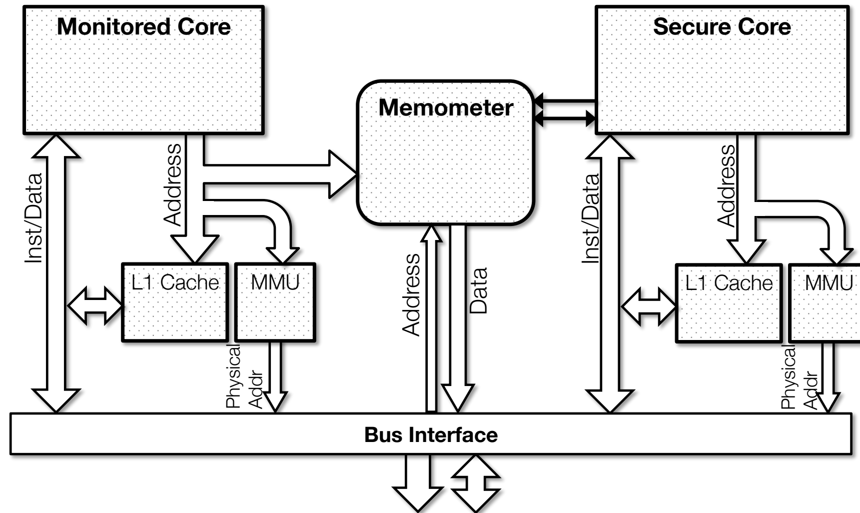


Figure 3.4: The secure core architecture for memory-behavior monitoring using Memometer.

the hook should be placed *before* the MMU as shown in Figure 3.4. In fact, we could monitor physical addresses if the target monitoring region has a linear mapping from virtual to physical address, e.g., kernel logical address. Monitoring the physical addresses is in fact desirable since otherwise, for example, an attacker could potentially execute a malicious code by modifying the memory mappings while making MHMs look normal.

The memory heat maps created by the Memometer can only be accessed by the secure core. The Memometer includes a *fast on-chip memory* for MHM storage that can be physically addressed from the secure core. When an MHM is ready to be analyzed the Memometer raises an interrupt on the secure core. The secure core then performs the analysis on the MHM. However, neither the memory accesses by the monitored core nor the process of monitoring, should be affected while the secure core analyzes the MHM. Thus, we implement a *double buffering* mechanism so that both, the monitoring as well as the analysis can continue in an uninterrupted manner. Figure 3.5 shows the internal structure of the Memometer; we will elaborate on each of the components shortly.

Memometer Controller

The secure core sets the monitoring parameters for the Memometer through control registers. The parameters are (a) the base address of the target monitoring region; (b) the size of the region; (c) the granularity (a power of 2) and (d) the monitoring interval. Parameters (a) – (c) are used to filter a snooped address and to calculate the target cell location (explained below). One memory heat map is created during each monitoring interval, for example, 10 ms.

Address Filtering and Target Cell Calculation

Once a memory address is obtained by the Memometer, a series of filtering operations and calculations are per-

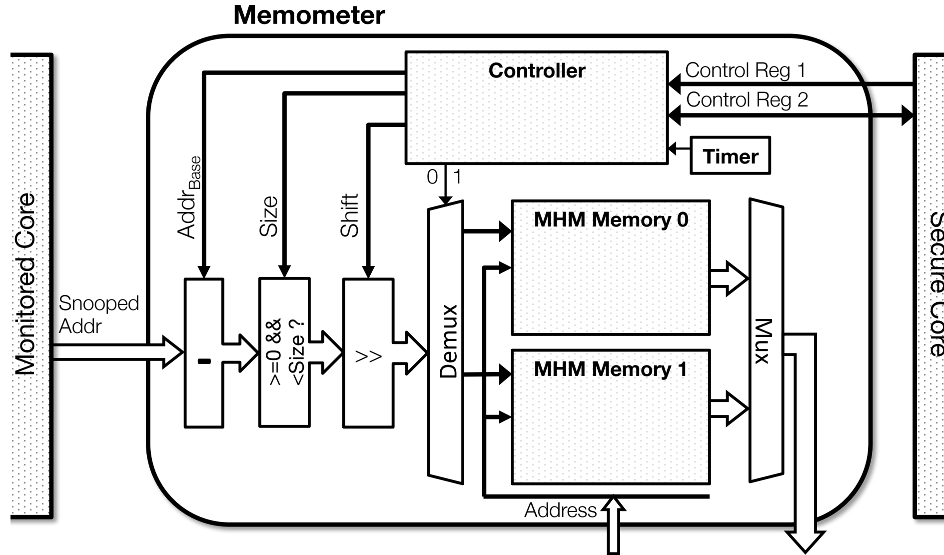


Figure 3.5: The internal structure of Memometer and the interfaces for the monitoring and the secure core.

formed to locate the target cell in the MHM that is currently being built. Let Addr^* be the address that is being accessed by the monitored core. Then, the following steps calculate the target cell index in the current MHM.

1. Calculate the offset, i.e., $\mathsf{offset} = \mathsf{Addr}^* - \mathsf{Addr}_{\mathsf{Base}}$.

2. Check if it is within the target region, that is, $0 \leq \mathsf{offset} < S$ where S is the region size. The process stops if this is false.

3. Logical right-shift offset by g bits where $g = \log_2 \delta$ and δ is the MHM granularity. The resultant is the target cell index,

$$\mathsf{idx} = \left\lfloor \frac{\mathsf{offset}}{2^g} \right\rfloor = \mathsf{offset} >> g.$$

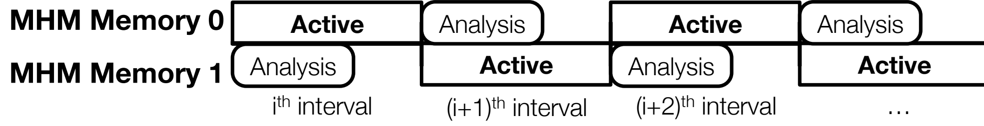
The resulting idx is then used to increment the count of the target cell. If it is out of the bounds (i.e., beyond the maximum cell number) then there is no effect on the MHM. Note that the maximum cell number depends on the MHM memory size and is implementation specific.[3] Note also that the MHM memory size determines the maximum number of cells an MHM can have and not necessarily the maximum size of target memory region. By controlling the granularity parameter, the maximum size of the target region can be determined.

MHM Double Buffering

As mentioned above, one of the key requirements is that the Memometer should be able to monitor/profile the memory heat maps in an uninterrupted fashion. Hence, it should be able to continue monitoring the memory accesses

[3]We skip the related logic from Figure 3.5.

of the main core while a recently completed MHM is being analyzed by the secure core. In our architecture, this is achieved by use of a *double buffering mechanism*. For this purpose, the Memometer has *two identical on-chip memory units*.



The timing diagram above illustrates how this double buffering mechanism works. At any time instant, a cell count update is carried out on what we call the *active* on-chip memory unit (say, '0'). Then, at a monitoring interval boundary, say, between the ith and the $(i+1)^{th}$ intervals the second on-chip memory unit ('1') is tagged as being the active one and starts storing the $(i+1)^{th}$ MHM. At the same time, the secure core starts analyzing the ith MHM residing on the first on-chip memory unit ('0'). Once the secure core is done with the analysis, the old MHM is reset. This double buffering mechanism is driven by Memometer controller. Let the start address of MHM memories '0' and '1' be $\mathsf{Addr_{m_0}}$ and $\mathsf{Addr_{m_1}}$, respectively. Let the size of each be $\mathsf{S_m}$. The two memories are contiguous and thus $\mathsf{S_m} = \mathsf{Addr_{m_1}} - \mathsf{Addr_{m_0}}$ is the size of each. Now, when a cell index idx is provided, the count at the address,

$$\mathsf{Addr_{m_0}} + \mathsf{sel} \cdot \mathsf{S_m} + 4 \cdot \mathsf{idx}$$

is increased by 1, assuming each count has a size of 4 bytes. Here, sel alternates between '0' and '1' at every interval. Thus, $\mathsf{sel} = 0$ (or $= 1$) *selects* MHM memory '0' (or '1'). Meanwhile, the secure core knows the value of sel and thus it can correctly retrieve the MHM from the *inactive* memory location starting from $\mathsf{Addr_{m_0}} + (1 - \mathsf{sel}) \cdot \mathsf{S_m}$.

3.4 Learning Memory Heat Maps

As part of the process of detecting anomalous behavior, we need methods to detect changes in MHMs. In this section, we show how a statistical learning algorithm, based on image recognition methods, can be used for this purpose. We address the following questions: (i) how to characterize the normal heat maps of untainted systems in an efficient and accurate manner and (ii) how to detect anomalies using such normal profiles.

3.4.1 Definitions and Overall Learning Process

Let $\mathcal{M} = \{\boldsymbol{M}_1, \boldsymbol{M}_2, \ldots, \boldsymbol{M}_N\}$ be the set of memory heat maps we obtained during normal (trusted) system executions on the architecture presented in Section 3.3. Each MHM is a vector of length L, where L is the size of the

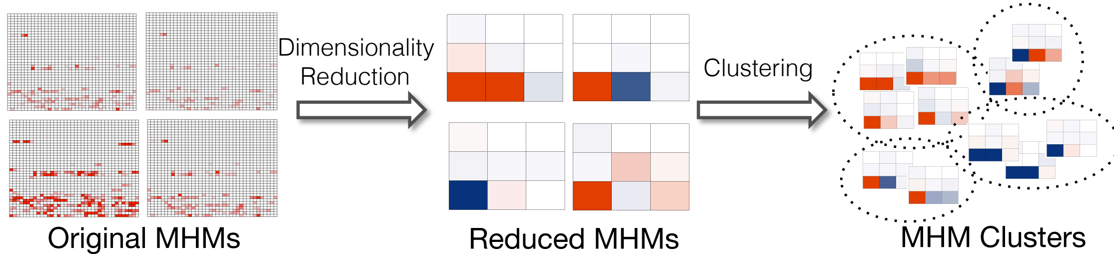


Figure 3.6: Learning normal MHM patterns.

MHM.[4] Hence, the nth MHM is represented as, $\boldsymbol{M}_n = [m_{n,1}, m_{n,2}, \ldots, m_{n,L}]^T$, where $m_{n,k}$ is a non-negative integer which represents the number of memory accesses to the kth cell, $\big[\mathsf{Addr}_{\mathsf{Base}} + (k{-}1)\delta, \mathsf{Addr}_{\mathsf{Base}} + k\delta\big)$, where $\mathsf{Addr}_{\mathsf{Base}}$ is the base address and δ is the cell granularity (see Section 3.2).

$\mathcal{M}$ is the *training set* that represents the normal/expected memory behavior. Using the statistical learning method presented in this section we will calculate the *likelihood* of whether an MHM is either *normal* or *abnormal* with respect to $\mathcal{M}$.

When a new MHM is presented for classification (normal/abnormal), one naive but straightforward way is to compare it with each of the training samples to find if any such heat map has been observed in the past. This approach would not work, especially for real-time systems, since (i) it is computationally prohibitive to calculate the similarity against every known MHM in the training set (N is often large) and (ii) the number of MHM configurations is exponential in L; hence, there can, for all practical purposes, be an infinite number of MHM configurations (particularly if L is large). Hence, it is desirable to find *patterns* in the normal MHMs and then calculate the statistical similarity for the newly observed MHM. This reduces the problem into a more tractable scope. We use an *image recognition algorithm* for this purpose. We treat each MHM as an image and apply an image recognition technique for comparison. The challenges that remain are (a) how to handle high dimensionality of MHMs and (b) how to find *representative* MHM patterns for efficient classification.

3.4.2 Eigenmemory

Memory heat maps are represented in a high dimensional space especially if we monitor a large memory region at a fine granularity. Using such MHMs as is can be computationally expensive even for simple mathematical processes such as Euclidean distance calculations. However, not all of the cells of an MHM include relevant information. Hence, given a large set of MHMs, we can somehow *compress* each one into a low-dimensional image without losing much information. Such a process can be carried out because each MHM is a composition of memory activities for

[4] As explained earlier, it depends on the size of the target memory region and the monitoring granularity. Also, remember that an MHM is a 1-D vector.

different, yet unknown, components in the system. Hence, an MHM can then be viewed as the combination of the memory activities of each of the components.

For purposes of compressing the information contained in an MHM, we use a dimensionality reduction/ feature extraction method, the Principle Component Analysis (PCA) [84], which has widespread uses in image analysis. The PCA transforms data with high dimensionality (in L-dimensions) into low-dimensional features (in L'-dimensions; $L' \ll L$) which of those are called *principal components*. These principal components can compactly represent the original data when many features/dimensions are correlated each other. In the context of image recognition, this is equivalent to the process of extracting a set of basic images, called *eigenfaces* [134, 144][5], that can be linearly composed to reconstruct the original images with a minimal approximation error. In our context, the primary activities of the target memory region are mapped to what we call *eigenmemory*.

The following steps transform a training set $\mathcal{M} = \{\boldsymbol{M}_1, \boldsymbol{M}_2, \ldots, \boldsymbol{M}_N\}$ (where each $\boldsymbol{M}_n$ is an L-dimensional vector) into $\mathcal{M}' = \{\boldsymbol{M}'_1, \boldsymbol{M}'_2, \ldots, \boldsymbol{M}'_N\}$ (where each $\boldsymbol{M}'_n$ is an L'-dimensional vector and $L' \ll L$):

1. Calculate the empirical mean MHM of the training set,

$$\boldsymbol{\Psi} = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{M}_n.$$

2. Obtain the mean-shifted MHM,

$$\boldsymbol{\Phi_n} = \boldsymbol{M}_n - \boldsymbol{\Psi}$$

   for all n.

3. Construct the empirical covariance matrix $\boldsymbol{C}$,

$$\boldsymbol{C} = \frac{1}{N} \sum_{n=1}^{N} \boldsymbol{\Phi}_n \boldsymbol{\Phi}_n^T = \boldsymbol{A}\boldsymbol{A}^T, \quad (3.1)$$

   where $\boldsymbol{A} = [\boldsymbol{\Phi}_1 \boldsymbol{\Phi}_2 \cdots \boldsymbol{\Phi}_N]$ is an L by N matrix. Thus, the size of $\boldsymbol{C}$ is L by L. Then, find the eigenvectors of $\boldsymbol{C}$.[6] The extracted eigenvectors are the *eigenmemories* which of those represent the principal components of the MHMs in the training set.

4. Order the eigenmemories according to their corresponding eigenvalues in decreasing order. Then, pick the L' best eigenmemories, $\boldsymbol{u} = [\boldsymbol{u}_1 \boldsymbol{u}_2 \cdots \boldsymbol{u}_{L'}]$, with largest eigenvalues.

5. Transform each MHM $\boldsymbol{M}_n$ into $\boldsymbol{M}'_n$ by projecting the mean-shifted MHM $\boldsymbol{\Phi}_n$ onto the (L'-dimensional)

[5] It is also known as *eigenpicture* or *eigenimage* as this technique generally applies to image recognition.

[6] When L is large, it can be computationally intractable to compute the eigenvectors of an L by L matrix. In that case, if $N < L$, one can use the Singular Value Decomposition [144] that finds the eigenvectors of $\boldsymbol{C}$ by using the eigenvectors of $\boldsymbol{A}^T\boldsymbol{A}$ which is an N by N matrix.

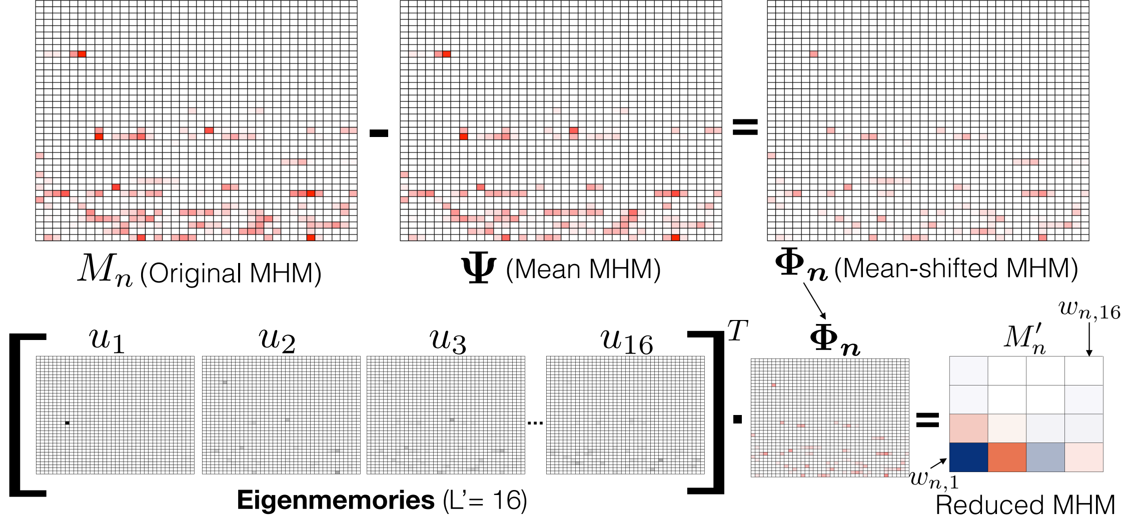


Figure 3.7: An example of the dimensionality reduction from $\boldsymbol{M}_n$ to $\boldsymbol{M}'_n$ using 16 eigenmemories. The 2D plots are only for illustrative purposes. Everything but $\boldsymbol{M}'_n$ is a vector of length $L = 1472$ and $\boldsymbol{M}'_n$ is a vector of length $L' = 16$. The original MHM, $\boldsymbol{M}_n$, can be approximately reconstructed by $\boldsymbol{u} \cdot \boldsymbol{M}'_n + \boldsymbol{\Psi} \approx \boldsymbol{M}_n$

eigenmemory space, i.e.,

$$\boldsymbol{M}'_n = \boldsymbol{u}^T \boldsymbol{\Phi}_n = [w_{n,1}, w_{n,2}, \dots, w_{n,L'}]^T. \tag{3.2}$$

Finally, we have $\boldsymbol{M}'_n$ which is in L'-dimensional space.

It is important to understand that the $w_{n,i}$ values, also called *weights*, represent the contribution of the eigenmemory $\boldsymbol{u}_i$ in representing the original (mean-shifted) MHM, $\boldsymbol{\Phi}_n$ of $\boldsymbol{M}_n$. We can view $\boldsymbol{\Phi}_n$ being approximated using a linear combination of eigenmemories, i.e., $\boldsymbol{\Phi}_n \approx \sum_{k=1}^{L'} w_{n,k} \boldsymbol{u}_k$. Hence, the more eigenmemories we use, the more accurate the approximation will be. Therefore, the best set of L' eigenmemories is one that will retain the best approximation for the original MHMs with regard to the principal components.

Figure 3.7 shows an example where the above process is applied. $\boldsymbol{M}_n$ is an (original) MHM of length $L = 1472$. In this example, we chose the best 16 eigenmemories. Each eigenmemory represents a primary activity; in this case an activity that touches upon the Linux kernel's `.text` segment. Here, u_1 is the most significant activity, u_2 is the next significant, and so on. Then, the resulting $\boldsymbol{M}'_n$ represents the contribution (i.e., weight) of each primary activity from the original memory heat map $\boldsymbol{M}_n$. Thus, different MHMs in the original space can be represented by different combinations of the contributions (weights). Also, notice that the original MHM can be reconstructed (in an approximate manner) by applying the operations in reverse: $\boldsymbol{u} \cdot \boldsymbol{M}'_n + \boldsymbol{\Psi} \approx \boldsymbol{M}_n$.

These eigenmemories are stored in the secure core and used to transform every newly obtained MHM $\boldsymbol{M}$ into $\boldsymbol{M}'$ (after appropriate mean-shifting) using Eq. (3.2). The classification process, presented in the next subsection, is carried out in the reduced dimensional space.

3.4.3 Finding MHM Patterns

Now that MHMs have been transformed into low-dimensional feature vectors, the problem becomes one of classification. That is, given $\mathcal{M}' = \{\boldsymbol{M}'_1, \boldsymbol{M}'_2, \ldots, \boldsymbol{M}'_N\}$ (see Figure 3.8 in Section 3.5.2 for an example), we need to check if a test MHM, $\boldsymbol{M}$, appropriately transformed into $\boldsymbol{M}'$ (say, one in Figure 3.10 in Section 3.5.3), represents normal or abnormal behavior. The simplest way is to look up the database of normal memory heat maps in the training set $\mathcal{M}'$ and check to see if similar MHMs exist there. This, however, would not work for our on-line analysis mechanism because of the high computational costs. Note that in what follows, we will use $\mathcal{M}$, $\boldsymbol{M}$, and L instead of $\mathcal{M}'$, $\boldsymbol{M}'$, and L' for notational convenience. Also, we will use the term memory heat maps to denote the ones in the *reduced* dimensional space.

A better approach is to identify a small set of *representative* MHM patterns (in the low-dimensional space) that are significant enough to cover most of the normal MHMs. When a test sample $\boldsymbol{M}$ is provided, we check if it is statistically similar to one of them. For this purpose, we use a cluster analysis, *Gaussian Mixture Model* (GMM) in particular. GMM has been widely used in image/signal processing including image clustering, segmentation, retrieval, etc. [36, 117, 41] due to its ability to approximate various probability distributions and its computational efficiency.

In GMMs, the *probability density* of a memory heat map is represented as a weighted sum of J multivariate Gaussian,

$$\Pr(\boldsymbol{M}; \text{GMM parameters}) = \sum_{j=1}^{J} \lambda_j f(\boldsymbol{M}|\boldsymbol{\mu_j}, \boldsymbol{\Sigma_j}), \tag{3.3}$$

where λ_j is a mixing parameter ($\sum_{j=1}^{J} \lambda_j = 1$ and $0 \leq \lambda_j \leq 1$) and represents the prior probability that MHMs have been generated from the jth Gaussian probability density,

$$f(\boldsymbol{M}|\boldsymbol{\mu_j}, \boldsymbol{\Sigma_j}) = \sqrt{(2\pi)^L |\boldsymbol{\Sigma}_j|}^{-1} \exp\{-\frac{1}{2}(\boldsymbol{M} - \boldsymbol{\mu}_j)^T \boldsymbol{\Sigma}_j (\boldsymbol{M} - \boldsymbol{\mu}_j)\},$$

where $\boldsymbol{\mu}_j$ and $\boldsymbol{\Sigma_j}$ are the mean vector and the covariance matrix of the jth component. By modeling the normal memory heat maps as a GMM we treat them as if they have been generated from a set of significant patterns, each of which is modeled as a Gaussian distribution (component). This is a valid model since if the system shows deterministic memory behavior, it can generate only a limited number of patterns; each MHM is then a result of small variations from one or more of these patterns. Intuitively speaking, the MHMs generated from the same basis pattern (a multivariate Gaussian) have similar weights for each eigenmemory (primary activity). Anomalies therefore inherently result in low likelihood because some of their components have not been seen in the normal memory behaviors.

In order to model the normal MHMs as a GMM, we need to estimate the parameters, $\boldsymbol{\mu}_j$, $\boldsymbol{\Sigma_j}$, λ_j, and even J.

For the estimation of the mixture parameters, we use the expectation maximization (EM) algorithm [54]. The EM algorithm is iterative in nature and optimizes parameters in such a way that the likelihood of the data set is maximized. However, the EM algorithm has a drawback in that the quality of parameters is sensitive to the initial values and the number of components, J, must be known. Since these techniques are out of the scope of this work, we employ the standard EM algorithm with a manually chosen J.[7]

In summary, given J and a training set $\mathcal{M}$ that is in the reduced-dimensional space,

1. We obtain the parameters by applying the EM algorithm to $\mathcal{M}$.

2. When a test MHM is presented at run-time, we calculate its probability density using Eq. (3.3). If it is below a threshold θ, we consider it to be anomalous.

3.5 Evaluation

In this section, we evaluate the memory heat map approach on a prototype running a set of embedded benchmark applications. We present the methods for training the analysis engine in Section 3.5.2 and then show the effectiveness of our methods in detecting anomalies in Section 3.5.3.

3.5.1 Prototype Implementation

We implemented a prototype of the memory heat map monitoring mechanism on the Simics full system simulation platform [98] that allows microarchitectural modifications. We used an ARM Cortex-A9 processor [4] that consists of two cores. Each core runs at 1000 MHz and has L1 instruction and data caches each of size 32 KB. The L1 instruction cache is virtually indexed and physically tagged while the L1 data cache is physically indexed and tagged. The cores share a unified L2 cache of size 512 KB. The main memory is 512 MB.

The Memometer is implemented as an on-chip hardware module in Simics as shown in Figure 3.4. The Memometer monitors instruction fetches by snooping on the address bus. The Memometer has two fast, on-chip, memories (Figure 3.5) each of size $\mathsf{S_m} = 8$ KB. Hence, it can support an MHM of at most about $2,000$ cells, each of which counts up to 2^{32}. The memories are physically addressable and can be *only read by the secure core*.

The monitored core side runs embedded Linux kernel 3.4 [7] that main applications run on. The Memometer is configured by the secure core to monitor the kernel's `.text` segment that is mapped between `0xC0008000` and `0xC02E7AA4` (about $2,943$ KB). The secure core runs a *secure monitor process* that performs the analysis on the MHMs and the initial configuration of the Memometer controller.

[7] Figueiredo et al. [62] present methods to deal with these problems.

Table 3.1: MiBench [69] applications running on the monitored core.

	Exec. Time	Period	Category
`FFT`	2 ms	10 ms	telecomm
`bitcount`	3 ms	20 ms	automotive
`basicmath`	9 ms	50 ms	automotive
`sha`	25 ms	100 ms	security

For the applications, we used MiBench [69], a representative benchmark suite for embedded systems. A set of MiBench applications (listed in Table 3.1) run on the monitored core where various kernel threads are running as well:[8] A longer *hyper-period* (i.e., the least common multiple of periods) would require a more number of training samples, eigenmemories, and/or GMM components.

3.5.2 Training

To obtain a training set of MHMs for characterizing the *normal* system state, we executed the system with the benchmarks (Table 3.1) and collected 10 sets of normal MHMs each of which spans 3 seconds. The system was reset before collecting each new set. We set the monitoring interval to 10 ms and the granularity, δ, to $2,048$ bytes, both of which are arbitrarily chosen. Each of the normal data set consists of 300 MHMs, which results in a grand total of $3,000$ MHMs each of which has $1,472$ cells ($\lceil 2,943KB/2KB \rceil$). The total number of accesses to the target region, which is the sum of all the cells in an MHM, varies between $21,000$ and $88,000$. Considering that each instruction takes a few CPU cycles this number is small enough compared to the monitoring interval of 10 ms. Meanwhile, the cell that has the highest access count is around $38,000$. In this case, a 2 byte-long cell size would have been enough.

We then applied the learning method (Section 3.4.2) on the training set to transform it into a low-dimensional space. To find a proper number of ***eigenmemories***, we used a heuristic that uses the cumulative eigenvalues, $cev_K = \sum_{k=1}^{K} \alpha_k / \sum_{k=1}^{L} \alpha_k$, where α_k is the eigenvalue of $\boldsymbol{u}_k$. The eigenvalues are sorted in decreasing order. One can choose the minimum K such that, $cev_K >= 95\%$, 99% or 99.99%, and use it as the number of eigenmemories. This implies that the first K eigenmemories can account for cev_K of the variances in the training set and hence a higher cev_K achieves a better approximation accuracy. We used 9 eigenmemories, since they could account for more than 99.99% of the variances in the original training set. Having more than 9 eigenmemories does not help improve the approximation accuracy.

Figure 3.8 shows the first 50 intervals (i.e., between $t = 0$ ms and $t = 500$ ms) for one of the training sets. As the figure shows, although there exist some variations (notably, for example, in the interval $23, 24, 39$, etc.), the patterns repeat every 100 ms which is actually the *hyper-period* of the applications. Also, notice that the first few eigenmemo-

[8] The applications were not picked for any specific reason other than the fact that they are representative embedded benchmarks. The execution times were measured on Simics. The periods are manually assigned based on the execution times and the system load (78%).

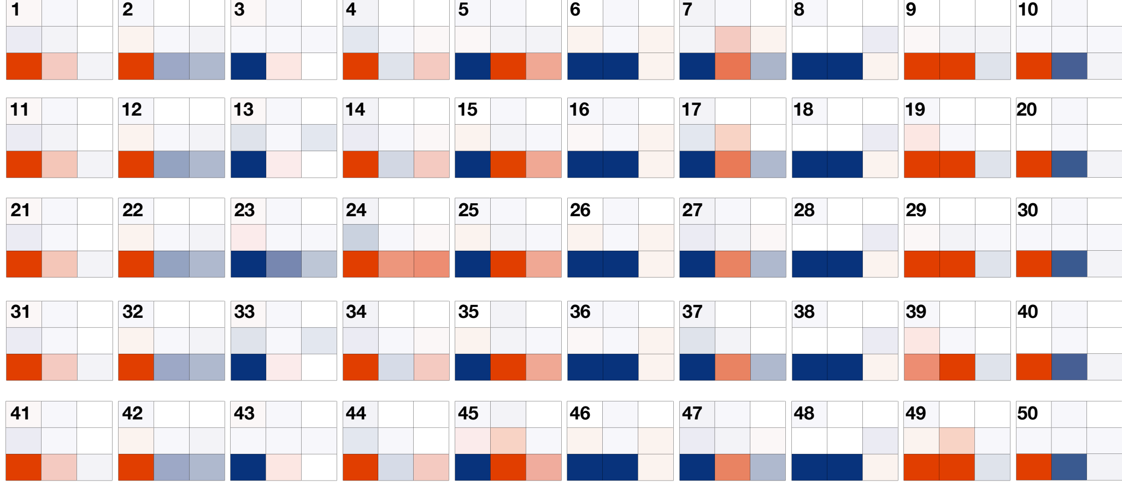


Figure 3.8: The reduced MHMs for the first 50 intervals when the system is in a normal state. The patterns repeat every 100 ms with small variations.

ries are significant enough to distinguish different patterns. In this example, the first two weights, i.e., the blocks at the bottom-left and at the bottom-center in each, are the weights corresponding to the first two eigenmemories. In fact, cev_2 is already 99.7%.

With these reduced MHMs, we learn the GMM parameters, $\{\boldsymbol{\mu}_j, \boldsymbol{\Sigma_j}, \lambda_j | j = 1, 2, \ldots, J\}$, using the EM algorithm [54]. For the number of Gaussian densities, we arbitrarily chose $J = 5$. Due to the local optimality of EM, we ran the algorithm 10 times and picked the one that resulted in the highest log-likelihood of the training data, i.e., $\sum_{i=1}^{N} \log \Pr(\boldsymbol{M}_i)$. Again, one can apply a deterministic learning method [62].

The last step is to find a proper threshold θ that will be used to check whether an MHM obtained at runtime indicates abnormal behavior. We collected another set of normal MHMs and calculated the probability density of each MHM in the set as if we were testing them. The resulting likelihoods are the probabilities that we would see such MHMs in the normal system state. We use this information to decide θ. Specifically, let $\mathcal{P}$ be the probability densities of this new set calculated by Eq. (3.3). Then, we set θ to the p-quantile of $\mathcal{P}$ where p can be 0.5%, 1%, and so on. This means the *expected* false positive rate is p. As θ increases the false positive rate would also increase while we would more likely detect abnormal MHMs. Hereafter, we denote the threshold corresponding to p-quantile as θ_p.

3.5.3 Anomaly Detection

To demonstrate our method's ability to detect a broad range of anomalies, we consider the following three scenarios: (1) addition/deletion of an application; (2) shellcode execution and (3) kernel rootkit loading and execution. We now look at each of these in more detail.

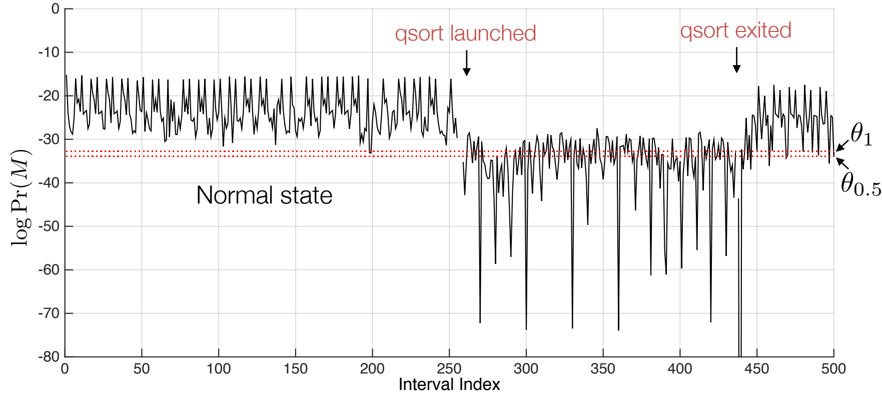


Figure 3.9: The log probability density of MHMs when `qsort` is launched and exited.

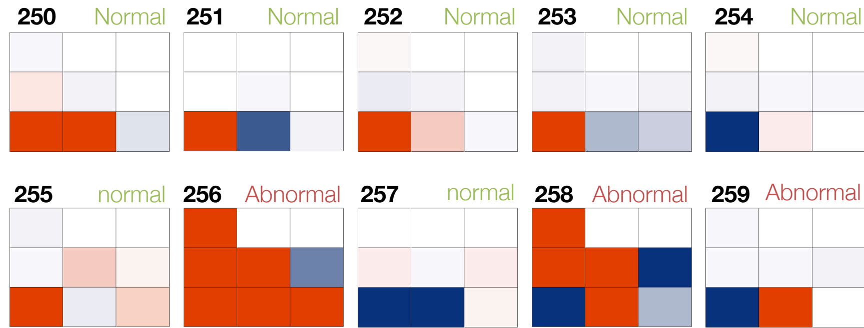


Figure 3.10: Reduced MHMs (using 9 eigenmemories) for the intervals between 250 and 259, in which `qsort` is launched.

1. Application Addition/Deletion

While the MiBench benchmark applications mentioned above are running, we launched another application, `qsort` (execution time: 6 ms, period: 30 ms). Figure 3.9 shows the log probability density of the MHMs monitored over 5 seconds (i.e., 500 intervals). The two horizontal lines show the thresholds corresponding to the 0.5% and 1% quantiles determined from the new normal set (explained above). Until the 250th interval, our anomaly detector determined that 0 and 2 MHMs are abnormal according to $\theta_{0.5}$ and θ_1, respectively; these values are the false positive rates of 0% and 0.8%, respectively.

The `qsort` application was launched some moments after the 250th interval. The figure shows that the probability densities drop immediately and stays low afterward. Figure 3.10 shows the sequence of reduced MHMs obtained for the intervals between 250 and 259. Compare these with the ones from Figure 3.8, for example from 9 to 18.[9] The MHMs of interval 256, 258 and 259 are notably different from those for normal execution. Hence, they represent low probability densities as shown in Figure 3.9. Now, if we look at the original MHMs, we can identify what makes them look abnormal. Figure 3.11 shows the original MHMs for intervals 236, 246 and 256. As we can see, the 256th

[9] Notice that the sequence was drifted by an interval. This is because we did not try to fine-tune the timer.

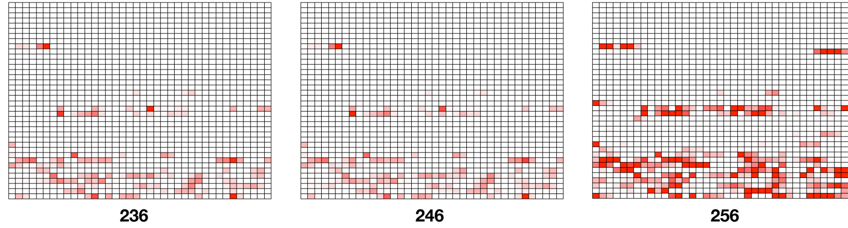


Figure 3.11: The original memory heat maps for intervals 236, 246, and 256 when `qsort` is launching.

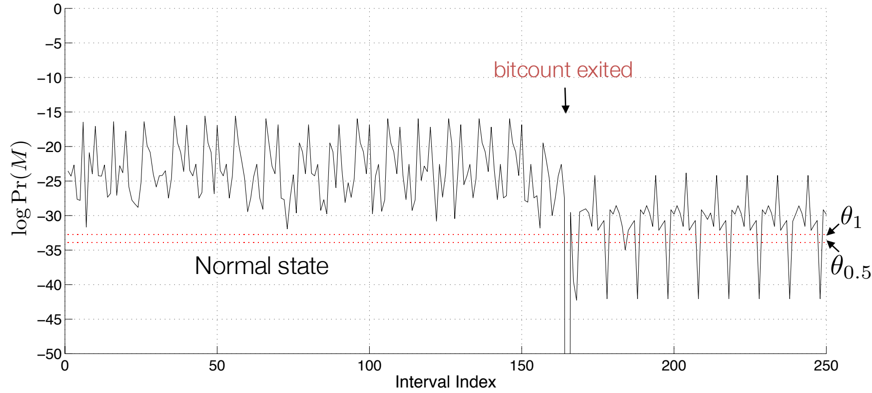


Figure 3.12: The log probability density when `bitcount` exits at around 170th interval.

interval is different from what is expected; its probability density was 0 – this explains why there is a discontinuation in the log probability density plot. In this particular situation, the abnormality is due to the use of kernel facilities to launch the new application (`qsort`). Also, notice that even after `qsort` is launched some of the MHMs look normal according to threshold θ_1. This is valid since during those intervals `qsort` does not execute. Nevertheless, they are low compared to normal because the timings of the other tasks are affected by `qsort`.

We tested another scenario where one of the applications, `bitcount`, exits at an unexpected time. As shown in Figure 3.12, the monitor was able to easily detect the point in time when the application exited and also its absence after that. This is due to that the contributions of `bitcount` to those MHMs are not present anymore, thus changing their nature.

2. Shellcode Execution

A shellcode is a small piece of code that can be injected in the guise of data. It is typically sent over a network or embedded in a file and can be executed by exploiting certain vulnerabilities, e.g., buffer overflows or format string vulnerabilities. In our evaluation, we inject a shellcode into the `bitcount` application which results in the execution of the code at a certain point. This is a much simpler way to launch a shellcode attack than exploiting a vulnerability such as buffer overflow. Thus, the result would be more significant if such attack was employed.

Figure 3.13: The log probability density when a shellcode kills all processes.

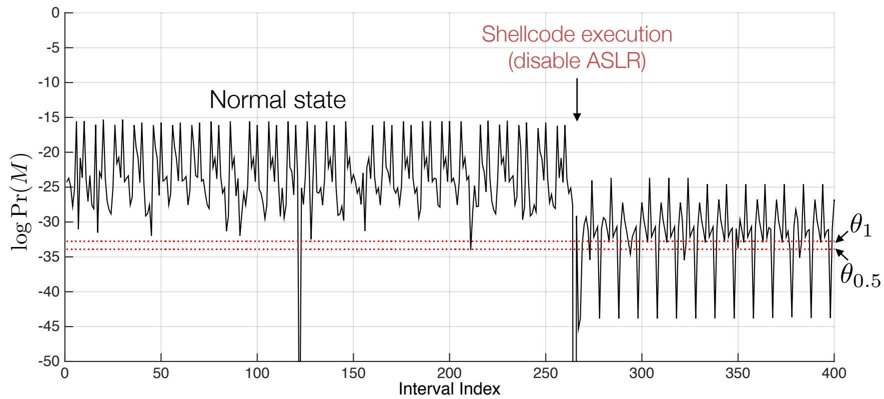


Figure 3.14: The log probability density when a shellcode disables ASLR.

We tested two pieces of shellcode that target Linux on ARM processor.[10] The first one kills all the processes by first obtaining root privileges (using `setuid(0)`) and then calling `kill(-1, SIGKILL)`. This shellcode is easily detectable as shown in Figure 3.13 since all the applications disappear. An escalation in the privileges for an application in itself makes the MHMs look very abnormal (and thus made the probability density to be zero) because such activity had not been seen before, during the normal system execution.

The next shellcode disables the address space layout randomization (ASLR)[11] mechanism in Linux/ARM [16]. Again, this shellcode was launched through the `bitcount` application. The result is shown in Figure 3.14. The shellcode executed some moments after the 250th interval. This shellcode was easily detectable because the shellcode eventually kills its original host, i.e., `bitcount`. In fact, most shellcodes can be detected because they typically kill the host process by spawning a shell.

[10] `http://shell-storm.org/shellcode`

[11] ASLR randomizes process memory layout to prevent an attacker from exploiting buffer overflows.

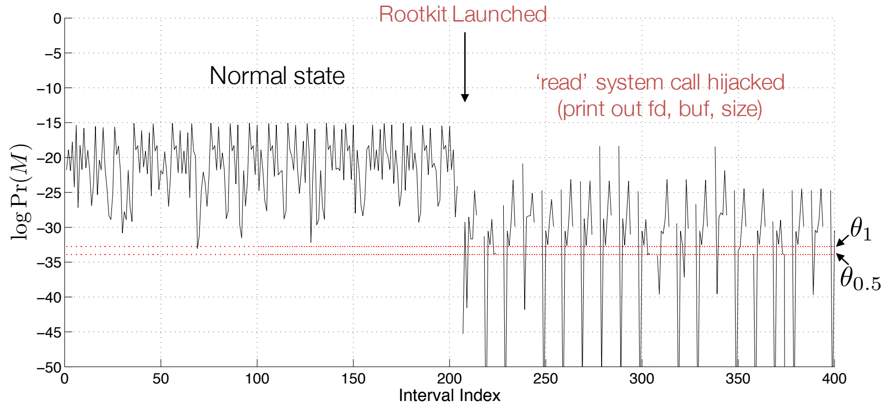


Figure 3.15: The log probability density when a rootkit hijacks `read` system calls.

3. Kernel Rootkit

Kernel rootkits [21] are software that subvert the kernel to obtain privileged access to kernel data structures and perform malicious activities while hiding from detectors. Such rootkits typically get into the system in the guise of loadable kernel modules (LKMs); an LKM gives attackers the same privilege as those for the base kernel. Most existing kernel rootkits that are publicly available do not work on our Linux kernel version (3.4) for a variety of reasons. Thus, we created a simple LKM that resembles the most representative type of such rootkits, i.e., ones that perform system call hijacking [121]. Our LKM redirects the `read` system call by modifying the corresponding entry in the system call table, `sys_call_table[__NR_READ]`. The new, malicious, `read` handler just prints out the file descriptor number, the buffer address and the number of bytes that are read. It then calls the original handler so that the process that issued the original `read` would be served correctly (thus escaping detection). Note that an LKM in Linux is loaded onto the kernel's *virtual* address space that is outside our target memory region (i.e., the *logical* address space). Thus, the execution of the new `read` handler is not captured, although that of the original handler is still noted.

Figure 3.15 shows the log probability density of MHMs monitored in this scenario. The rootkit was launched by `modprobe` at around the 200th interval. This resulted in a zero probability density due to the kernel functions such as `load_module` that loads kernel modules. After the launch, some MHMs indicated low probability densities especially those when `sha` executes. It calls some `fread` function which eventually leads to the `read` system call. Our rootkit hijacks it and prints out information using `printk` function; this makes the MHMs look abnormal. As the figure shows abnormal MHMs repeat roughly every 10 intervals – this is `sha`'s period.

Next, we made the rootkit stealthier; our `read` handler first executes the original handler and then just reads the buffer that is returned by the original handler and nothing else. Figure 3.16 shows the memory traffic volume of the monitored region. The moment when the rootkit is being loaded is distinguishable as expected. However, after the

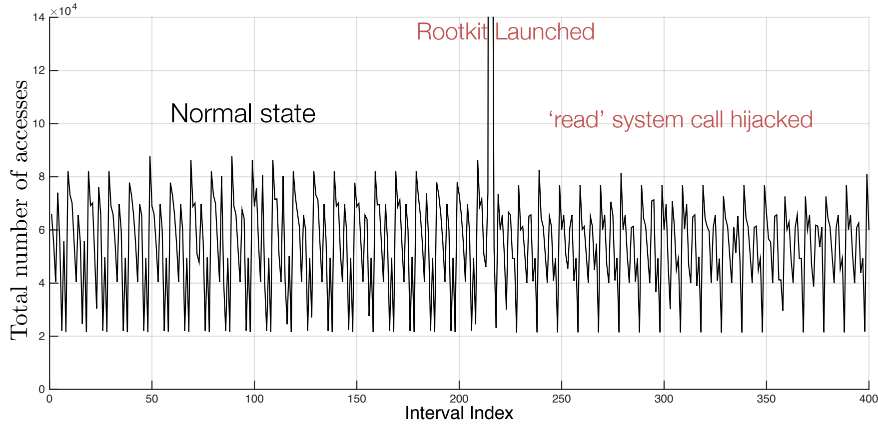


Figure 3.16: The memory traffic volume when the `read` system call is hijacked by a rootkit.

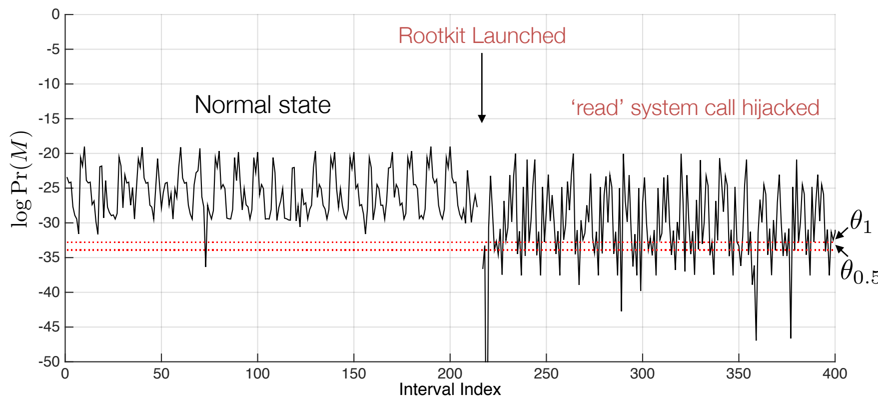


Figure 3.17: The log probability density when a (stealthier) rootkit hijacks `read` system calls.

launch the traffic does not show abnormalities in terms of the volume. This is because the rootkit still calls the original `read` handler which resides in the region being monitored.

Nevertheless, even such stealthy activities (reading the buffer) showed somewhat low probability densities, though not always statistically distinguishable, as shown in Figure 3.17. In fact, even those MHMs that show low probability densities after the rootkit was launched did not provide visible clues on what caused such results. However, given that many MHMs are normal and the abnormal ones' appear synchronized with `sha` (whose period is 100 ms), it is likely that the delays due to `read` system call hijacking have resulted in timing changes to `sha`'s execution (which uses many `read` system calls) and, as a result, its contributions to the MHMs.

It should be noted that finding the location of the `sys_call_table` is not a straightforward task (for attackers) since it is not a global symbol in newer kernels. It necessitates a search in the kernel address space and this would likely increase the chance that the memory heat maps look abnormal. In our implementation, we obtained the address for the `sys_call_table` from the `System.map` and hard-coded it in the module. Thus, no run-time search was

needed which, in reality, makes it harder to detect than the real-world scenarios. In spite of this, there were enough differences in the MHMs. When real attack codes search for the table the differences will stand out more.

3.5.4 Analysis Time

We measured the time to perform the analysis on a newly observed MHM, i.e., how long it takes to decide whether it is normal or not. For the parameters used in the evaluations above ($L = 1472$, $L' = 9$, $J = 5$), it took 358 μs, on average, on Simics. This time is very short compared to the interval (10 ms). For a coarse cell granularity of 8 KB (results in $L = 368$), it took 100 μs on average. For a smaller number of eigenmemories ($L' = 5$), the average time is 216 μs since the information is less precise. Each number is based on $1,000$ samples of MHMs. The results show the computational efficiency of our method. Note that these are the times spent on the *secure core*. Hence, our methods do not impose any overheads on the execution of the main system (i.e., the OS and applications running on the monitored core).

3.5.5 Limitation

The proposed architecture is based on a dual-core configuration in which one core, secure core, analyzes the memory heat map created from another core. When extending the proposed technique to support more than two cores, while the on-line analysis overhead would be still negligibly small as explained in Section 3.5.4, the required hardware change in the Memometer could be overhead. For AMP (Asymmetric Multiprocessing) architectures on which multiple operating systems run, the Memometer should be replicated for each OS instance. However, for SMP (Symmetric Multiprocessing) architectures on which a single operation system runs, the Memometer would need only one set of MHM memories (see Figure 3.5) as there is only one OS kernel running on the system. Nevertheless, each monitored core still needs to be hooked by the Memometer because the OS kernel can run at any core at any given time. Hence, the address snoop and filtering logic should be replicated on the Memometer.

One solution to scale the proposed technique well with additional cores could be placing the Memometer at a lower part of the memory subsystem such as the shared cache or even on the bus. In this case, we would need only a single Memometer for the cores being monitored, which significantly simplifies the architecture.[12] However, the accuracy of the memory behavior model could drop since the Memometer loses parts of memory access information due to cache hits. Nevertheless, we expect that the accuracy drop would not be significant because of the predictive nature of real-time application executions.

Some systems may exhibit highly unpredictable, but yet legitimate, memory usage caused by, for example, network

[12]However, an AMP configuration requires an additional logic to figure out from which core a memory transaction comes. Thus, an SMP architecture would be more suitable for the proposed technique.

activities or user interactions. In these cases, our current model may alarm many false positives. To deal with such problems, we plan to build a robust classification algorithm by extracting local features from MHMs in an unsupervised manner as in Deep Learning [74].

3.6 Conclusion

We showed that the use of memory heat maps can be effective in detecting anomalous system-wide behavior of real-time embedded systems. We demonstrated a novel use of image recognition algorithm and a multicore-based architecture to make the process of detecting anomalous behavior more efficient. Our evaluation using a prototype showed that we are able to detect a wide variety of system-wide anomalies.

In this work, we used a general-purpose operating system. Our techniques will be even more effective when applied to a real-time operating system (RTOS) as it has a *more* deterministic memory usage. Possible future work also includes extending the architecture to support more than two cores and evaluating the required hardware changes, and exploring Deep Learning-based technique to deal with more complex embedded systems.

Chapter 4

Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded Systems

This chapter proposes a lightweight method for detecting anomalous executions using a *distribution of system call frequencies*. In contrast to previous chapters, we target general types of (more complex) embedded systems that do not necessarily show regular patterns in low-level (e.g., timing or memory) behaviors. We aim to detect abnormal *high-level execution contexts*. We use a cluster analysis to learn the legitimate execution contexts of embedded applications and then monitor them at run-time to capture abnormal executions. Our prototype applied to a real-world open-source embedded application shows that the proposed method can effectively detect anomalous executions without using sophisticated analyses or affecting the critical execution paths.

4.1 Introduction

Traditional behavior-based anomaly detection systems rely on specific *signals* such as network traffic [71, 135], control flow [35, 52], system calls [63, 104, 46], etc. The use of system calls, especially in the form of sequences [63, 76, 150, 100, 61, 140], has been extensively studied in behavior-based anomaly detection for general-purpose systems since malicious activities often use system calls to execute privileged operations on system resources. In this chapter, we present an anomaly detection mechanism for embedded systems using a *system call frequency distribution* (SCFD). Figure 4.1 presents an example. It represents the *numbers of occurrences of each system call type* for each execution run of an application. The key idea is that the normal executions of an application whose behavior is regular can be

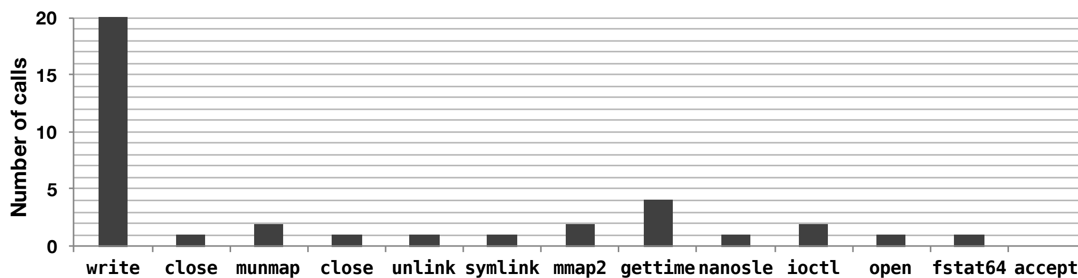


Figure 4.1: A system call frequency distribution (SCFD) obtained from `Motion`.

modeled by a small set of distinct system call distributions (e.g., Figure 4.12 in Section 4.5), each of which corresponds to a high-level *execution context*. We use a *cluster analysis* to learn *distinct* execution contexts from a set of SCFDs and to detect anomalous behavior.

Our detection method is lightweight, has a *deterministic* time complexity – hence, it fits well for resource-constrained embedded systems. This is due to the coarse-grained and concise representation of SCFDs. Although it can be used for offline analysis, we demonstrate an implementation on an embedded computing board [20] and show that minor modifications to the operating system and architectural supports from modern embedded processors enable us to monitor and analyze the run-time system call usage of applications in a secure, non-intrusive manner. We use a real-world open-source application [18] and demonstrate that SCFDs can effectively detect certain types of abnormal execution contexts that are difficult for traditional sequence-based approaches. Detailed results including the sequence-based security analysis is presented in Section 4.5.

Hence, the high-level contributions of this work are:

- We introduce a lightweight method, utilizing the predictable nature of embedded system behaviors, with a deterministic time complexity for detecting anomalous execution contexts of embedded systems based on the *distribution of system call frequencies*.
- We present a detailed security analysis on a real-world application and successful attacks that can fundamentally circumvent sequence-based detection methods.
- We demonstrate our technique with the target application on an embedded computing board and evaluate its advantages and limitations using various attack scenarios.

4.2 Overview

The main idea behind SCFD is to learn the normal system call profiles, i.e., *patterns in system call frequency distributions*, collected during legitimate executions of a sanitized system. Analyzing profiles is challenging especially when such profiles change, often dramatically, depending on the execution modes, events, and inputs. We address this issue by *clustering* the distribution of system calls capturing legitimate behavior. Each cluster then can be a *signature* that represents a high-level execution context, either in a specific mode/event or for similar input data. Then, given an observation at run-time, we test how similar it is to each previously calculated cluster. If there is no strong statistical evidence that it is a result of a specific execution context then we consider the execution to be malicious with respect to the learned model.

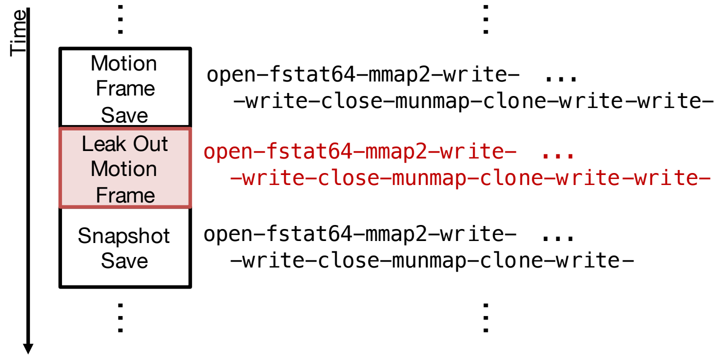


Figure 4.2: Sequence of system calls made by `Motion` (used in the evaluation in Section 4.4). An attacker can use the exact same routine used by the legitimate code to leak an image out.

4.2.1 Attacks against Sequence-based Approach

Although sequence-based methods can capture detailed, temporal relations in system call usages, they may fail to detect abnormal *execution contexts*. This is because sequence-based approaches fundamentally profile the *local, temporal* relations among system calls within a limited time frame. Figure 4.2 highlights such a case. The system calls shown in the figure are generated by `Motion` [18], an open-source motion detection application used in our evaluation. Each `Motion` loop saves the current motion frame to the filesystem if a motion is detected (the top block in the figure). A snapshot is saved too (the bottom block), independently, at a regular interval (e.g., once per 5 seconds). These two blocks use the same routine to save the images to files and thus generate same sequence of system calls as depicted. We were able to insert a small piece of code that leaks out the current motion frame to a desired location in the filesystem while making the resulting system call sequences still look legitimate (the detail is given in Section 4.5.1). This was possible because (i) the sequence patterns generated only by the inserted block are identical to those made by the other two blocks (since the same routine is used) and (ii) no new patterns are generated by transitions across the blocks. Note that if only one of the legitimate blocks execute, the resulting sequences are still legitimate because the inserted block looks like the other block that did not execute. The only way a sequence-based approach can detect such a malicious execution is to know patterns that are long enough to learn the temporal relationship between the two legitimate blocks. That is, the expected sequence patterns must know what system calls should follow after two file operations. However, this is highly unlikely because the required pattern lengths are often too long and also can vary greatly due to variations in data (i.e., image) sizes.

An attacker who has access to the target application code can implement such a stealthy, malicious code that modifies the high-level execution context while not disturbing the system call sequence patterns. This is more probable especially when the target application has such a vulnerable structure as described above. In contrast to sequence-based techniques, our SCFD method can easily detect abnormal deviations in high-level, naturally variable execution

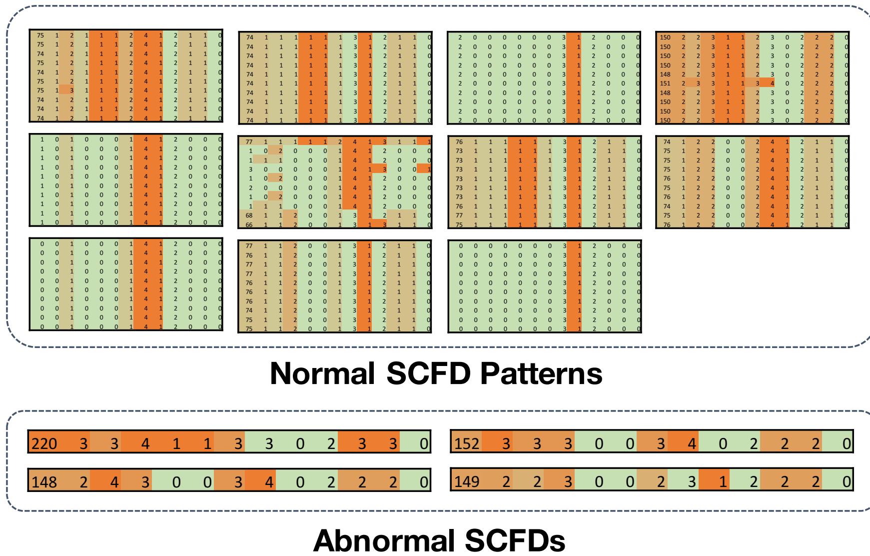


Figure 4.3: The normal SCFD patterns (top) obtained from `Motion` in normal executions and abnormal SCFD examples (bottom) due to the execution of the malicious block in Figure 4.2.

contexts such as the one illustrated above (Figure 4.2) since the SCFD significantly changes due to the malicious execution; the SCFDs at the bottom in Figure 4.3 are clearly abnormal when compared to each of the normal patterns for `Motion` (the 11 patterns shown in the top of the figure). Also, if the attacker corrupts the integrity of the data (for instance, erases the motion frame so that no motion can be detected) then our method is able to detect it – this is not easy for sequence-based methods as we explain in Section 4.5.1. Hence, by using these two approaches together, one can improve the overall accuracy of the system call-based anomaly detection.

4.2.2 Adversary Model

We consider threat models that involve changes to the behavior of system call usage. If an attack does not invoke or change any system calls, the activity at least has to affect executions afterward so that the future system call usage may change. The methods in this work, as they stand, cannot detect attacks that never alter system call usage and that just replace certain system calls by *hijacking* them (e.g., altering kernel system call table) [13]. These attacks cannot be detected by any system call pattern monitoring methods.

We especially consider stealthy, indirect attacks, e.g., ones that collect important system information or leak out sensitive data while the system/application is functioning normally; or attacks that degrade the availability of such systems. The attacker may have installed the target program in the system or induced users to download the modified source code or the executable binary using, for example, a social engineering tactic. We do not focus on more active attacks such as process killing, privilege escalation, etc., as these will change the system call usage in an obvious way.

4.2.3 Assumptions

The following assumptions are made in this work:

- We consider applications that execute in a *repetitive fashion* which fits well for embedded applications (e.g., sensing and computation). `Motion`, used in our prototype and evaluation, is an example. We monitor and perform a legitimacy test at the end of each invocation of a task.
- We limit ourselves to applications where most of the possible execution contexts can be profiled ahead of time. Hence, the behavior model is learned under the *stationarity* assumption – this is a general requirement of most behavior-based anomaly detection systems [56]. This can be justified by the fact that most embedded applications have a limited set of execution modes and input data falls within fairly narrow ranges. Also, a significant amount of analysis of embedded systems is carried out post-design/implementation anyways [64] for a variety of reasons. Hence, the information about the usage of system calls can be rolled into such a-priori analysis. Our method may not work well for applications that do not exhibit execution regularities but such systems are not the focus of our work anyways.
- The profiling is carried out prior to system deployment when the application is trustworthy. Also, any updates to the applications or the system must be accompanied by a repeat of the profiling process. We assume that the stored profile cannot be tampered with (for example, by hardware-based protections [153, 162, 156]). As mentioned earlier, these assumptions must hold for any behavior-based monitoring/detection mechanisms.

As mentioned above, we assume that an abnormal execution will exhibit a different pattern of system call usage. For example, an execution that leaks out a sensitive information would make use of network-related system calls (e.g., `socket`, `connect`, `write`, etc.) thus changing the frequencies of these calls.

4.3 Anomaly Detection Using Execution Contexts Learned from System Call Distributions

We now present our novel method to detect abnormal execution contexts in embedded applications by monitoring changes in system call frequency distributions.

4.3.1 Definitions

Let $\mathcal{S} = \{s_1, s_2, \ldots, s_D\}$ be the set of all system calls provided by an operating system, where s_d represents the system call of type d. During the nth execution of an application, it calls a multiset σ^n of $\mathcal{S}$. Let us denote the nth

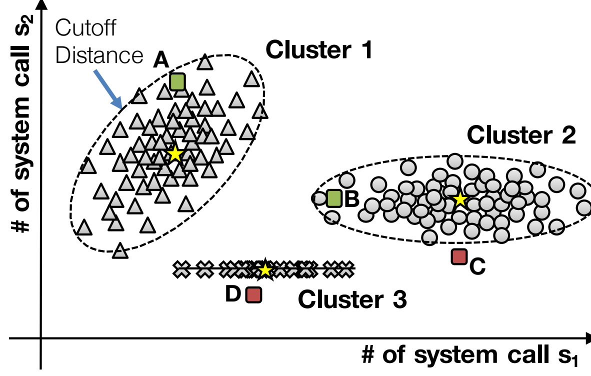


Figure 4.4: System call frequency distributions for $\mathcal{S} = \{s_1, s_2\}$ and clusters. The gray-colored objects are SCFDs in the training set. Each star-shaped point inside each cluster is its centroid. The ellipsoid around each cluster draws the cutoff line of the cluster; the points inside of the line are legitimate with respect to the cluster.

system call frequency distribution (or just *system call distribution*) as $\mathbf{x}^n = [m(\sigma^n, s_1), m(\sigma^n, s_2), \ldots, m(\sigma^n, s_D)]^T$, where $m(\sigma^n, s_d)$ is the multiplicity of the system call of type d in σ^n. Hereafter, we simplify $m(\sigma^n, s_d)$ as x_d^n. Thus, $\mathbf{x}^n = [x_1^n, x_2^n, \ldots, x_D^n]^T$.

We define a *training set*, i.e., the execution profiles of a sanitized system, as a set of N system call frequency distributions collected from N executions, and is denoted by $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N]^T$. The clustering algorithm (Section 4.3.3) then maps each $\mathbf{x}^n \in \mathbb{N}^D$ to a cluster $c_i \in \mathcal{C} = \{c_1, c_2, \ldots, c_k\}$. We denote by $c : \{\mathbf{x}^1, \cdots, \mathbf{x}^N\} \rightarrow \mathcal{C}$ the cluster that $\mathbf{x}^n \in \mathbf{X}$ belongs to.

4.3.2 Learning Single Execution Context

The variations in the usage of system calls will be limited if the application under monitoring has a simple execution context. In such a case, it is reasonable to consider that the executions follow a certain distribution of system call frequencies, clustered around a *centroid*, and cause a small variation from it due to, for example, input data or execution flow. This is a valid model for many embedded systems since the code in such system tends to be fairly limited in what it can do. Hence, such analysis is quite powerful in detecting variations and thus catching anomalies.

For a multivariate distribution, the mean vector $\mu = [\mu_1, \mu_2, \ldots, \mu_D]^T$, where $\mu_d = (\sum_n^N x_d^n)/N$, can be used as the centroid. Figure 4.4 plots the frequency distributions of two system call types (i.e., $D = 2$). For now, let us consider only the data points (triangles) on the left-hand side of the figure. The data points are clustered around the star-shaped marker that indicates the centroid of the distribution formed by the points. Now, given a new observation from the monitoring phase, e.g., the point marked 'A', a *legitimacy test* can be devised that tests *the likelihood* that such an observation is actually part of the expected execution context. This can be done by measuring *how far* the new observation is from the centroid. Here, the key consideration is on the *distance* measure for testing legitimacy.

One may use the Euclidean distance between the new observation $\mathbf{x}^*$ and the mean vector of a cluster, i.e., $||\mathbf{x}^* - \boldsymbol{\mu}|| = \sqrt{(\mathbf{x}^* - \boldsymbol{\mu})^T(\mathbf{x}^* - \boldsymbol{\mu})}$. Although the Euclidean distance (or L^2-norm) is simple and straightforward to use, the distance is built on a strong assumption that each coordinate (dimension) contributes *equally* while computing the distance. In other words, the same amount of differences in x_1^n and x_2^n are considered equivalent even if, e.g., a small variation in the usage of system call s_2 is the stronger indicator of abnormality than system call s_1. Thus, it is more desirable to allow such a variable contribute more. For this reason, we use the *Mahalanobis* distance [99], defined as $\sqrt{(\mathbf{x}^n - \mu)^T \Sigma^{-1} (\mathbf{x}^n - \mu)}$, for a group of data set $\mathbf{X}$, where Σ is the covariance matrix of $\mathbf{X}$.[1] Notice that the existence of Σ^{-1} is the necessary condition to define the Mahalanobis distance; i.e., the difference of the frequency of each system call from the mean (i.e., what is expected) is augmented by the *inverse of its variance*.

Accordingly, if we observe a small variance for certain system calls during the training, e.g., `execve` or `socket`, we would expect to see a similar, small, variation in the usage of the system calls during actual executions as well. On the other hand, if the variance of a certain system call type is large, e.g., `read` or `write`, the Mahalanobis distance metric gives a small weight to it in order to keep the distance (i.e., abnormality) less sensitive to changes in such system calls. Cluster 2 in Figure 4.4 shows an example of the advantage of using the Mahalanobis distance over the Euclidean distance. Although C is closer to the centroid than B is in terms of the Euclidean distance, it is more reasonable to determine that C is an outlier and B is legitimate because we have not seen (during the normal executions) frequency distributions such as the one exhibited by C while we have seen a statistically meaningful amount of examples like B. As an extreme case, let us consider D which is quite close to Cluster 3's center in terms of the Euclidean distance. However, it should be considered malicious because s_2 (i.e., the y-axis) should never vary.

Using covariance values also make it possible to learn *dependencies* among different system call types. For instance, an occurrence of the `socket` call usually accompanies `open` and many `read` or `write` calls. Thus, we can easily expect that changes in `socket`'s frequency would also lead to variations in the frequencies of `open`, `read` and `write`. Cluster 1 in Figure 4.4 is such an example that shows covariance between the two system call types. On the other hand, they are independent in Cluster 2 and 3. Thus, using the Mahalanobis distance we can not only learn how many occurrences of each individual system call should exist but also how they should vary together.

Now, given a set of system call distributions, $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N]^T$, we calculate the mean vector, μ, and the covariance matrix, Σ, for this data set. It then can be represented as a single cluster, c, whose centroid is defined as (μ, Σ). Now, the Mahalanobis distance of a newly observed SCFD, $\mathbf{x}^*$, from the centroid is

$$dist(\mathbf{x}^*, c) = \sqrt{(\mathbf{x}^* - \mu)^T \Sigma^{-1} (\mathbf{x}^* - \mu)}. \tag{4.1}$$

[1] Σ is the positive definite. If we set $\Sigma = \mathbf{I}$, the Mahalanobis distance is equivalent to the Euclidean distance. Thus, the Mahalanobis distance is more expressive than the Euclidean distance.

If this distance is greater than a cutoff distance θ, we consider that the execution to be malicious. For example, B in Figure 4.4 is considered legitimate with respect to Cluster 2. One analytic way to derive this threshold, θ, is to think of the Mahalanobis distance with respect to the multinomial normal distribution,

$$p(\mathbf{x}^*) = \sqrt{|\Sigma|(2\pi)^D}^{-1} \exp\Big(-\frac{1}{2}dist(\mathbf{x}^*, c)^2\Big). \tag{4.2}$$

That is, we can choose a θ such that the p-value under the null hypothesis is less than a significant level p_0, e.g., 1% or 5%. In general, there is no analytic solution for calculating the cumulative distribution function (CDF) for multivariate normal distributions. However, it is possible to derive the CDF with Mahalanobis distance. The cutoff distance θ can be derived by finding the smallest distance that makes the probability that a data point $\mathbf{x}$, which in fact belongs to the cluster and has a distance farther than θ, is not greater than $p_0 = 0.01$ or 0.05. First, let z be a Mahalanobis distance from a multivariate normal distribution. Then,

$$\int_0^\theta c \cdot e^{-\frac{1}{2}z^2} dz = 1 - p_0, \tag{4.3}$$

where c is a normalizing constant that satisfies Eq. (4.3) with $\theta = \infty$ and $p_0 = 0$ by the definition of a probability density function. This results in $c = 1/1.25331$ because

$$\int_0^\infty e^{-\frac{1}{2}z^2} dz \backsimeq \Big[1.25331 \cdot \mathrm{erf}(0.707107 \cdot z)\Big]_0^\infty = 1.25331,$$

where $\mathrm{erf}(z)$ is the *error function* and is 1 and 0 for $z = \infty$ and $z = 0$, respectively. Accordingly, Eq. (4.3) becomes

$$\begin{aligned} \frac{1}{1.25331} \int_0^\theta e^{-\frac{1}{2}z^2} dz &\backsimeq \frac{1}{1.25331} \Big[1.25331 \cdot \mathrm{erf}(0.707107 \cdot z)\Big]_0^\theta \\ &= \mathrm{erf}(0.707107 \cdot \theta) = 1 - p_0. \end{aligned}$$

Therefore, the cutoff distance θ for a significant level p_0 is

$$\theta = \frac{\mathrm{erf}^{-1}(1 - p_0)}{0.707107}. \tag{4.4}$$

For $p_0 = 1\%$ and 5%, $\theta \sim 2.57583$ and 1.95996, respectively. Figure 4.5 shows the cutoff distance for $0\% \leq p_0 \leq 100\%$. The cutoff distance is not bounded (i.e., $\theta = \infty$) when $p_0 = 0\%$ and is 0 when $p_0 = 100\%$.

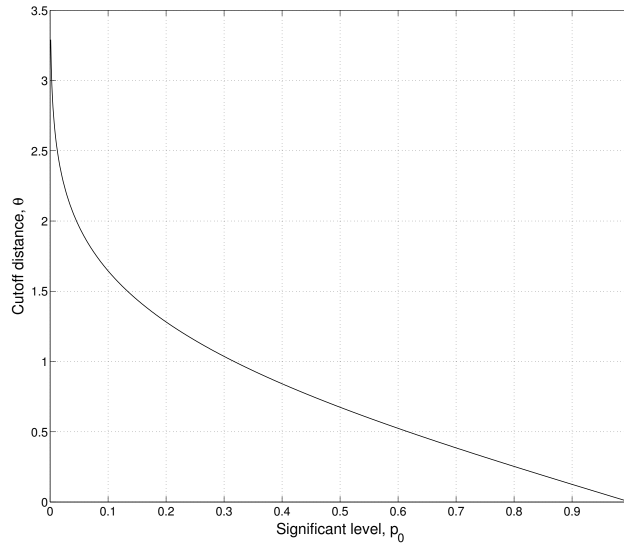


Figure 4.5: The cutoff distance θ for significant level p_0.

4.3.3 Learning Multiple Execution Contexts

In general, an application may show widely varying system call distributions due to multiple execution modes and varying inputs. In such scenarios, finding a single cluster/centroid for the whole set can result in inaccurate models because it would include many non-legitimate points that belong to none of the execution contexts – i.e., the empty space between clusters in Figure 4.4. Thus, it is more desirable to consider that observations are generated from a set of *distinct* distributions, each of which corresponds to one or more execution contexts. Then, the legitimacy test for a new observation $\mathbf{x}^*$ is reduced to identifying the *most probable* cluster that may have generated $\mathbf{x}^*$. If there is no strong evidence that $\mathbf{x}^*$ is a result of an execution corresponding to any cluster then we determine that $\mathbf{x}^*$ is most likely due to malicious execution.

Suppose we collect a training set $\mathbf{X} = [\mathbf{x}^1, \mathbf{x}^2, \ldots, \mathbf{x}^N]^T$ where $\mathbf{x}^n \in \mathbb{N}^D$. To learn the distinct distributions, we use the k-means algorithm [96] to partition the N data points on a D-dimensional space into k clusters. The k-means algorithm works as follows:

1. Initialization: Create k initial clusters by picking k random data points from $\mathbf{X}$.

2. Assignment: For each $\mathbf{x}^n \in \mathbf{X}$, assign it to the closest cluster $c(\mathbf{x}^n)$, i.e.,

$$c(\mathbf{x}^n) = \arg\min_{c_k \in \mathcal{C}} dist(\mathbf{x}^n, c_k). \tag{4.5}$$

3. Update: Re-compute the centroid (i.e., μ and Σ) of each cluster based on the new assignments.

Algorithm 1 GLOBAL K-MEANS($\mathbf{X}, \mathtt{MAX_K}, \mathtt{Bound_{TD}}$)

```
1: {X: the training set}
2: {MAX_K: the maximum number of clusters}
3: {Bound_TD: the total distance bound}
4: Create c_1 with X. Calculate μ_1 and Σ_1.
5: C ← {c_1}, k ← 2, Min_TD ← ∞
6: while k ≤ MAX_K or total-dist(X, C) > Bound_TD do
7:    for n = 1, ..., N do
8:       Create c_k with x^n as its initial point.
9:       C' ← k-means(X, C ∪ c_k)
10:      if total-dist(X, C') < Min_TD then
11:         {Note: The best clustering for k so far}
12:         C* ← C'
13:         Min_TD ← total-dist(X, C')
14:      end if
15:   end for
16:   C ← C*
17:   k ← k + 1
18: end while
19: return C
```

The algorithm repeats steps 2 and 3 until the assignments stop changing. Intuitively speaking, the algorithm keeps updating the k centroids until the total distance of each point $\mathbf{x}^n$ to its cluster,

$$total\text{-}dist(\mathbf{X}, \mathcal{C}) = \sum_{n=1}^{N} dist(\mathbf{x}^n, c(\mathbf{x}^n)), \tag{4.6}$$

is minimized.

The k-means algorithm requires a strong assumption that we already know k, the number of clusters. However, this assumption does not hold in reality because the number of distinct execution contexts is not known ahead of time. Moreover, the accuracy of the final model heavily depends on the initial clusters chosen randomly.[2] Hence, we use the *global k-means* method [94] to find the number of clusters as well as the initial assignments that lead to *deterministic accuracy*. Algorithm 1 illustrates the global k-means algorithm. Given a training set $\mathbf{X}$ of N system call frequency distributions, the algorithm finds the best number of clusters and assignments. This is an incremental learning algorithm that starts from a single cluster, c_1, consisting of the entire data set. In the case of $k = 2$, the algorithm considers each $\mathbf{x}^n \in \mathbf{X}$ as the initial point for c_2 and runs the assignment and updates steps of k-means algorithm. After N trials, we select the final centroids that resulted in the smallest total distance calculated by Eq. (4.6). These two centroids are then used as the initial points for the two clusters, respectively, in the case of $k = 3$. This procedure repeats until either k reaches a pre-defined $\mathtt{MAX_K}$, the maximum number of clusters, or the total distance value becomes less than the total distance bound $\mathtt{Bound_{TD}}$. Note that the total distance in Eq. (4.6) decreases

[2]Finding the optimal assignment in the k-means algorithm with the Euclidean distance is NP-hard. Thus, finding the optimal assignments with the Mahalanobis distance is at least NP-hard because the Mahalanobis distance is more general than the Euclidean distance.

monotonically with the number of clusters. For example, if every point is its own cluster then the total distance is zero since each point itself is the centroid.

The original global k-means algorithm assumes the Euclidean distance. As explained above, we use the Mahalanobis distance as in Eq. (4.1). Meanwhile, k-means$(\mathbf{X}, \mathcal{C})$ (line 9) is the standard k-means algorithm without the random initialization; it assigns the points in $\mathbf{X}$ to a $c_k \in \mathcal{C}$, update the centroids, repeats until stops, and then returns the clusters with the *updated* centroids. The standard k-means algorithm uses the Euclidean distance and thus the centroids of the initial clusters are the data points that were picked first. Remember, however, that the Mahalanobis distance requires a covariance matrix. Since there would be only one data point in each initial cluster we use the global covariance matrix of the entire data set $\mathbf{X}$ for the initial clusters. After the first iteration, however, the covariance matrix of each cluster is updated using the data points assigned to it.

The clustering algorithm finally assigns each data point in the training set into a cluster. Then, each cluster $c_i \in \mathcal{C}$ can be represented by the centroid, (μ_i, Σ_i), that now makes it possible to calculate the Mahalanobis distance of a newly observed SCFD $\mathbf{x}^*$ to each cluster using Eq. (4.1). The legitimacy test of $\mathbf{x}^*$ is then performed by finding the closest cluster, c^*, using Eq. (4.5). Thus, if

$$dist(\mathbf{x}^*, c^*) = \min_{c_i \in \mathcal{C}} dist(\mathbf{x}^*, c_i) > \theta$$

for a given threshold θ, we determine that the execution does not fall into any of the execution contexts specified by the clusters since $dist(\mathbf{x}^*, c_i) > \theta$ for all $i = 1, \ldots, k$. We then consider the execution to be malicious. As an example, for the new observation C in Figure 4.4, Cluster 2 is the closest one and C is outside its cutoff distance. Thus, we consider that C is malicious. Note that, as shown in the figure, the same cutoff distance defines different ellipsoids for different clusters; each ellipsoid is a equidistant line from the mean vector measured in terms of the *Mahalanobis* distance. Thus, a cluster with small variances (i.e., less varying execution context) would have a smaller ellipsoid in the Euclidean space.

4.3.4 Reduced SCFD

The number of system call types, i.e., D, is quite large in general. Thus, the matrix calculations in Eq. (4.1) might result in an unacceptable amount of analysis overhead.[3] However, embedded applications normally use a limited subset of system calls. Furthermore, we can significantly reduce the dimensionality by ignoring system call types that never vary. Consider Cluster 3 from Figure 4.4. Here, x_2 can be ignored since we can reasonably expect it to never vary

[3]Note that μ and Σ values are calculated from the clustering algorithm which is an *offline* analysis. Thus we store Σ^{-1} for computational efficiency.

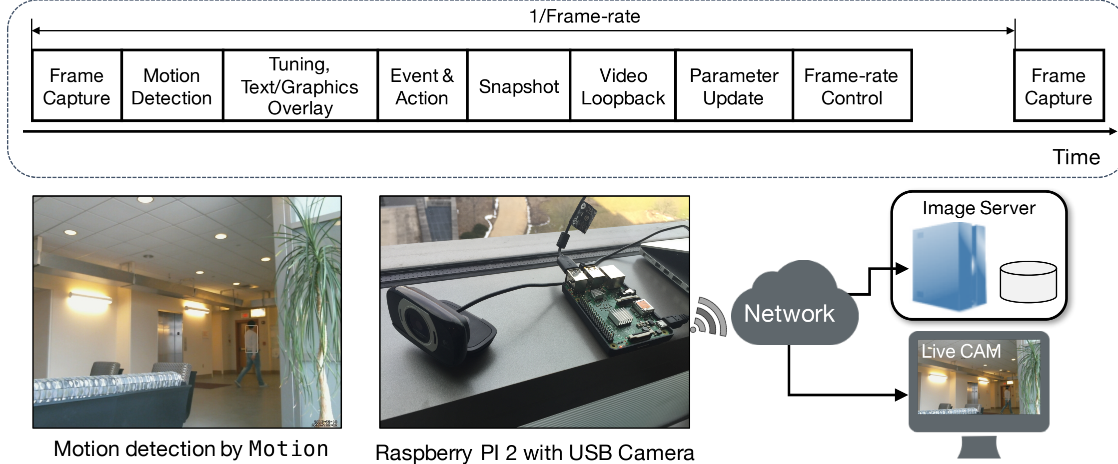


Figure 4.6: `Motion`'s main execution process (top). The main loop repeats at the frame rate (e.g., 3 frames per second). Some of the blocks execute only when certain event occurs and hence they may not appear in every loop.

during the normal execution.[4] Thus, before running the clustering algorithm, we reduce $\mathcal{S}$ to $\mathcal{S}' = \{s_{d_1}, s_{d_2}, \ldots, s_{D'}\}$, where $D' \leq D$, such that the variance of x_d for each $s_d \in \mathcal{S}'$ is non-zero in the entire training set $\mathbf{X}$. However, we should still be able to detect any changes in such system calls that never varied (including those that never appeared). Thus, we merge all such x_d in $\mathcal{S} - \mathcal{S}'$; the sum should not change in normal executions. In case D' is still large, one may apply a statistical dimensionality reduction technique such as Principal Component Analysis (PCA) [84].

4.4 Evaluation Framework

In this section, we present the implementation details for our prototype.

4.4.1 Target Application

We use `Motion` [18], an open-source program that monitors images captured from a camera and detects motion by tracking changes between image frames as illustrated in Figure 4.6. It is often used for surveillance purpose and provides live streaming and external program execution when certain events (e.g., motion detection, on file creation, etc.) are detected.

Figure 4.6 also shows `Motion`'s main execution process. The main loop consists of a series of blocks. Each loop starts by capturing an image frame from the camera using the Video4Linux (V4L) interface. Next, motion detection algorithm looks for changes from the previous frames. When a change (i.e., motion) is detected, each frame is saved to the filesystem. Following this, some pre-defined event actions could trigger external programs (such as

[4] In fact, s_2 cannot be ignored in the example depicted in Figure 4.4 since its variance is non-zero in clusters 1 and 2.

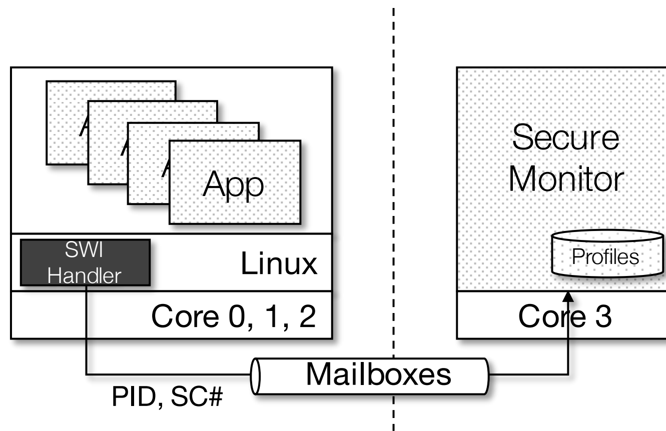


Figure 4.7: System call monitoring framework implemented on Broadcom BCM2836 SoC. System calls made by applications are captured by the software interrupt handler in Linux and are reported to Core 3 through a set of mailboxes.

executing a script file, uploading image to a remote image server, etc.). This main loop repeats at the specified frame rate (such as 3 frames per second). Depending on the events, some of the blocks may not execute in every loop. In our configuration, a python script that logs the current time in a file executes (by `on_motion_detected` event handler in `Motion`) when a motion is detected, and the `wput` Linux command is executed to upload the newly created images (by `on_picture_save` event handler) to a remote server. These external commands are executed by separate processes forked by the main process.

4.4.2 System Implementation

We implemented a prototype of our SCFD-based anomaly detection system on a Raspberry PI 2 Model B board [20]. It has a quad-core ARM Cortex-A7 CPU. Each core runs at 900 MHz. The system has a memory of 1 GB and runs Linux 3.18.

Figure 4.7 shows our system call monitoring framework implemented on Broadcom BCM2836 SoC (System-on-Chip) [6] on the Raspberry PI 2 board. We inserted a hook in the software interrupt (SWI) handler that dispatches each system call handler. The hook sends the system call number and the PID (Process ID) of the caller to the monitoring process (called Secure Monitor) on Core 3 through a set of *mailboxes*. Each mailbox on BCM2836 SoC is a 32-bit wide core-to-core communication channel. The secure monitor performs the detection process presented in Section 4.3 using the SCFD built from the reported information.[5] We implemented the secure monitor as a baremetal application for the purposes of our proof-of-concept. We created a Linux kernel module that tells (through another mailbox) the secure monitor what processes to monitor.

[5] We created a custom system call that indicates an execution boundary. We inserted a special system call at the end of `Motion`'s main loop. The secure monitor recognizes this system call as the end of one execution.

Since we collect system call usage information at the operating system layer (i.e., software interrupt handler), the OS is our trusted computing base (TCB). Note that we could either run the secure monitor inside or on top of the OS as done by most system call monitoring/auditing modules [23, 110]. One can add more security by utilizing a hardware-supported partitioning mechanism, e.g., the ARM TrustZone [153], and protecting the security monitor even if the main system is compromised.

4.4.3 Attack Scenarios

Considering the purpose and the functionality of `Motion`, the primary security concerns are *privacy* and *availability*. Hence, we consider the following attack scenarios:

- Attack 1: One attack is the leaking of images captured by `Motion` while leaving the original functionality intact. We consider the case where an attacker saves the images at a desired location in the filesystem with the intention that the collection will be used/retrieved later.
- Attack 2: The attacker corrupts the images captured from the camera so that no motion can be detected. Specifically, the attacker erases frame(s) by calling `memset`. Note that this attack does not require any system calls.

The attacker tries to implement the above attacks as simply as possible (e.g., using existing routines/libraries) because otherwise the system call usage will diverge in an obvious way. Note that we consider only the cases that change the system call usage of `Motion` directly or indirectly. If the attacker had higher privileges in the system, then he could perform more active attacks such as killing the `Motion` process, copying the legitimately-saved images out of the device, deleting files, disabling the camera, etc. Such attempts can be detected by other techniques. Also, we do not make any assumptions as to how the compromised program is present on the device. The modified program may have already been installed or the user may have downloaded the modified (open) source code or the executable binary.

4.5 Evaluation Results

We now evaluate the SCFD method on the prototype described in the previous section. We obtained a training set that consists of 2420 loop executions of `Motion` that ran under normal conditions (i.e., no attack present) for about 15 minutes. `Motion` used total 15 types of system calls.

```
Motion loop {
    gettimeofday-gettimeofday
    (ioctl)-rt_sigprocmask-ioctl-ioctl-rt_sigprocmask   /* Frame Capture */
    If (motion_detected) {   /* Run external command upon 'on_motion_detected' event*/
        clone
        open-fstat64-mmap2-write- ... -write-close-munmap-clone-write  /* Save frame image*/
        write
    }                                        /* write chain. Length depends on image size. */
    If (time to take a snapshot) {
        open-fstat64-mmap2-write- ... -write-close-munmap-clone-write  /* Save snapshot image*/
        unlink-symlink   /* Update symbolic link to the latest snapshot file*/
    }
    select     /* Wait for a webcam-client */
    If (a webcam-client is waiting) {
        /* Several variations are made with the these calls*/
        (accept-ioctl-write)-(write-munmap-close)-(mmap2-gettimeofday)-(write)-(write)-(munmap)
    }
    gettimeofday-(nanosleep)   /* Frame-rate Control */
}
```

Figure 4.8: The system call sequences made by `Motion` in each loop. Total 15 different types of system calls are used. The three `if` blocks can independently execute and thus create various execution contexts. Further variations are made by the `write` chain when saving images to files. The parenthesized calls (e.g., `(ioctl)`) may sometime present/skip.

4.5.1 Sequence-based Security Analysis

We first show that it is feasible to implement the attack scenarios described in Section 4.4.3 while avoiding detection from sequence-based approaches.

Figure 4.8 summarizes the system call sequences made in each loop in normal situations.[6] Each loop always starts by storing the current times (for time keeping) and taking the current image frame from the camera. Then, if motion is detected, a separate process is forked to execute an external command upon the `on_motion_detected` event and the image frame is saved as a file. If the time to take a snapshot arrives (e.g., once every 5 seconds, for archiving purposes) the resulting image is saved as a file. Next, the loop feeds an image to a webcam-client if any are viewing. This `if` block generates several variations that cannot be represented by a single sequence. The loop then ends with the frame-rate control. The shortest sequence is when all the `if` conditions are false.

The three `if` blocks are independent; each loop may execute only one, a pair, or all of them depending on the current situation. This *creates various execution contexts*. The system call usages can vary further when images are saved to files, as can be seen from the first two `if` blocks. This is due to the varying length of the `write` chain that depends on the image size.

[6] The sequence shown in the figure is the most representative one. In some loops, for example, the first `ioctl` in the second line may skip, and `nanosleep` in the last line may not be called if there is no need to insert a delay to meet the next frame time.

```
If (motion_detected) {
      clone
  (A) open-fstat64-mmap2-write- ... -write-close-munmap-clone-write
      write
  (B) open-fstat64-mmap2-write- ... -write-close-munmap-clone-write
      write
}                                              Inserted by attacker

If (time to take a snapshot) {
  (C) open-fstat64-mmap2-write- ... -write-close-munmap-clone-write
      unlink-symlink
}
```

Figure 4.9: Attacker can insert a simple piece of code that uses the same routine as the legitimate code calls to save an image data to the filesystem. If the code is inserted as depicted, the resulting system call sequences would look normal.

We now explain how the attacks can be carried out while avoiding detection from sequence-based approaches.

Attack 1: Notice, from Figure 4.8, the system call sequence that is made when writing to files:

`open-fstat64-mmap2-write-···-write-close-munmap-clone-write`

It is generated by a common routine, `put_picture`, in `Motion`. Hence, the attacker can use the very same function to save the current frame image at a desired location by inserting the following small piece of code:

```
const char* org_filepath = cnt->conf.filepath;
cnt->conf.filepath = "/path/to/attackers_desired_location";
event(cnt, EVENT_IMAGE_DETECTED, cnt->imgs.image_ring[cnt->imgs.image_ring_out].image,
       NULL, NULL, &cnt->imgs.image_ring[cnt->imgs.image_ring_out].timestamp_tm);
cnt->conf.filepath = org_filepath;
```

The `event` function above is identical to what is called by the original code (in the first two `if` blocks in Figure 4.8) and it in turn calls the `put_picture` library routine. The attacker only needs to change the path to store the image (i.e., `cnt->conf.filepath`) in the configuration and restore it back before and after calling the `event` function, respectively, as depicted.[7] Now, the attacker can place this piece of code (followed by a bogus `write` call) between the two file write operations, as depicted in Figure 4.9. The figure shows the case when the attacker wants to steal every motion frame. The attacker can instead steal snapshots by inserting the code at the beginning of the second `if` block (i.e., before C).

[7] In the same way, the attacker can even execute an arbitrary external command too by changing the configuration parameter that stores the string of external command that will execute upon image file creation. Hence, the attacker can upload the leaked image to a remote server by setting `Motion` parameter to externally execute `wput` Linux command.

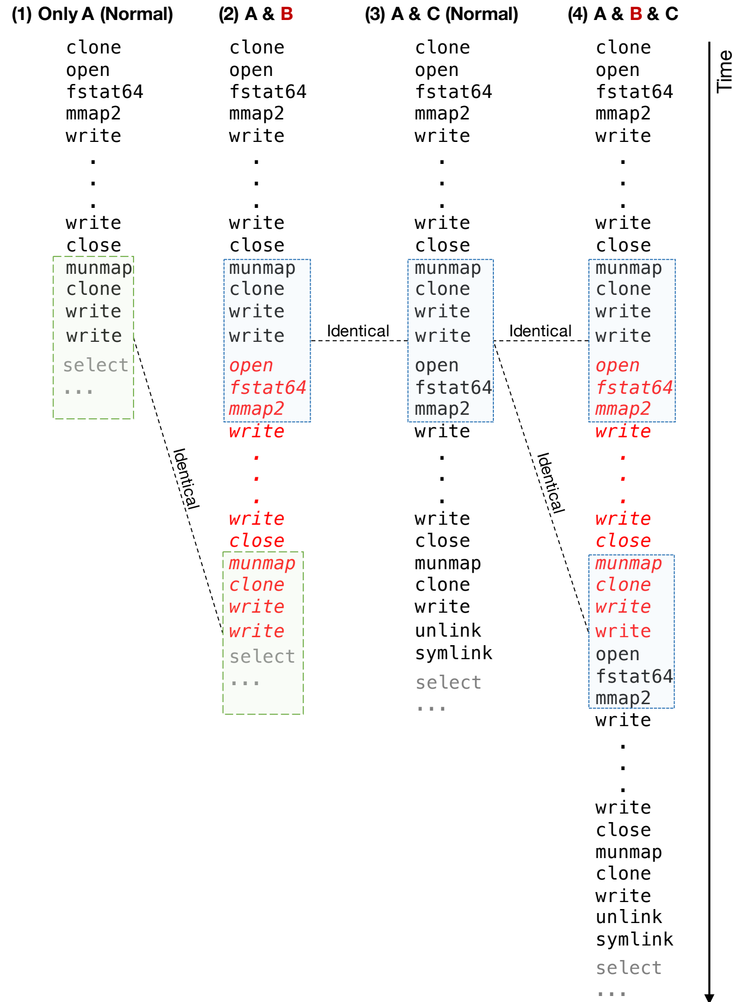


Figure 4.10: Sequence-based approaches are fundamentally limited to catch the execution of the attacker's inserted code in Figure 4.9 because the resulting sequence patterns (that are generated by (2) and (4)) cannot be differentiated from the normal patterns (that are generated by (1) and (3)).

Now, it is difficult for sequence-based approaches to catch this attack because the resulting subsequences look normal. Let us consider the following cases:

1. Only the first `if` block executes (Case (2) in Figure 4.10): First of all, any subsequences made only by the inserted code itself (marked as B in Figure Figure 4.9) are same with those made by the legitimate code (marked as A). Now, the subsequences that span the legitimate (A) and the inserted codes (B) are same with those made by the transition from A to C when the attack code was not inserted (Case (3) in Figure 4.10 which is a normal sequence). That is, the execution looks as if both frame image and snapshot are saved (i.e., both `if` blocks execute) although the second `if` block did not execute. Also, the subsequences that include the tail of B look identical to those of A when only A executes (Case (1) in Figure 4.10). Hence, they are legitimate too.

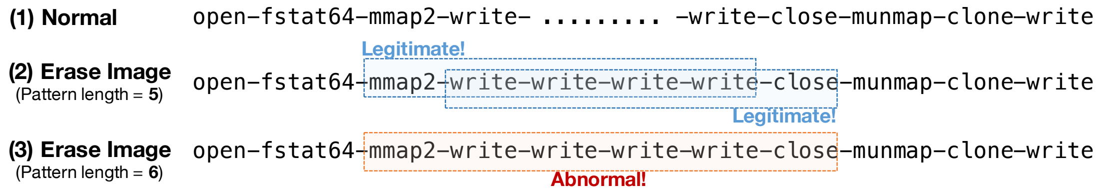


Figure 4.11: The `write` chain becomes short when frame image is erased. For a sequence-based method to detect the anomaly, the patterns should be long enough to cover the whole chain and also what leads and follows.

2. Both `if` blocks execute (Case (4) in Figure 4.10): Now suppose the inserted code executes between the two file write operations, as depicted in Case (4) of Figure 4.10. Similar to the situation above, the subsequences generated by the transitions from A to B and B to C are still legitimate with respect to the patterns that can be learned from the normal case (i.e., Case (3)). This is because what A and B generate together (that is, patterns that span across A and B) do not look different from what A and C would generate, and for the same reason, the patterns generated by B and C look normal too.

The only way any sequence-based method could detect an execution of the inserted code for **Attack 1** is to know the legitimate patterns that are long enough to learn that the second appearance of `open-fstat64-`$\cdots$ sequence should end with $\cdots$`-clone-write-unlink-symlink` as shown in Case (3) of Figure 4.10. If we were able to learn such patterns, the inserted code (i.e., B) can be detected because it ends differently (i.e., Case (4)). To do so, we need to learn the patterns much longer than the length of the whole sequence generated by a file operation. However, this is unlikely[8] because the sequences are too long and both length and patterns can vary greatly due to the varying length of the `write` chain. Hence, this kind of attack (that uses a legitimate routine) can be effective if the attacker can find such routines that are commonly used in places and that can carry out the attacker's desired operation.

Attack 2: The attacker's goal is to corrupt the image frames so that motions cannot be detected. The simplest way is to erase the buffer that contains the frame image. This can be done by inserting a single line of code that calls `memset` right after a frame is captured. Note that this does not require any system calls. Instead, this affects the execution of code segments that follow in the rest of the loop. First of all, the first `if` block in Figure 4.8 does not execute as there is no motion change. However, this behavior looks legitimate as this can happen in the normal executions. Next, the length of the `write` chain when saving the snapshot image to a file changes. This is because the erased frame produces 14 KB of green images. The reduced size of the images result in shortened `write` chain as depicted in Case (2) of Figure 4.11. A single `write` call writes 4KB of data to the file and hence the `write` chain's length becomes 4. It is significantly shorter than a normal length which is longer than 70 in general.

[8] No matter how patterns are learned. They could be learned by the fixed-length method such as N-gram [76] or variable-length such as Markovian model (VMM) [42] or *Probabilistic Suffix Tree* [140], etc.

Sequence-based methods may or may not be able to detect such anomalies, depending on the length of patterns they learn.[9] If we have learned the patterns of length at most 5, the subsequences are still legitimate with respect to the normal subsequences, as can be seen from Case (2) of Figure 4.11. On the other hand, if we have learned longer (e.g., at least 6) patterns, we can know that `mmap2-write-write-write-write` should be followed by another `write` in normal situations (as in Case (1) of the figure). The abnormal situation illustrated in Case (3) is thus malicious with respect to such normal subsequences. Note that the difficulty of capturing such an abnormal situation stems from the long chain of `write` calls in the normal execution scenarios. Hence, it is not straightforward for sequence-based methods to learn the relationship between `mmap2` and `close` especially because the `write` chain's length can vary with data as well. Note also that, if the image size got larger (say, by writing random values to image frame) instead and thus made the `write` chain longer than usual, sequence-based methods cannot detect this behavior because the only change would be that there are more subsequences that consists only of `write` that have a legitimate length.

4.5.2 SCFD Training

With the training set obtained from the system under normal conditions, we applied the SCFD learning algorithm presented in Section 4.3. Out of 15 types of system calls used by `Motion`, two types, `select` and `rt_sigprocmask`, showed zero variance. As shown in Figure 4.8, these two system calls always execute once and twice, respectively, in each loop. Hence, the algorithm first reduces the dimensionality of SCFDs to 13.

Figure 4.12 visualizes the training result obtained with settings $\mathtt{MAX}_{\mathtt{K}} = 20$ and $\mathtt{Bound}_{\mathtt{TD}} = 1000$. The table in the middle is the training set (only unique SCFDs are shown) and the ones around it are the resulting clusters. Each row represents an SCFD and the colors represent high (orange color) and low (green color) counts for each system call type. As can be seen from the result, the 2420 SCFDs are clustered into 11 clusters. Table 4.1 summarizes the results by providing the mean and the standard deviation of the training set and those of each cluster. From observing the resulting clusters, we find the following execution contexts:

- Cluster 1 represents the case when no event occurs during a loop shown in Figure 4.8. The loop only takes the current image frame and none of the `if` blocks in Figure 4.8 execute.
- Clusters 2, 4 and 5 are also the cases when no images are saved to files, because the related system calls (e.g., `open`, `fstat64`, `close`) do not appear and also the number of `write` calls is few. The differences among the three clusters are due to the last `if` block (i.e., webcam remote view-related) in Figure 4.8 which shows several variations.

[9] Again, we do not make any assumptions on the way that the sequence patterns are learned. It could be a fixed-length or variable-length.

Table 4.1: The mean and the standard deviation of the system call frequency distributions in the entire training set and in each cluster after running the learning method.

	# pts		write	close	munmap	clone	unlink	symlink	mmap2	gettim	nanosl	ioctl	open	fstat64	accept
All	2420	Mean	28.015	0.376	0.565	0.681	0.067	0.067	0.565	3.192	0.988	2.009	0.374	0.374	0.002
		Stdev	39.070	0.525	0.620	0.955	0.249	0.249	0.614	0.394	0.148	0.095	0.524	0.524	0.046
Cluster 1	1129	Mean	0.000	0.000	0.000	0.000	0.000	0.000	0.000	3.000	1.000	2.000	0.000	0.000	0.000
		Stdev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 2	248	Mean	0.000	0.000	1.000	0.000	0.000	0.000	1.000	4.000	1.000	2.000	0.000	0.000	0.000
		Stdev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 3	601	Mean	74.767	1.000	1.000	2.000	0.000	0.000	1.000	3.000	1.000	2.000	1.000	1.000	0.000
		Stdev	0.954	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 4	75	Mean	2.000	0.000	0.000	0.000	0.000	0.000	0.000	3.000	1.000	2.000	0.000	0.000	0.000
		Stdev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 5	90	Mean	1.000	0.000	1.000	0.000	0.000	0.000	1.000	4.000	1.000	2.000	0.000	0.000	0.000
		Stdev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 6	73	Mean	74.000	1.000	1.000	1.000	1.000	1.000	1.000	3.000	1.000	2.000	1.000	1.000	0.000
		Stdev	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 7	47	Mean	40.064	0.660	1.064	1.043	0.043	0.043	1.085	3.553	0.872	2.468	0.532	0.532	0.106
		Stdev	39.447	0.556	0.697	1.010	0.202	0.202	0.453	0.497	0.334	0.499	0.540	0.540	0.308
Cluster 8	71	Mean	74.972	1.000	2.000	2.000	0.000	0.000	2.000	4.000	1.000	2.000	1.000	1.000	0.000
		Stdev	0.691	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 9	48	Mean	149.604	2.000	2.083	3.000	1.000	1.000	2.083	3.083	0.000	2.000	2.000	2.000	0.000
		Stdev	1.303	0.000	0.276	0.000	0.000	0.000	0.276	0.276	0.000	0.000	0.000	0.000	0.000
Cluster 10	13	Mean	74.615	1.000	1.000	1.000	1.000	1.000	1.000	3.000	1.000	2.000	1.000	1.000	0.000
		Stdev	1.546	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Cluster 11	25	Mean	74.440	1.000	2.040	1.000	1.000	1.000	2.000	4.000	1.000	2.000	1.000	1.000	0.000
		Stdev	0.496	0.000	0.196	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000

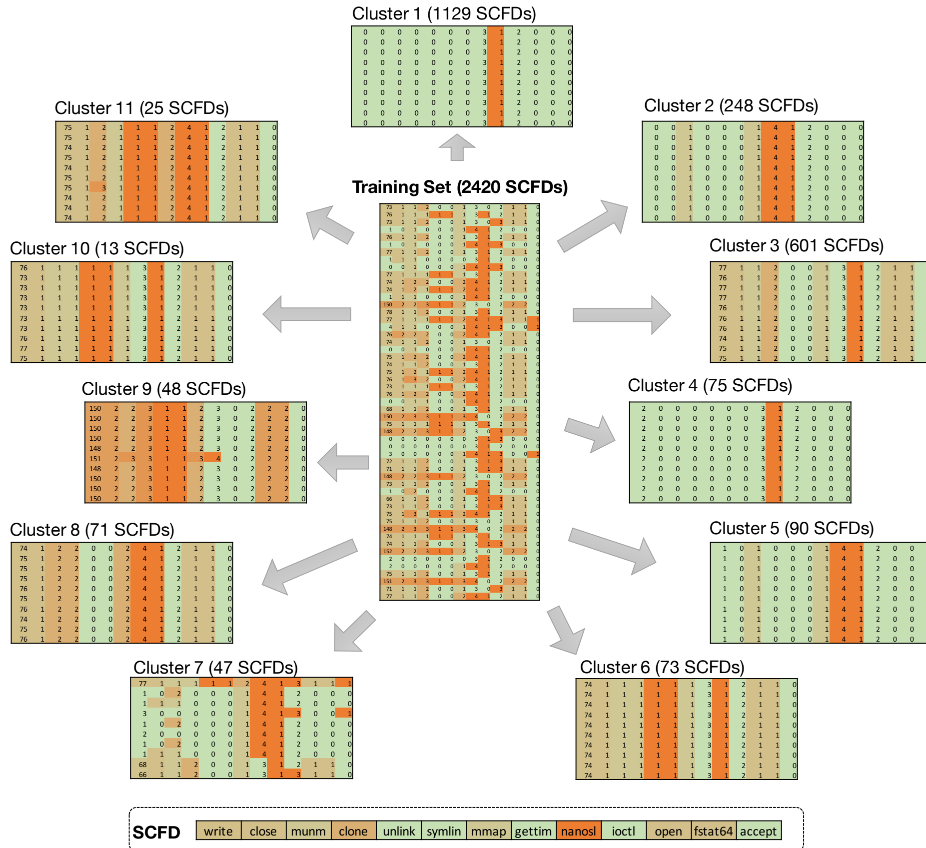


Figure 4.12: The result of clustering 2420 SCFDs. Each row represents an SCFD of 13 system call types – (from left to right) `write`, `close`, `munmap`, `clone`, `unlink`, `symlink`, `mmap2`, `gettimeofday`, `nanosl`, `ioctl`, `open`, `fstat64`, `accept`. The training set shows only the unique SCFDs and each cluster shows only 10 SCFD examples that belong to it.

- Clusters 3, 6, 8, 10, and 11 correspond to the executions that write an image file once, because the file-related system calls appear just once per SCFD (i.e., per loop). In addition, the `write` calls are used accordingly. Among them, Clusters 6, 10, and 11 write snapshot images (i.e., the second `if` block in Figure 4.8). Cluster 11 is when an image is fed to a webcam-client as more `mmap` and `unmap` are observed. The only difference between Clusters 6 and 10 is the number of `write` calls; it is fixed to 74 in Cluster 6, while Cluster 10 has everything but 74.

  The fewer number of `unlink` and `symlink` in Clusters 3 and 8 (than 6, 10, and 11) suggest that these two correspond to the executions that write frame images (i.e., the first `if` block). Also, `clone` should be called twice in that case.

- Cluster 9 corresponds to the case when both, the motion frame and snapshot files are saved (because of the reasons explained above). This cluster covers both the cases when an image is fed or not fed to webcam-client. Increasing the number of clusters will split the two cases.

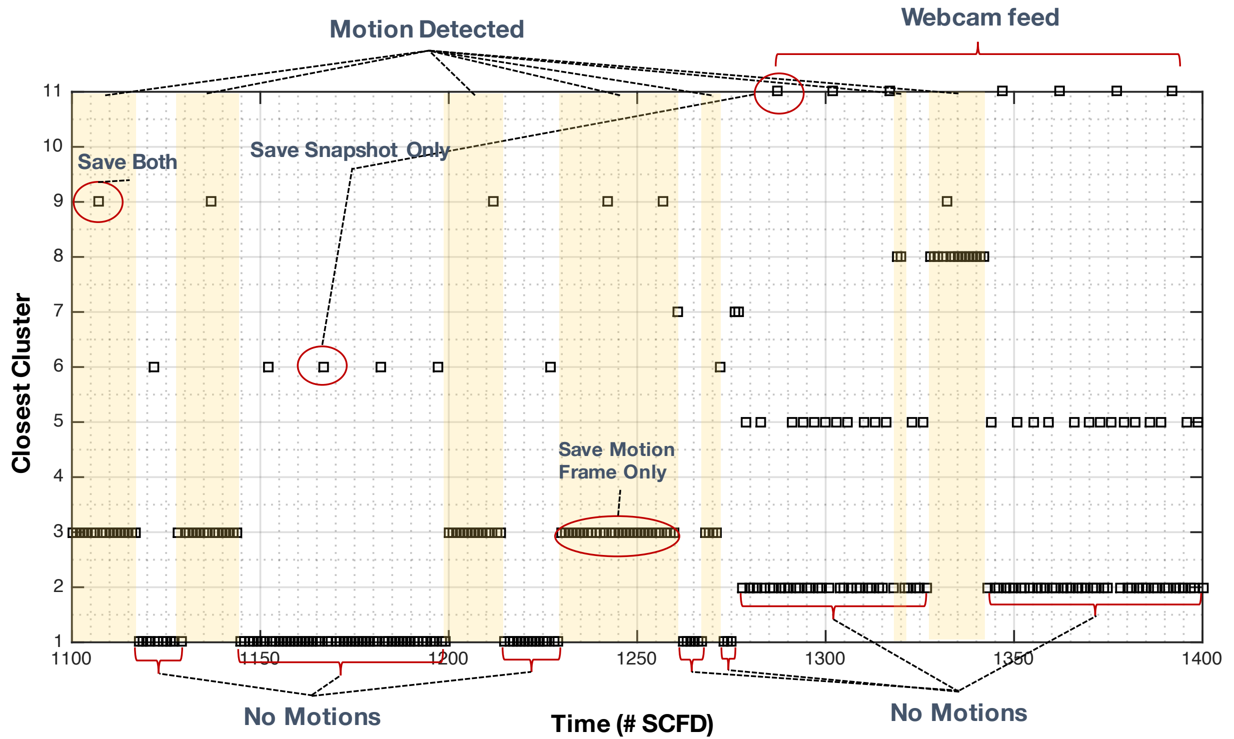


Figure 4.13: SCFDs in normal situation, their closest clusters assigned by our detection algorithm, and the corresponding execution contexts.

- Cluster 7 is a mixture of some rare SCFDs that are similar to other clusters but vary in a very small way (due to the last if block in Figure 4.8). Such differences caused them to stand out in comparison to other clusters. Each one was also not representative enough to create its own cluster. For example, only 4 out of 2420 SCFDs in the training set had a non-zero count of `accept` and these are assigned to Cluster 7. This cluster can be split into smaller ones if we increased the number of clusters.

Figure 4.13 shows the closest cluster for each SCFD (for 300 SCFDs obtained during a normal situation) and the corresponding execution context. The shaded areas represent the time period when motion is detected – during which a frame image is saved to a file. We can also see that a snapshot is saved at regular intervals (every 5 sec) regardless of motion detection. Overall, the results show the changes in the execution contexts as various events occur individually or together.

4.5.3 Accuracy

Now, we evaluate the *accuracy* of our anomaly detection method. We enabled each of the attacks from Section 4.4.3. An execution should be considered abnormal if any of the following is true: (i) any system call other than the 15

observed types is detected; (ii) any system call whose variance was zero during the profile (2 out of 15 in the case above) actually exhibits variance or (iii) the distance of an observation from its closest cluster is longer than the threshold. Among these, rules (i) and (ii) were never observed because Attacks 1 re-uses the same functions from normal executions and Attack 2 makes no system calls at all.

- Attack 1: We inserted the code block that leaks out the current frame image to the attacker's desired location (code block 'B' as shown in Figure 4.9) and then obtained a test set of 1003 SCFDs. Note that not all of them include the attack because the inserted code executes only when a motion is detected. 603 out of the 1003 SCFDs correspond to the case that did not detect a motion and thus are normal.

  The rest of the SCFDs can be divided into two groups as shown in Figure 4.14. The first group (upper-right) looks very similar to the ones in Cluster 9 (in Figure 4.12) that saves both motion frame and snapshot images. If the test SCFDs were legitimate, then they should have used `unlink` and `symlink` system calls once as shown in the `Motion`'s normal system call usage in Figure 4.8. Since the test SCFDs did not use the calls, they are classified as abnormal with respect to the learned patterns. Of course, the attacker could insert bogus `unlink` and `symlink` for this particular execution context. However, then the resulting sequences are identical to those made by the normal code (when both images are saved) and no system call-based detection methods can differentiate the two cases, which does not fall into our threat model.

  The second group (at bottom-right in Figure 4.14) consists of SCFDs observed when the inserted code executes between the two legitimate file operations (see Figure 4.9). The resulting SCFDs are clearly abnormal as there are three file operations and hence too many `write` calls.

- Attack 2: This attack does not use any system calls; it just changes the values of the data (i.e., image). As explained earlier, this attack produces 14 KB of frame images, which results in shortened `write` chains as depicted in Figure 4.11. Hence, calls to `write` is much less frequent when compared to normal executions.

  The SCFDs shown above, obtained when Attack 2 is enabled, are quite similar to Cluster 6 (top) and Cluster 11 (bottom), respectively. However, these SCFDs are always classified to be abnormal because the image sizes (due to the number of `write` calls) are not typical when saving snapshot files during normal executions. The attacker could have circumvented our detection method if, for example, the frame image is just replaced with another that has a similar size as the unmodified frame images. However, again, such case is out of scope of our threat model because the system call usage does not change.

The *false positive rate* is just as important as the detection rate because frequent false alarms degrade system availability. To measure the false positive rates, we obtained a new set of SCFDs by running the system without activating any attacks and measured how many times the secure monitor classifies an execution as being abnormal.

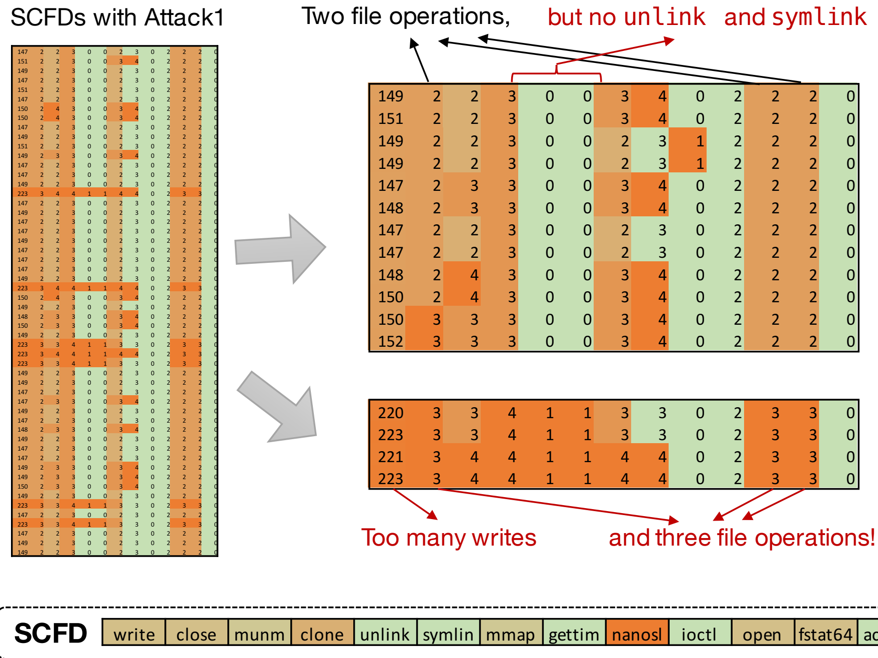
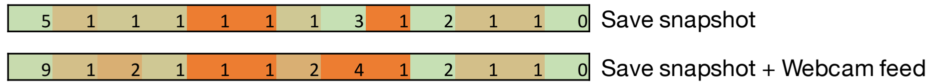


Figure 4.14: The SCFDs when the attacker leaks out current motion frames (Attack 1). They are abnormal with respect to the normal SCFDs learned in Figure 4.12. Not all SCFDs are shown.



For the cut-off distance θ with $p_0 = 5\%$, 4 out of 1755 executions (0.23%) were classified as malicious. With $p_0 = 1\%$, i.e., a farther cut-off distance, it was reduced to just 1 (0.06%). Such a lower significant level relaxes the cutoff distance and produces fewer false alarms because even some rarely-seen data points are considered normal. However, this may result in lower detection rates as well. In the attack scenarios listed above, however, the results did not change even with the lower significant level. This is a consideration for system designers to take into account when implementing our detection methods; they will have a better feel for when certain executions are normal and when some are not. Hence, they can decide to adjust values for p_0 based on the actual system(s) being monitored.

While it is true that the accuracy of the method may depend on the attacks that are launched against the system, in reality an attacker would need to not only know the exact distributions of system call frequencies but also be able to implement an attack with such a limited set of calls – both of these requirements significantly raise the difficulty levels for would-be attackers.

In Section 4.5.1, we showed that sequence-based approaches may fail to detect abnormal deviations in situations that naturally have a high-level variance in the execution contexts or use data that is very diverse. Such instances require a global view on the frequencies of different system call types made during the entire execution and the

Table 4.2: Average (standard deviation) of times to find the closest cluster given a test SCFD.

SCFD Dimension	5 Clusters	10 Clusters	15 Clusters
5	4.710 μs (0.522 μs)	8.348 μs (0.545 μs)	11.843 μs (0.373 μs)
10	11.262 μs (0.501 μs)	21.318 μs (0.470 μs)	31.474 μs (0.503 μs)
13	16.306 μs (0.463 μs)	31.582 μs (0.501 μs)	46.859 μs (0.356 μs)

correlations among different types. Sequence-based approaches are sensitive to local, temporal variations, e.g., an unusual transition from one system call to another. Our SCFD might not catch such a small, local variation. Hence, one can use these two approaches together to improve the overall accuracy of the system call-based anomaly detection for embedded systems.

4.5.4 Time Complexity

To evaluate the time complexity of the proposed detection method, we measured the times to perform the analysis. The times are measured from the moment when a new observation is given until the closest cluster is found (Eq. (4.5)). We tested for different configurations of the SCFD dimensionality and the number of clusters.[10] The statistic is based on 10000 samples per configuration collected on our prototype system.

As Table 4.2 shows, the detection process is fast. This is possible because we store Σ^{-1}, the inverse of the covariance matrix, of each cluster, not Σ. A Mahalanobis distance is calculated in $\mathcal{O}(D^2)$, where D is the number of system call types being monitored (i.e., SCFD dimensionality), since in $(\mathbf{x}^* - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\mathbf{x}^* - \boldsymbol{\mu})$, the first multiplication takes $\mathcal{O}(D^2)$ and the second one takes $\mathcal{O}(D)$. The results (top to bottom) in the table above show such a trend. Note that it would have taken $\mathcal{O}(D^3)$ if we stored the covariance matrix itself instead of its inverse; since a $D \times D$ matrix inversion takes $\mathcal{O}(D^3)$. We can also see from the results that the analysis time increases linearly with the number of clusters.

More importantly, the time complexity of our method is *independent* of how often and many times the application uses system calls; it only depends on the number of system call types being monitored. Hence, the SCFD method has a deterministic time complexity. This is determined in the training phase and does not change during the monitoring phase (see Section 4.3.4). On the other hand, the overheads of sequence-based approaches are highly dependent on the application complexity (i.e., how frequently system calls are made).

4.5.5 Limitations and Discussion

One of the limitations of our detection algorithm is that it checks for anomalies *after* execution is complete (for each invocation). Combining a sequence-based method with our SCFD can be a solution if such attacks can be detectable

[10] Note that we found 11 clusters from the training. To test for 15 clusters, we simply added 4 duplicated clusters (i.e., means and inverse covariance matrices).

by the former. If not, one can increase the chances of detecting such problems by splitting the whole execution range into blocks [162] and checking for the distribution of system calls made in each block as soon as the execution passes each block boundary. This also can relax the assumption of repetitive execution of the target application (explained in Section 4.2) because an analysis is applied at block-level. This, however, would need more computation in the secure monitor at run-time, more storage for profiles, and a few more code modifications.

Another way to handle this problem is to combine this analysis/detection with other behavioral signals, especially ones that have a finer granularity of checks, e.g., timing [162]. Since some blocks may use very few system calls or even a very stable subset of such calls, we can monitor the execution time spent in such a block to reduce the SCFD-based overheads (which is still low). This keeps the profile from bloating and prevents the system from having to carry out the legitimacy tests. We can also use the timing information in conjunction with the system call distribution; i.e., by learning the normal time to execute a distribution of system calls, we can enforce a policy where each application block executes all of its system calls within (fairly) tight ranges. This is, of course, provided that the system calls do not show unpredictable timing behavior. This makes it much harder for an attacker who imitates system calls [147] or one that replaces certain system calls with malicious functions [13].

We can model a primitive operation (such as a network activity, a file operation, etc.) as a topic and then represent an execution context as a mixture of several primitive operations [44]. In this work, we build instead a pragmatic lightweight module. One of the main drawbacks of the k-means clustering algorithm is that one may need to know or pre-define the number of clusters. That is, system behaviors should be correctly represented by k numbers of multinomial (Gaussian) distributions of histogram. Some large-scale systems would have many heterogeneous modes (distributions). In this case, the appropriate solutions would be using non-parametric topic models such as Dirichlet process. However, we empirically observed that many embedded systems with predictable behavior can be represented by a tractable number of clusters. Thus, we used a simpler model with the k-means cluster.

4.6 Conclusion

In this chapter, we presented a lightweight anomaly detection method that uses application execution contexts learned from system call frequency distributions of embedded applications. We demonstrated our technique for a real-world open-source application and showed that the proposed detection mechanism could effectively complement sequence-based approaches by detecting anomalous behavior due to changes in high-level execution contexts. An interesting extension of this work could be using the topic modeling approach to deal with large-scale heterogeneous behaviors of complex embedded applications.

Chapter 5

The DragonBeam Framework: Hardware-Protected Security Modules for In-Place Intrusion Detection

The sophistication of malicious adversaries is increasing every day and most defenses are often easily overcome by such attackers. Many existing defensive mechanisms often make differing assumptions about the underlying systems and use varied architectures to implement their solutions. This often leads to fragmentation among solutions and could even open up additional vulnerabilities in the system.

In this chapter, we present the *DragonBeam Framework*, an extension of the SecureCore architecture presented in Chapter 2 for general-purpose systems. It enables system designers to *implement their own monitoring methods and analyses engines* to detect intrusions. It is built upon a novel hardware/software co-designed mechanism. Depending on the type of monitoring that is implemented using this framework, the impact on the monitored system is very low. This is demonstrated by the use cases that also showcase how the DragonBeam framework can be used to detect different types of attack.

5.1 Introduction

As attackers expand their reach into well protected systems, no layer is safe from intrusions. The targets of attacks in recent years have ranged from applications to middleware services, operating system (OS) kernels and device drivers, hypervisors and even firmware. The sophistication of such attacks, then, makes it harder to identify and build a trusted computing base (TCB) to develop security mechanisms. The common approach has been to move security monitoring (e.g., the reference monitor) functionality into a secure domain (e.g., a virtual machine separated from the virtual machine that must be secured) [65, 113]. This suffers from the existence of a *semantic gap* between the interface used for monitoring and the interface useful for security decisions. Another problem is that different security mechanisms use a variety of architectures to implement their solutions. Trying to combine one or more of these to improve the overall security of the system could result in a spaghetti of architectural mechanisms that, in itself, might open up new vulnerabilities. Hence, there is a need to provide system designers with a cohesive framework for implementing their monitoring and analysis techniques.

Our approach to solving such problems is to start with secure hardware and then to *bootstrap security into higher*

layers. We propose a system where the *secure hardware* is the first level TCB and introduces security monitoring components into the system layer above (in this case, the OS kernel). By running the monitoring component in the layer that is the target for attackers, we gain significant visibility into local operations as well as effects of attacks – thus avoiding the semantic gap. The secure hardware ensures the run-time integrity of the upper-layer security monitoring component(s) by *protecting it* and *continuously validating its liveness and behavior*. Designers can then implement methods/hooks (to monitor the system resources/components that they care about) in our security monitoring component. The gathered information can be offloaded to a different core of the processor where the designers can apply their own analysis techniques on the collected data. We call this the *DragonBeam framework*.

The DragonBeam framework is a set of software and hardware mechanisms that allow us to develop and maintain a *two-level monitoring system* that consists of: (i) a *kernel module* that resides in the OS kernel – it carries out any desired security checks and measurements and (ii) capabilities to monitor the *behavior* of this module and to establish its integrity by a combination of (a) integrity measurement and (b) run-time challenge-response protocol – these latter components actually execute on a trusted computing base that resides on the secure core. One advantage of using such methods is that the DragonBeam framework *does not* require modifications to the OS or the applications. In addition, since the DragonBeam architecture provides a separate core for executing the TCB-related components, (i) the overhead for the continuous security interactions is low and (ii) the effects on the critical executions paths is limited (and often negligible).

The methods presented in this chapter lay the groundwork for DragonBeam to become a *generic framework for developing/implementing security solutions*. System designers can implement their favorite monitoring and/or data capturing methods in the kernel module that can then either analyze the data itself or relay it to additional analysis components on the TCB. These components can perform more extensive analyses as required.

Our work makes the following contributions:

- We introduce a novel framework, DragonBeam, for implementing monitoring and intrusion detection solutions – it provides hardware-guaranteed integrity for the security monitoring system. The latter can be extended to other system layers to create a multi-level monitoring system rooted in the secure hardware. An overview of the framework is presented in Section 5.2 while details are in Section 5.3.

- We have implemented the framework and carried out evaluations on an FPGA softcore processor (Section 5.5). This helps us in gauging the real hardware costs for implementing such a system. As we see later, the additional hardware costs are less than 1%.

- Two use cases as well as a performance evaluation are used to illustrate how to use the framework in Section 5.6. These use cases not only demonstrate the ease of use but also highlight the flexibility of our approach by

showcasing different types of attack detection methods. The evaluation shows negligibly small performance overheads.

5.1.1 Threat Model and Assumptions

We aim to make our threat model as broad as possible which is in line with recent developments [53, 92]. An attacker can breach any part of the software stack (OS kernels, middleware, run-time libraries, applications to name a few) and can even have full control of any software running on the main (monitored) system. We assume that the attacker does not perform physical attacks against the hardware. Thus trust can be placed in the hardware components, and in particular in our TCB. We further assume that the whole system (both software and hardware) is secure during boot time as well as immediately after the boot sequence is complete; also updates to the TCB require physical access.

While our threat model is quite broad, an attacker may attempt to carry out malicious activities between operations that verify the integrity of the DragonBeam framework – the attacker could corrupt some components of our framework and restore it just before the next check. Such transient attacks [151, 79] cannot completely be ruled out in external monitoring mechanisms [118] and the DragonBeam framework is no exception. On the other hand, some of the mechanisms presented in this work, viz., the randomization techniques (Sections 5.3.4, 5.6.1 and 5.6.2), will help mitigate such attacks.

5.2 Overview

The main idea that we propose in this work is that of a *hardware/software mechanism* to detect intrusions. This is achieved by using a two-level monitoring framework that we call DragonBeam. This architecture takes advantage of the redundancy available in computing resources on a modern multicore architecture – we trade off performance to improve overall security by using one of the cores to *monitor* the other core(s). This mechanism can monitor the operating system, the applications executing on the monitored core, or both. Figure 5.1 presents the DragonBeam architecture, which we illustrate with use cases in the remainder of this section.

5.2.1 High-level Architecture

Figure 5.1 shows the DragonBeam framework, in which a trusted on-chip entity, the *secure core*, continuously monitors the run-time behavior of another (potentially untrustworthy) entity, the *monitored core*. The secure core is part of our trusted computing base. Since both cores are on the same die, minor hardware modifications are required to extract the relevant information directly from the monitored core. This increases the trustworthiness of the monitored signal since they are transparent to any code that executes on the monitored core. On the other hand, the amount of

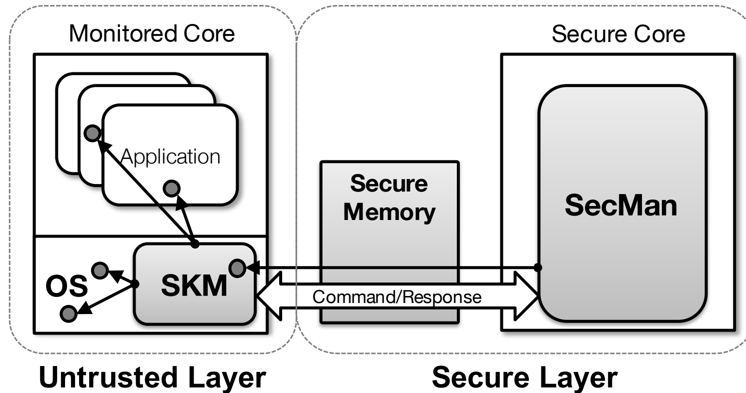


Figure 5.1: Overview of the DragonBeam architecture.

information that can be gathered is somewhat limited by the amount of hardware changes (essentially probes) that can be made. Such changes are often intrusive and might require significant efforts in design and verification.

To solve these problems and to increase the amount of information that the secure core can gather, we introduce a *software module* that executes inside the monitored core as a kernel module. We call it the *Secure Kernel Module* (SKM). The SKM is responsible for capturing information about the applications as well as the OS that executes on the monitored core and passes this information (with help from a hardware unit) to the secure core. The SKM is controlled by a software module that executes on the secure core called the *SKM Manager* (SecMan). The SecMan also ensures that the SKM is not attacked or prevented from executing or carrying out its intended functions. The SKM acts as a bridge to gather information about the behavior of the OS and/or applications on the monitored core; this information is then sent to the secure core for analysis.

Hence, the overall architecture of our solutions is composed of *three* major components, viz.,

1. the Secure Kernel Module (SKM), which runs in the untrusted operating system,
2. the SKM Manager (SecMan), which protects the integrity of the SKM, and
3. the secure core, which runs the SecMan.

We now elaborate on the SKM and SecMan components in the following sections.

Secure Kernel Module (SKM)

The SKM is a *kernel module* that resides on the monitored core. It is controlled by the SecMan and carries out a variety of security functions, chief among which is to *gather information about the execution behavior* of important OS components and applications. It could either actively analyze or passively send to the secure core the information that is gathered. For instance, it could (a) perform integrity checks on the kernel code and/or data structures, (b)

monitor the behavior of some critical applications, (c) actively monitor what processes/modules are executing, and (d) what low level resources are being requested by what processes and how they are being used among other things. The SKM executes in the most privileged mode as do other kernel components. Hence it has access to all of the kernel data structures. This helps it detect anomalous activities on the monitored core.

The DragonBeam framework does not prescribe in any way the actual functionality of the SKM. The goal of the framework is to ensure that the SKM operates without any external effects on its code or data, even when the attacker has root-level access to the system. The execution of the SKM is closely tracked by the SecMan – this prevents the situation that the SKM itself is taken over or prevented from executing. The SKM is developed by the system designers themselves and also has a fixed, limited, functionality. Hence, it is easier to verify that the SKM itself does not expose any security vulnerabilities.

Secure Kernel Module Manager (SecMan)

The SecMan actively manages and monitors the execution of the SKM. The SecMan and SKM follow a *command and response protocol* that has been developed by the system designer. Consider the following example to illustrate this: (a) the SecMan tells the SKM to capture the current state of the process list in the kernel; (b) the SKM wakes up and gathers this information; (c) the SKM communicates this information back to the SecMan and finally, (d) analysis modules on the secure core will compare the state of the process table to ones that were captured previously to check for unexpected processes – this might allow us to detect the execution of malicious process on the monitored core. The overall protocol will include a fixed set of commands that bound the operations of the SKM. At run-time, the SecMan will issue one of these commands (or a sequence of them) and the SKM will execute them in the order received.

Another important function of the SecMan is to *guarantee the integrity and the liveness of the SKM itself* via code hash and a heartbeat mechanism. This is why we call this a two-level monitoring architecture – the SKM monitors the behavior of kernel components and applications while the SecMan monitors the execution of the SKM itself.

As shown in Figure 5.1, the secure core is supplemented with a *secure memory* module that facilities secure communication between the monitored and the secure core. The secure memory relays commands from the SecMan to the SKM and also the data from the SKM (i.e., results from executing the commands) to the SecMan. The main aim is to carry out these operations in a trusted manner. This secure memory can only be accessed by either the SKM or the secure core. This ensures that the commands and responses cannot be intercepted, corrupted, or faked by an adversary.

5.2.2 Sample Use Cases

The integrity of the system call table in the kernel is very important. Some kernel rootkits often overwrite the entries in the system call table to hijack the execution of benign processes and hide the presence of malicious processes or files [13]. One way to detect such rootkits is to regularly check the state of the system call table – the state can be checked against what was stored at the secure-startup. Hence, the SecMan can send commands to the SKM to capture the current state of the system call table and copy it into the secure memory. If the recently captured state of the table deviates from what is expected, then the secure core can take corrective action or raise alarms. Similar operations can be carried out to verify the integrity of the interrupt vector and to find malicious kernel module or user processes.

5.2.3 Requirements and Challenges

For the rest of this chapter, we will address the following requirements and challenges in implementing the two-level SKM-based monitoring architecture:

1. The SKM must not be compromised even if the kernel on the monitored core is successfully attacked; also, the SecMan must be able to detect if the SKM is no longer operating as expected.
2. The SKM should act promptly in response to commands from the SecMan.
3. The secure memory must not be corrupted by an adversary, even when the adversary has gained root access on the monitored core. The access controller for the secure memory must ensure that only the SKM and the secure core have access to the secure memory.

5.3 Detailed Architecture

We now present more details about the DragonBeam framework.

5.3.1 DragonBeam Framework Operations

We will explain the details behind each step in the DragonBeam framework using the example of detecting problems in the system call table (explained in Section 5.2.2). An overview is presented in Figure 5.2. The steps are:

1. The SecMan sends a command (via the secure memory) to check the system call table.
2. The above process results in an inter-core interrupt that is handled by an interrupt handler on the monitored core.[1] The interrupt-service routine, `skm_ISR()`, is the main body of the SKM and handles the command sent

[1] We will use 'monitored core', 'application core' and 'main core' interchangeably.

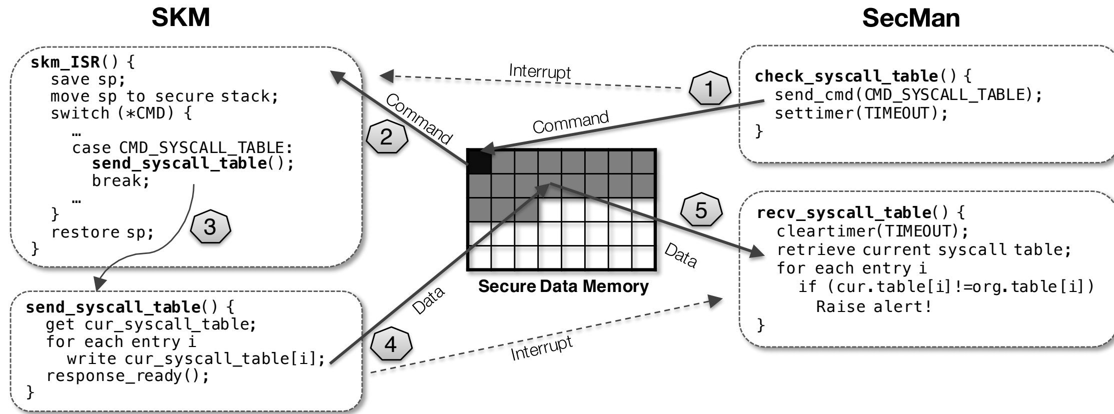


Figure 5.2: Overview of the execution of an example security task (integrity check of system call table) in the DragonBeam framework. The SecMan issues a request (1) to the SKM via an interrupt. The SKM collects the data needed (i.e., the contents of the system call table) and passes the data back to the SecMan via the secure memory in steps (3) and (4). Finally, the SecMan verifies the integrity of the received data (5).

by the SecMan. No entity on the main core (other than the SKM) can know the command because *access to the secure memory is restricted.*

3. The SKM calls the appropriate function to carry out the required task.

4. The function carries out its operations by placing the required information in the secure memory.

5. The secure core receives an interrupt that the data it requested is now available in the secure memory. On receiving the interrupt, the SecMan reads out the information from the secure memory region and then sends it to the appropriate module in the secure core that can analyze this information.

The above process repeats for every command that is sent by the SecMan to the SKM. Let us now look at the details that enable the above process.

5.3.2 SKM Registration

The SKM is loaded onto the monitored core during the booting phase of the OS. Our assumption here is that the system is in a clean state at boot time (we could even use a secure-boot mechanism such as IMA [123]).

The first task for the SKM (at load time) is to request the SecMan to register it. Figure 5.3 shows the SKM registration process. For the SecMan to test the veracity of the SKM's request for registration, it uses a hash of the latter's `.text` section. Since we do not trust the SKM yet, we do not wait for it to send its hash; rather, we calculate it *directly from the main memory.* This prevents malicious modules from copying the SKM's hash in order to pass off as legitimate modules.

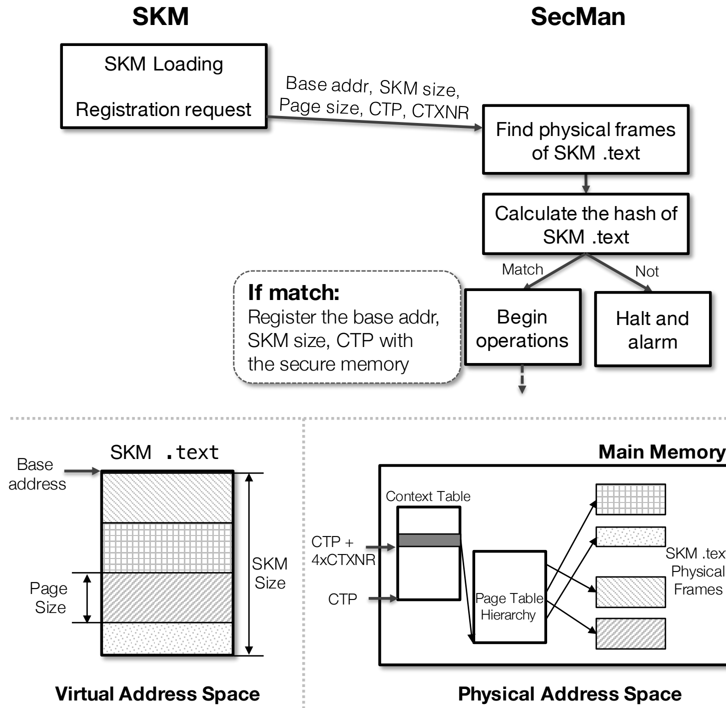


Figure 5.3: SKM registration during secure boot.

The SecMan needs the following information about the SKM to compute this hash value: (a) the virtual base address, (b) the size of the `.text` section, (c) the page size, and (d) the page table information—all for the SKM. In SPARC processors (that our prototype is based on; see Section 5.5), the last piece of information corresponds to the *context table pointer* (`CTP`) and the *context number* (`CTXNR`) [137]. The `CTP` points to the root of the page table tree and the `CTXNR` is used to index the context table (i.e., the page table) of the current process (hence `CTXNR` is unique for each process). The SecMan then translates the `.text` section of the SKM as defined by the base (virtual) address and its size to a set of physical addresses that host the SKM code. The SecMan then calculates a hash of the physical frames that store the SKM code using, for instance, the SHA-1 algorithm [59] (or whatever hashing algorithm that the systems designers favor). If the newly computed hash matches what was calculated at the design time, then the SecMan registers the base address for the SKM's `.text` section, its size, the page size, `CTP` and `CTXNR` with itself. Also, the base address, size, and `CTP` are registered with the secure memory controller. From then on, the secure memory can *only* be accessed by the code that is verified to be part of the SKM.

Note that a malicious module may make a registration request to the SecMan. However, with the registration process described above, the only way for the attacker to pass the registration process is to implement the malicious code in such a way that its `.text` section has the same hash value as the SKM's one – remember that the hash is *directly* calculated by the SecMan from the given information about the malicious module's `.text` section.

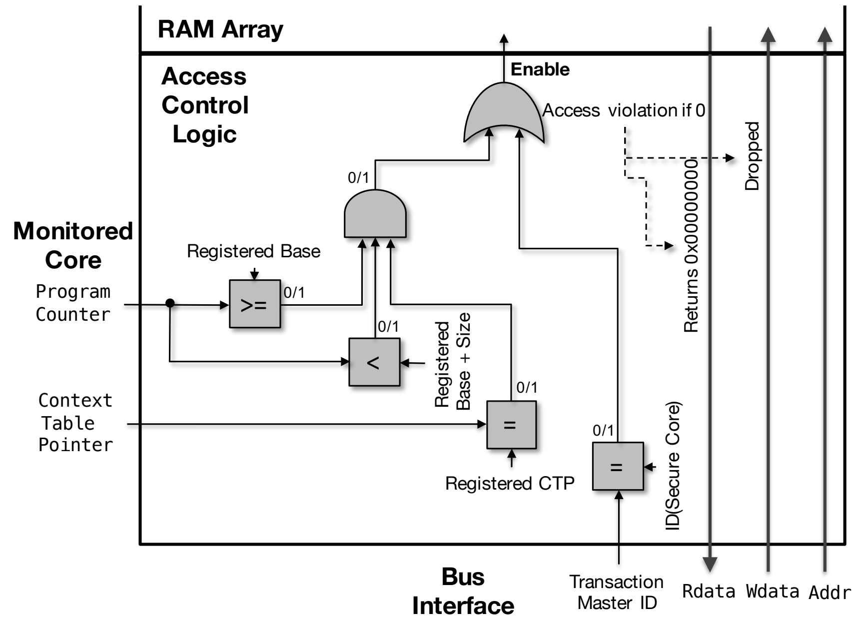


Figure 5.4: Secure memory access control.

5.3.3 Secure Memory and Access Control

The secure memory enables secure and trusted communication between the SKM and the SecMan. The secure memory controller only allows the SKM on the main core (and any from the secure core) to access this on-chip memory. To implement this access control, we use the program counter value to identify *who initiated* a memory transaction to the secure memory. The program counter-based memory access control has been used in the context of embedded devices with no support of virtual memory [139, 105, 90]. In systems with virtual memory, however, the program counter cannot be used as a unique identifier. Hence, we combine it with a coarse-grained check on the address mapping information.

Figure 5.4 presents the overview of the secure memory access control process. The secure memory controller accesses the program counter (a virtual address) register (`PC`) on the main core as well as the context table pointer register (`CTP`). It then checks if (i) the `CTP` register matches the one registered by the SecMan during the SKM registration phase and (ii) the value of the `PC` register is within the `.text` region of the SKM. Note that we do not use the context number register (`CTXNR`) because the kernel memory address mappings are identical across all contexts on SPARC [137] and thus the context number is ignored by the MMU during kernel address translations. If the above two conditions are satisfied, we can ensure that only the SKM can access the secure memory. If the memory controller detects an illegal access attempt, the controller returns 0 for read or drops write transactions.

However, it is entirely possible that a smart adversary may have modified the page table referenced by the legitimate `CTP` register so that the virtual addresses indexed by [Base, Base+Size] are mapped to the malicious code

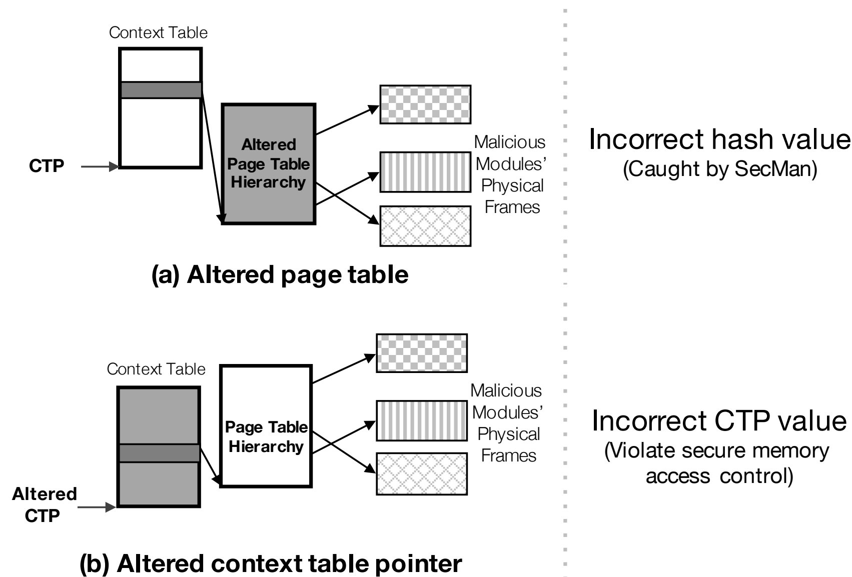


Figure 5.5: Possible attack scenarios on the SKM page mapping.

instead of the SKM's physical addresses as illustrated in Figure 5.5(a). Also, as shown in Figure 5.5(b), the adversary may have set up a whole new context table tree at a different location. The malicious module might then be able to access the secure memory by modifying the address mappings. To prevent such problems, we could check the physical addresses of the instructions trying to access the secure memory to verify if it falls within the SKM's physical frames. However, we avoid this option since it would involve an address translation each time the secure memory is accessed—this would result in huge performance overheads. Our solution (as will be elaborated in Section 5.3.4) is for the SecMan to (a) translate the SKM's `.text` section (indexed by [Base, Base+Size]) from virtual to physical addresses using the registered `CTP` and `CTXNR` register values and (b) calculate the hash of the resultant frames—following exactly the same process as the registration, only done *often during execution*, not just at the registration time. If the attacker has modified the page table so that the virtual address range [Base, Base+Size] points to its malicious code (case (a) in Figure 5.5), the resultant hash value will not match the legitimate value. Otherwise, we can be sure that the range [Base, Base+Size] is pointing to the SKM. Also, an altered `CTP` value (case (b) in Figure 5.5) cannot be effective because this simply violates the secure memory access control. Of course, these do not eliminate the possibility where a carefully constructed malicious module changes the address mappings and restores it to the original between two hash check points. However, regular hashing with random time intervals can significantly reduce the possibility of success for such attempts.[2]

[2]The attacker would want to do this type of transient attack in an attempt to impersonate the SKM and send fake response through the secure memory. The attacker would therefore try to alter the address mappings before the SKM is activated by a command from the SecMan. Such a threat can be significantly reduced if the SecMan performs the hashing right before sending the command to the SKM.

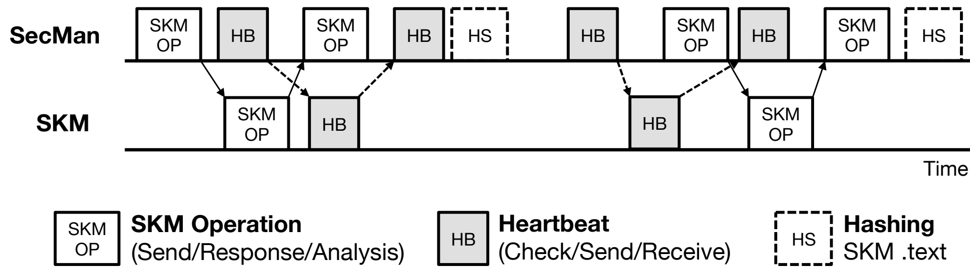


Figure 5.6: Timing diagram of SKM operation, heartbeat and hashing.

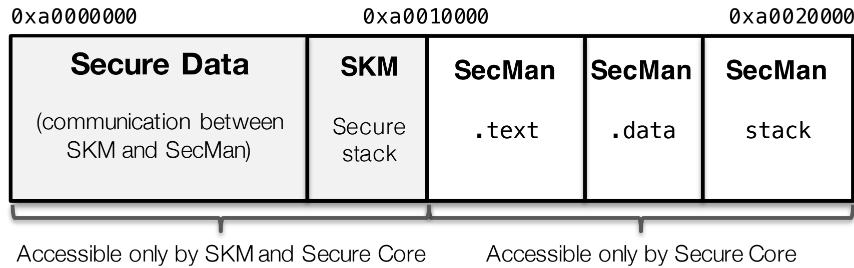


Figure 5.7: The secure memory map.

5.3.4 Heartbeat and SKM Integrity Check

As mentioned earlier, we need to continuously check if (i) the SKM's code has not been tampered with; (ii) the SKM is actually starting up when it is commanded to do so by the SecMan; and finally (iii) the SKM responds to requests in a timely manner. We have already discussed a technique to check for (i). The issues of liveness of the SKM, viz., (ii) and (iii), require a more active approach.

Our solution for this problem is to occasionally send out a special *heartbeat* command to the SKM as illustrated in Figure 5.6. The SKM should respond to this command right away. This ensures that the SKM has not been deactivated by an adversary. No other process can respond to this command since the communication happens through the secure memory that can be accessed only by the SKM. When a heartbeat command is sent to the SKM, the SecMan starts a countdown timer. If it does not receive the expected response from the SKM then this is also indicative of an attack. The timer can be used for any/all commands that the SecMan sends across.

Another precaution that we can take is to send out these heartbeat commands, in addition to the SKM operations and hashing, at *random* intervals so that an adversary cannot guess the pattern of checks by the SecMan.

5.3.5 Secure Memory for SecMan and Secure Stack

To ensure the integrity of the SecMan we load its code and data onto a part of the secure memory region that can be accessible only by the secure core, as shown in Figure 5.7. Hence, an attacker that takes control of the main core

cannot corrupt the SecMan. The SKM is loaded onto the main memory (along with the rest of the kernel),[3] and we monitor and protect it by the mechanisms described earlier.

The secure memory also hosts the *secure stack* for the SKM. When an interrupt is raised and handled by the SKM, the first step taken by the ISR is to change the stack pointer to the secure stack (see Step 2 in Figure 5.2). Hence, a malicious module cannot read or alter the data stored in the secure stack even if it knew where the stack is.

5.4 Security Guarantees of the DragonBeam Framework

The DragonBeam framework provides for the secure operation of a kernel module, in the presence of attackers with full access to the system, including kernel-level access. We outline here the potential attack vectors and the defenses that the DragonBeam framework provides.

SKM Code Integrity: An attacker may try to replace the SKM with a malicious module and thus have it loaded onto the system, either during the boot-phase or later during system operation. This is prevented by the SKM registration process which requires the exact hash value obtained at the design time. Also, the hash is directly computed by the SecMan using the page table information of the caller to dissuade such attempts. The attacker may also try to modify the SKM's ISR. However, since it is placed in the SKM's `.text` section, any attempts by the attacker to change the ISR will be detected by the hashing mechanism mentioned earlier.

SKM Control Flow: The attacker may try to change the state or parameters used by the SKM in order to cause a buffer overflow and change its execution flow. This can take many forms, from code-injection attacks, to return-to-library attacks and to return-oriented-programming attacks. All of these attacks rely on a software module processing its inputs in a vulnerable way; for the purposes of this work, we assume the SKM is well designed and implemented to avoid such problems. This is a realistic assumption as the SKM is supposed to be functionally self-contained and small.

SKM Availability: The attacker may try to disable the SKM by preventing it from being scheduled for execution on the CPU or disabling interrupts. Hence, to guarantee the liveness of the SKM, we use a heartbeat mechanism (Section 5.3.4). The attacker may try to impersonate the SKM by redirecting the interrupts to itself and by responding to the heartbeats. We prevent this situation by using the secure memory as a communication channel; no entity, other than the SKM, can write a response to the secure memory and thus the fake response cannot be delivered to the SecMan.

SecMan Integrity: The attacker may try to corrupt the SecMan directly so that none of its security functions are

[3]The SKM could be loaded onto the secure memory. However, this requires a modification of the OS, which we avoid.

Table 5.1: Details about the Implementation Platform.

Implementation Artifact	Value
Platform	Leon3 on Xilinx ZC702 FPGA
Processor	SPARC V8 Dual Core @ 83.3 MHz each
Main Memory	256 MB
L1 cache	Split, 16 KB, LRU
TLB	Split, 8 Entries, 4KB page, LRU
Secure Memory	128 KB, Single-port
Monitored Core OS	Linux 3.8
Secure Core OS	None (Bare Metal Execution)

invoked in the first place. This cannot happen in our architecture since the attacker cannot access the memory region of the SecMan.

OS Integrity: An attacker may try to create a distinction between the code and data of the actual running kernel and the code and data inspected by the SKM. Indeed such attacks are possible – more so with the run-time splicing support present in kernels these days. Such a distinction between the running and observed kernel states might allow an attacker to perform malicious tasks – the SKM may not detect these anomalies if it is not well designed. For example, an attacker who wishes to hijack the system call table could modify the software interrupt table to point to a new system call dispatch handler, which in turn uses a new system call table placed elsewhere in memory. Thus, the SKM's task is not just to read and check the system call table but also to verify that the code that is supposed to use this table is valid and active. At a minimum the SKM must check the interrupt descriptor table, the code of the appropriate interrupt handler and, finally, the system call table used by that code. We leave the design of such checks to the designer of the SKM while we ensure the integrity and confidentiality of the checks via the DragonBeam framework.

5.5 Implementation

We implemented the DragonBeam framework on a Leon3 processor [14] for a Xilinx ZC702 FPGA [26]. Leon3 is a soft-core processor based on 32-bit, in-order, 7-stage pipeline SPARC V8 architecture [137]. From the soft-core implementation, we (a) demonstrate the ease with which we can make the required modifications and (b) measure the hardware costs for such an implementation.

5.5.1 System Configuration

Figure 5.8 shows our DragonBeam framework implementation on Leon3 processor and Table 5.1 lists the details about the implementation. The system consists of two cores each of which runs at 83.3 MHz and the system has

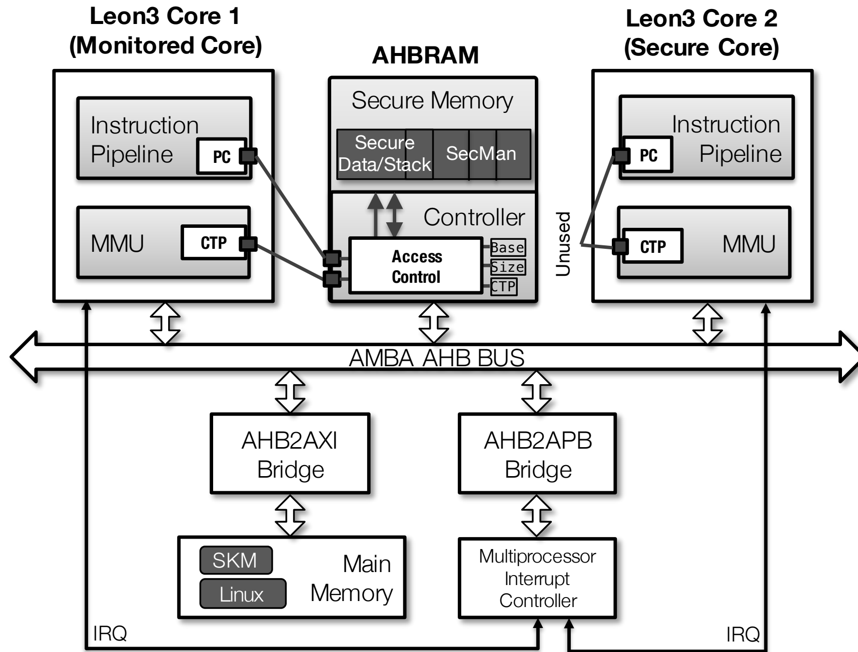


Figure 5.8: The DragonBeam framework implementation on Leon3.

a main memory of 256 MB. Each core has L1 instruction (16 KB) and data (16 KB) caches. The MMU has split TLBs for data and instruction. The Leon3 processor also includes a single-port on-chip RAM, AHBRAM, to which the cores can access through AMBA [1] AHB (Advanced High-Performance Bus) bus, as depicted in Figure 5.8. We instantiated it as an 128 KB on-chip SRAM that is addressable at 0xa0000000. The first half is used as the secure communication channel between SKM and SecMan and also for the secure stack of the SKM, as shown in Figure 5.7. The bottom half is used by the SecMan. The entire region is set to be *uncacheable*, otherwise a non-SKM process can access the cached data without accessing the secure memory.

5.5.2 Secure Memory Implementation

We modified the control logic of AHBRAM, i.e., the on-chip SRAM, to implement the secure memory. As explained in Sections 5.3.2 and 5.3.3, the information about SKM's `.text` and page table are stored in the secure memory for the access control. For this, we designate the first 16 bytes of the secure memory (i.e., 0xa0000000–0xa0000010) as special memory-mapped registers in which the SecMan can write the information during the SKM registration phase. The controller checks the AHB master ID of the memory transaction and grants access to these registers only from the secure core. Hence, the information can be set only by the SecMan. The bottom half of the secure memory used by the SecMan (explained in Section 5.3.5) is protected in the same way.

After the SecMan has validated the SKM at registration time, the SecMan *locks* the memory that contains the

control data (i.e., the first 16 bytes) by asserting the lock bit in the control register. From then on, access control to the secure data and secure stack regions become enabled. As explained in Section 5.3.3, the access control logic requires the current program counter (`PC`) and context table pointer (`CTP`) values from the monitored core. As shown in Figure 5.8, we extract the `PC` value from the fetch stage of the core's instruction pipeline and the `CTP` value from the MMU and feed them to the secure memory controller. It returns 0 or drops the transaction for an illegal read or write request, respectively.

5.5.3 Software Configuration

The monitored core runs an *unmodified* Linux 3.8 kernel residing on the main memory as shown in Figure 5.8. The SKM is implemented as a Linux Kernel Module and resides in the main memory. The SKM has about 350 lines of C code (including spaces) that implement the two use cases introduced in our evaluation (Section 5.6) along with interrupt handling and inter-core communication routines.

We implemented the SecMan as a *bare-metal executive* running on the secure core for the purposes of our proof-of-concept. A complete system can also have an OS and analysis modules running on the secure core. The SecMan has about 450 lines of C code (excluding the SHA-1 library), a majority of which is for interrupt and timer-related functionality. As mentioned above, the SecMan resides in the bottom half of the secure memory, accessible only by the secure core.

5.6 Evaluation

In this section, we evaluate the DragonBeam framework along the following lines: (a) how it can be used by system designers to implement different detection mechanisms to catch malicious activities; (b) the overheads imposed by the DragonBeam framework on the main system; and (c) the costs for implementation in hardware.

5.6.1 Implementation of Detection Mechanisms

To demonstrate the effectiveness and versatility of our framework we implemented two existing detection mechanisms. Note that we are *not* proposing new detection techniques for any of the use cases presented here. We instead intend to demonstrate *how to use* our framework for the benefit of system designers. We also intend to show how SKM closes the semantic gap by running directly inside the untrusted OS and collecting information useful to security decisions.

Hidden Module Detection

Many kernel rootkits such as modhide [17], suterusu [24], etc. hide themselves from the kernel module list to avoid detection by anti-virus software. The hidden modules are invisible from even lsmod or \proc\modules, both of which read the list of currently loaded modules from the same kernel data structure. For the following experiments, we used the suterusu kernel rootkit that hides by deleting itself from the kernel module list.

Hidden kernel modules can be detected by scanning the memory region where modules are typically placed. The main idea is that every page that is present in this memory region should be allocated to one of the modules present in the kernel module list as such pages are not swapped out. If even one page cannot be matched up to a known module, then it is an indication of a hidden module. For instance, in our experimental setup, the memory space where kernel modules reside lies in the range of 12 MB which hosts 3072 pages of size 4 KB each.

We implemented the detection method using our DragonBeam framework as follows:

1. For each page in the module memory space, we check if it has been loaded into memory by checking the requisite flag, viz., the present bit. We collect information on all pages in this region.
2. We traverse the module list. For each module, we obtain its base address and size – this corresponds to the list of pages used by the module. We mark off the pages associated with each module from the list of pages obtained above.
3. If any of the pages from Step 1 have not been marked off at the end of Step 2, then it is an indication of the existence of hidden modules in the system.

The SecMan sends commands to the SKM to execute the procedure described above.[4] The SKM replies with the results of this checking procedure. To prevent attackers from evading the checking procedure mentioned above, the SecMan must send the commands at *random* points in time.

Using our checking mechanism, the SKM found *two pages* that had been allocated for suterusu's code and data. Other rootkits that operate in a similar manner will also be caught by this procedure.

System Call Table Integrity Check

Many rootkits *hijack* system calls [13] to intercept sensitive data, hide malicious processes or files, etc. One way to detect such attacks is to verify that the current state of the system call table matches the original state obtained immediately after a secure boot. This will detect rootkits that rewrite entries in the table.

[4]Note that the SecMan could have just asked for the pieces of information – the list of modules and their page usage along with the list of allocated pages in the kernel space. It could have then executed the matching procedure as well. We do it in the SKM for this experiment only for convenience – it is up to the system designer to decide where to carry out the analysis/checking.

First, the SecMan asks the SKM to capture the *initial* state of the system call entry table. This information is passed via the secure memory to the SecMan. This happens right after the SKM has registered with the SecMan, at which point the OS state is still trustworthy. The SecMan stores this initial state in its internal memory region, part of the secure memory. During regular execution, the SecMan asks the SKM to send snapshots of the system call table. The SecMan compares the newly received state information with the one obtained initially. If it detects a change in the table, then that is an indication of a rootkit having hijacked certain system calls.

We used the `modhide1` [17] rootkit for our experiment – it hijacks the open system call to prevent the detection of a module being inserted into the kernel (e.g., cat \hide hides the module). Using the method described above, the SecMan was able to detect this rootkit's presence. More generally this approach can check the state of any critical kernel data such as the interrupt descriptor table and handlers (as described in Section 5.4), page tables and translation base register, etc.

5.6.2 Performance Evaluation

We now analyze the overhead imposed on the main core due to the execution of the DragonBeam mechanisms.

Table 5.2 shows the latencies of the heartbeat and hashing operations as well as the operations necessary for use cases explained in Section 5.6.1. The latency is measured between the time points when the SecMan sends a command to the SKM and to when the SecMan completes the analysis after receiving data from the SKM. Each entry in the table presents the average of 1000 measurements and the standard deviation. Note that the hashing operation does not involve the SKM and thus the latency is simply the time required to (i) copy SKM `.text` to the secure memory and (ii) perform the SHA-1 operation. The time spent for the system call table check is to copy a table of 343 entries (total size 1.3 KB in Linux 3.8 on SPARC) and then to compare it (with the stored version of the table) entry by entry. The hidden module detection operations take much longer because it checks (i.e., looking up each entry in the three-level page table) all the pages in the kernel module space (3072 pages).

Beyond microbenchmarks, we used the SPECINT2006 [72] benchmark suite[5] and measured execution times for two situations: when the SKM is enabled versus when it is disabled. Specifically, we executed each benchmark 20 times for each of the following scenarios: (a) when the SKM is not running, (b) heartbeat, (c) SKM `.text` hashing, (d) system call table integrity check, and (e) hidden module detection is enabled. To obtain stable results, we used a fixed period (100 ms at the CPU clock speed of 83.3 MHz) for the operations. Hence, the SKM was sent commands to execute each operation about 1600–6400 times during one benchmark execution.

Figure 5.9 shows the average (geometric mean) ratios of the execution times for each benchmark when the SKM

[5] We used five benchmarks – `bzip2, hmmer, libquantum, mcf, sjeng`– from the suite after excluding ones that are failed to cross-compile to the Leon 3 SPARC platform or ones that took very long time to execute one single execution trace (since the FPGA softcore is slower on average than regular processors). The average execution time for each of these benchmarks (without the DragonBeam framework) are 370, 198, 161, 511 and 639 seconds, respectively.

Table 5.2: Average latencies of SKM operations.

SKM Operation	Avg Latency (stdev)
Heartbeat	0.010 ms (4.100 us)
SKM `.text` hashing	2.679 ms (7.474 us)
System call table check	0.108 ms (5.727 us)
Hidden module detection	3.914 ms (4.772 us)

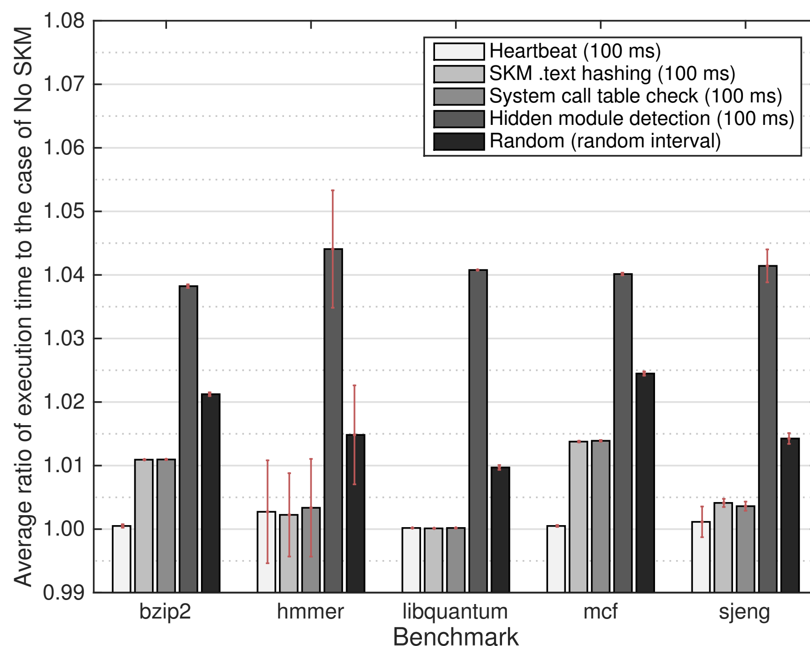


Figure 5.9: The average ratio of execution time to the case when SKM is disabled for different SKM operations.

is enabled (with an SKM operation) compared to when the SKM is disabled. As the plot shows, the overheads associated with the SKM operations are very small. Among them, the hidden module detection incurs the biggest overhead (around 4%). This is because it is an *in-SKM procedure*, i.e., the analysis carried out inside the SKM module. We could reduce this overhead by offloading the analysis to the SecMan – i.e., the SKM can just send the list of modules with their base addresses and sizes to the SecMan. Since the latter can physically address the main memory it can obtain the page information directly and check the flags. Overall, the overheads are consistent with the latency of each SKM operation (except hashing) shown in Table 5.2. As mentioned above, the hashing operation does not directly involve the SKM, however it generates a decent amount of bus traffic when copying the SKM's `.text` from the main memory to the secure memory. Hence, the operation indirectly imposes overheads on the main core by interfering with the memory traffic of the core. The system call table integrity check operation affects the main core's memory traffic in the same way because it copies the system call table between the main and secure memories. As we can see from Figure 5.9 the overheads imposed by these two operations vary more (as compared to the other two sets

of bars) across the benchmarks. This is because of the different memory footprints of the benchmarks. `bzip2` and `mcf` have a substantially larger memory footprint than the other three [73] and hence they experience more overheads by such SKM operations that require large memory transfers.

Lastly, we performed a similar experiment with *random* operations that execute at *arbitrary intervals*. The SecMan sends a command randomly chosen from the four operations and then schedules the next event at some Δt after the current time. We configured Δt to be randomly drawn from $[0, 1, \ldots, 200]$ ms so that the median is 100 ms. However, we configured the SecMan to send the next command as soon as the SKM responds to the current one if $\Delta t < 10$ ms. That is, with 5% of probability, SKM operations can be *back-to-back*. These make it very difficult for an adversary to predict when and what kind of SKM operation will occur next thus leading to a significantly reduced chance of success for transient attacks. The right-most bar of each benchmark group in Figure 5.9 shows the overheads imposed by this randomized check. The results are consistent with those of individual operation; the overhead is close to the average of the four operations.

These results show that our DragonBeam framework imposes very little overheads on the main system and it can still be an effective method to implement security mechanisms. One can also use the results when finding a proper combination of random periods (for different operations) according to the expected system load and allowable overheads.

5.6.3 Hardware Costs

Finally, we evaluate the hardware cost of the proposed architecture. The top half of Table 5.3 shows the cost of the hardware change in terms of the FPGA resource utilization (based on Xilinx ZC702 board [26]). The number in each cell is the number of resources used and the last column shows the extra resource due to the DragonBeam framework. The table shows that the DragonBeam framework adds a very small amount of hardware resources. This was possible because the only hardware changes in the framework are (i) the logic to fetch the 30-bit `CTP` (Context Table Pointer) register from the MMU, (ii) the 32-bit `PC` (Program Counter) register from the instruction pipeline, and (iii) the logic to implement the access control policy of the secure memory (see Figure 5.4). The logic for the two register fetch components is duplicated for each core although those of the secure core are not used in the access control. This result indicates that we can establish an on-chip secure communication channel between the SKM and the SecMan running on different cores (and also protect the latter's memory region) using less than 1% of additional hardware resources.

5.6.4 Extension to Multiple Monitored Cores

So far we have considered the situation where the DragonBeam framework runs on a dual core processor. While we have shown how this works, the issue remains that most modern systems have more than two cores. A typical

Table 5.3: FPGA Resource Utilization* of Leon3 Processor with and without the DragonBeam framework.

	Resource	Default	W/ DragonBeam	Δ
Dual Core	Registers	10258	10356	**0.96%**
	LUTs	19482	19511	**0.15%**
Quad Core	Registers	18932	19029	**0.51%**
	LUTs	37777	37835	**0.15%**

*Available resource: Registers (106400), LUTs (Look-Up Table, 53200)

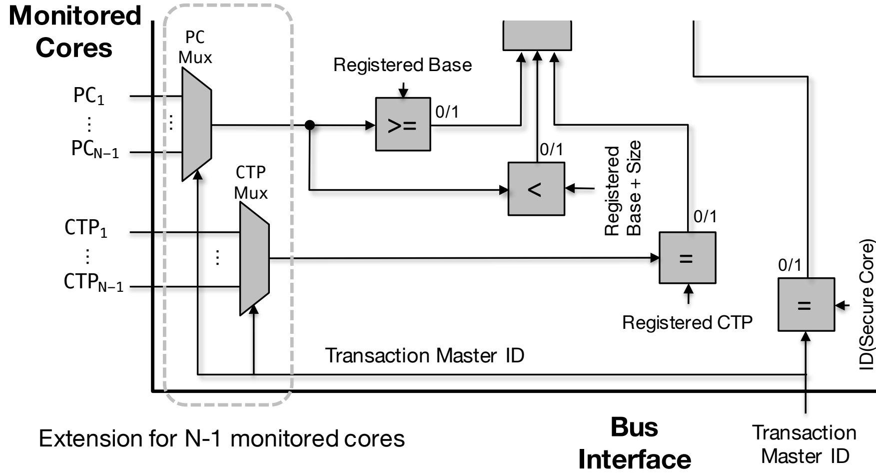


Figure 5.10: The secure memory access control for monitoring N-1 cores running a single OS and an SKM.

configuration for such systems is that they run a single OS that manages all cores. In such situations, the SKM can run on any core at any given time. Hence, *every* monitored core needs to be hooked to the secure memory controller. Now, when a memory transaction comes in, the program counters and the context table pointer registers of all monitored cores are *multiplexed* with the transaction master ID. The rest of the logic remains unchanged as only one SKM exists and thus only one set of base, size, and `CTP` is registered.

We extended the original architecture to a quad-core configuration in which three cores are monitored by one secure core as shown in Figure 5.10. The bottom half of Table 5.3 indicates that the extended architecture still imposes a very small hardware cost. This SMP configuration with a single SKM instance can reduce the performance overheads, especially due to in-SKM operations such as the hidden module detection, on the main cores because the SKM can run in parallel with the main applications running on different cores. To see this, we performed a similar experiment to the hidden module detection from Figure 5.9 in Section 5.6.2 with the quad-core configuration. The average overheads are between 0.8% and 2.1%, which are substantially lower than what we observed (about 4%) with a single monitored core on the dual-core setup.

For systems that have multiple operating systems executing on different cores (e.g., in the case of modern cloud

computing systems with virtual machines that share a single underlying processor), the secure memory controller should be re-architected as there can exist multiple SKMs running on different monitored cores. The controller, at the very least, needs to have a separate register set for each individual SKM's information (i.e., the base, size, and `CTP` value) and these should be multiplexed with the transaction master ID. Also, the secure memory should be partitioned so that the SKMs do not interfere with each other. However, we have not fully investigated if there needs to be additionally required HW changes.

5.6.5 Limitations

There exist some limitations of our approach. First of all, it is constrained to integrity checks. In other words, the DragonBeam framework does not cover information-leakage attacks (e.g., via side channels). In our model, access control mechanisms that are part of the DragonBeam framework are used to ensure the integrity and liveness of security components running in the untrusted OS but with no guarantees about the various side (or covert) channels created during the normal operation of the system.

A second limitation, specific to our prototype implementation is that an SKM cannot use kernel functions such as sprintf when the arguments are in the secure memory. One way to get around this issue is to write a wrapper function in the SKM that copies the required data to some part of the main memory and then pass it to the required library functions. Automating this through the means of an API that will automatically migrate data between the secure and insecure memory can mitigate this limitation.

Another important issue is that we trade off some performance for security – essentially one of the cores is reserved for security monitoring, which could otherwise be used as part of the main system. But this is something that system designers know about and can account for. This loss of performance comes with increased security guarantees – that might be fine for many systems where security is often very critical.

5.7 Conclusion

System designers must often contend with different modes and entry vectors of attacks, multiple security solutions and various monitoring and analysis techniques. A framework that can be used for integrating the different intrusion detection and analysis methods will provide significant value to such designers. In this chapter, we presented such a framework that we call DragonBeam. The use of this framework allows designers to implement and carry out their own monitoring and analysis without affecting the execution of the main system, i.e., little to no effects on the critical paths of the system.

Chapter 6

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

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. This chapter presents 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/logical system behavior 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.

6.1 Introduction

The booming 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. General-purpose operating systems, especially Linux-based OS variants, are becoming the leading OS for intelligent vehicles [28, 32], 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 [20], Qualcomm Snapdragon flight control board [32], and Intel's Aero board [34]. 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 [2]. 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.

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 work, 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.

Virtualization also allows for the use of a rich set of existing security techniques such as virtual machine introspection [65] that can be found in general-purpose systems. 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 work 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 and also 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 including attempts to hijack and to cause safety violations. We also show how the framework can provide critical safety measures at the protected layer and also close some of the side channels.

6.2 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 work 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.

6.2.1 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 1.2 in Section 1.2.2) 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.

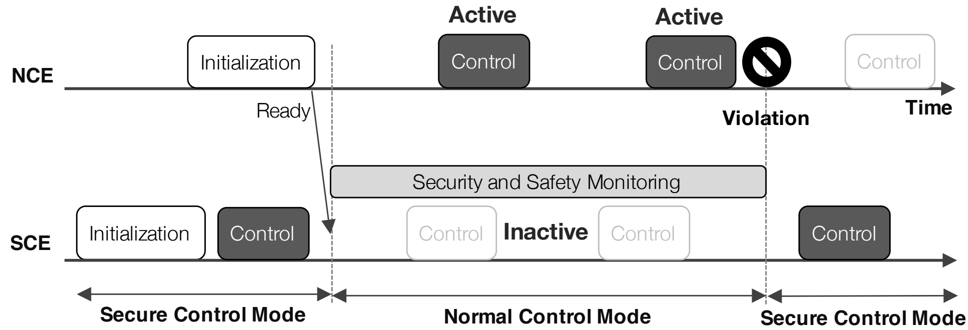


Figure 6.1: Switching between the SCE and the NCE.

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 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 6.1, providing the user with the full functionality including network access. 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 6.2 presents the high-level overview of the VirtualDrone framework. The key components in the framework are

- the virtual machine that sandboxes the full-featured but untrusted operations including the control and relevant applications (e.g., mission, imaging, networking),
- the virtualized sensors (e.g., inertial measurement unit), actuators (e.g., motors), communication channels (e.g., telemetry with the ground station) for use by the normal environment,
- the interface between the real I/O devices and the virtualized ones,
- the security and safety monitor that continuously monitors on the logical and physical behaviors of the system driven by the normal environment, and lastly
- the trusted controller for a robust backup and recovery.

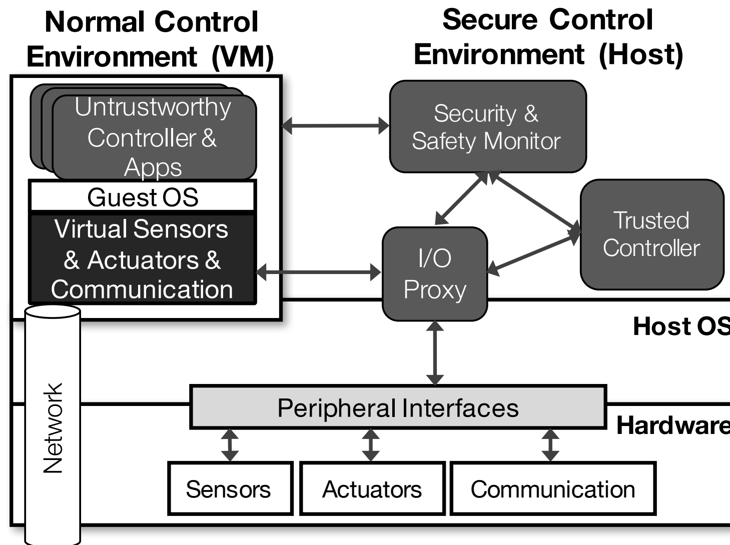


Figure 6.2: Overview of the VirtualDrone Framework.

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.

6.2.2 Assumptions and Adversary Model

In this work, the following assumptions are made:

- Since we utilize a virtualization technique for the isolation of potentially untrustworthy software components, the virtual machine monitor is our trusted computing base (TCB).
- The software components running in the secure control environment is static and trustworthy. This can be justified by the fact that, as previously explained, they go through a rigorous verification/certification process and require no or infrequent updates. If needed, any updates to the secure control environment require physical accesses to the hardware; over-the-air management is not allowed unless a secure communication channel is used.
- 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.

- We do not consider physical attacks [143, 88, 136], such as GPS spoofing, against the hardware. Such attacks can be handled by control-theoretic approaches such as [106, 102, 108].

6.2.3 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 [95]) 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 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

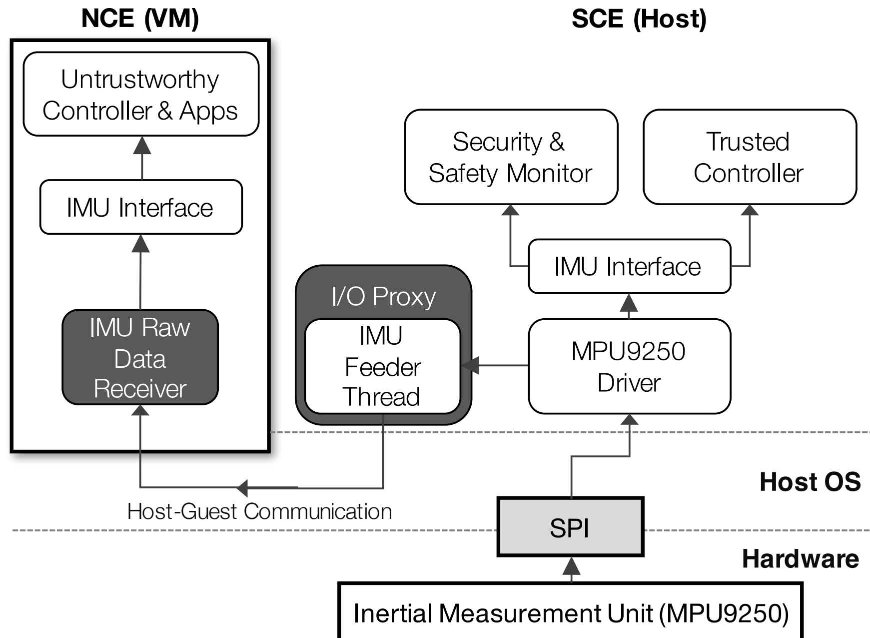


Figure 6.3: Sensor virtualization in the VirtualDrone framework.

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 6.2 in Section 6.3). 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 6.3 illustrates how an IMU (inertial measurement unit) sensor is virtualized in our prototype based on an

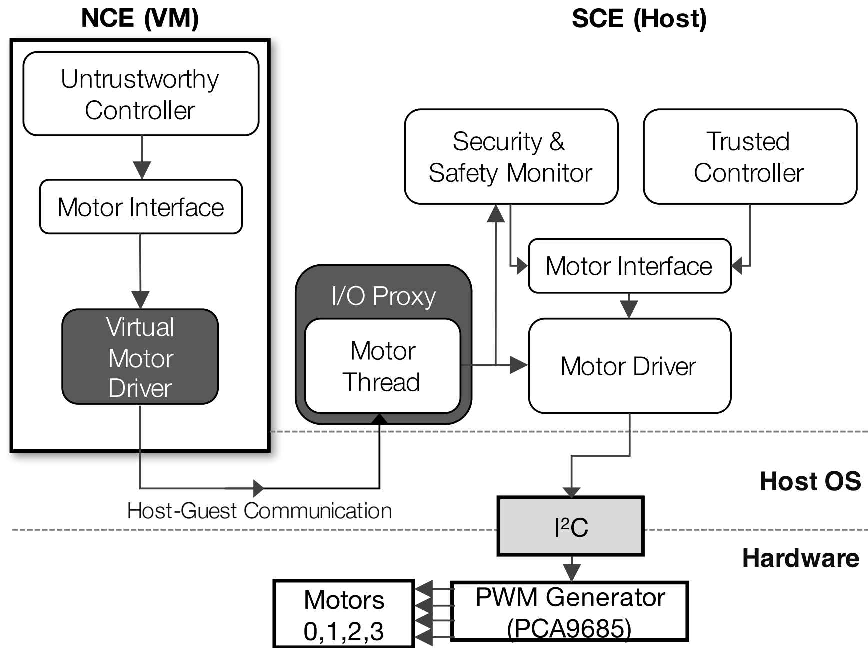


Figure 6.4: Actuator virtualization in the VirtualDrone framework.

open-source autopilot suite [2]. 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. Figure 6.4 shows how an actuator is virtualized in the VirtualDrone framework, which is similar to the sensor virtualization explained above. 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 PWM generator.

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 6.2, the framework 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.

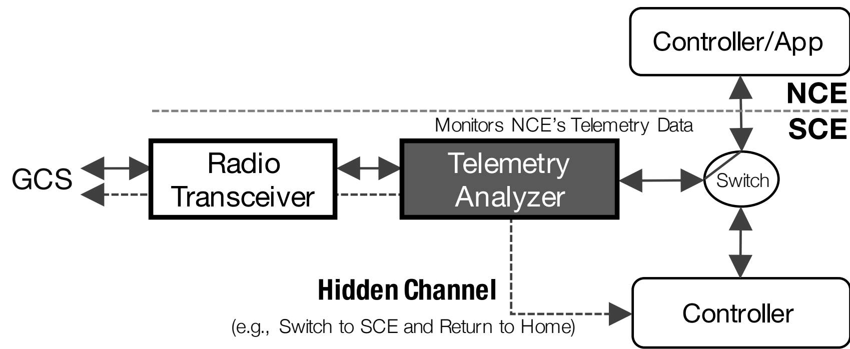


Figure 6.5: Telemetry virtualization creates a hidden communication channel between the SCE and the GCS.

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 6.5. 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 6.4.1; 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 6.12 in Section 6.4) is used by a human pilot to wire-lessly fly a vehicle. The flight controller on the vehicle is responsible for computation of PWM inputs to motor units to achieve the desired attitude set by the controller stick movements. 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.

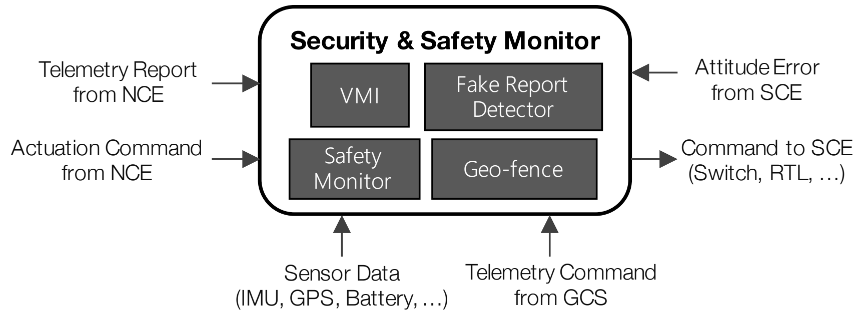


Figure 6.6: Example configuration of the security and safety monitor.

6.2.4 Security and Safety Monitoring

Figure 6.6 shows an example configuration of the security and safety monitor and data flow, based on the prototype implementation presented in Section 6.3. Notice from the figure (and also from Figure 6.3) that the monitor receives 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 [148, 83]. The monitor also analyzes the actuation outputs from the NCE, as shown in Figure 6.4, 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 [106, 102, 108] 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 6.5, 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) [65]. The monitoring module in the SCE can implement various VMI techniques to monitor the

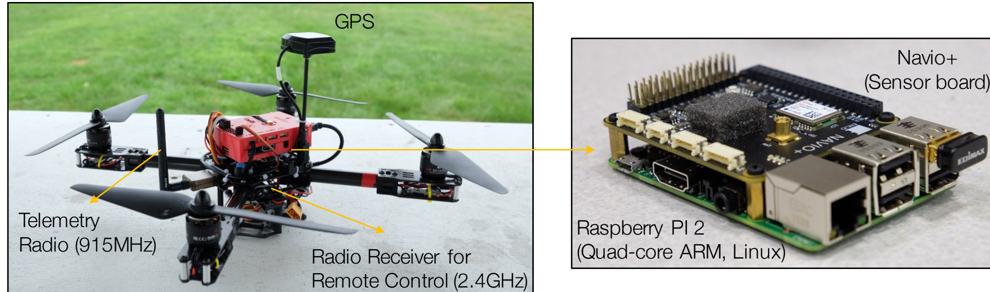


Figure 6.7: Quadcopter prototype implementation with Raspberry Pi 2 and Navio+ sensor boards.

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

6.3 Implementation

In this section, we present the implementation details for our prototype of the VirtualDrone framework on a quadcopter drone, shown in Figure 6.7, running an open-source autopilot on an embedded computing board.

6.3.1 Quadcopter Control

We built a quadcopter (shown in Figure 6.7) as our experimental platform. A quadcopter is a mechanically simple aerial vehicle which has four independent fixed-pitch propellers arranged in a cross type configuration and are driven by four brushless motors (motor-prop units). The motor-prop units generate four independent thrust vectors which provide lift. By precisely spinning the four motor-prop units at specific speeds, different orientations and flight maneuvers of the quadcopter can be achieved, e.g., to hover, fly forward/backward or left/right, and yaw. Quadcopters are aerodynamically unstable and their actuators, i.e., the motor-prop units, must be controlled directly by an on-board computer for stable flight. To maintain an accurate estimate of the vehicle's orientation and position, an Inertial Measurement Unit (IMU) sensor must be used to provide real-time attitude and acceleration data for the on-board computer. The flight maneuver commands may be supplied by input from a human operator using a radio controller (RC) or by the on-board computer which may compute the correct input according to a pre-programmed flight mission.

Table 6.1: Details about the VirtualDrone Prototype Implementation.

Component	Description
Sensor Board	Navio+
Platform	Raspberry PI 2 Model B
Host Processor	ARM Cortex-A7 Quad Core @ 900 MHz
Host Memory	1 GB
Host OS	Linux 3.18
VMM	QEMU Linaro v2.3.50
VM	ARM Versatile Express A15, 256 MB
Guest OS	Linux 4.3

6.3.2 System Implementation

We implemented the VirtualDrone framework on a Raspberry PI 2 Model B (RPI2) board [20], as depicted in Figure 6.8. Table 6.1 lists the details about the implementation. 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 [3]. 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 [19], an open-source virtual machine monitor. As explained in Section 6.2.2, QEMU, i.e., the VMM, is our trusted computing base (TCB). We created one VM[1] that emulates an ARM Versatile Express A15 board [25] (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 our prototype, we chose to use Linux as the host OS. However, it is desirable to use a formally verified OS (e.g., [67]) on the host instead of 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.

6.3.3 Autopilot

We stack Navio+ sensor board [10] on top of the Raspberry PI 2 (as shown in Figures 6.7 and 6.8) to provide various sensor data for flight control. The NCE (i.e., the virtual machine) runs the open-source ArduPilot (a.k.a. APM) [2] autopilot suite as the flight control software for our quadcopter drone. APM combines sensor data and RC flight

[1] It is more desirable to run multiple virtual machines that host applications requiring different levels of criticality.

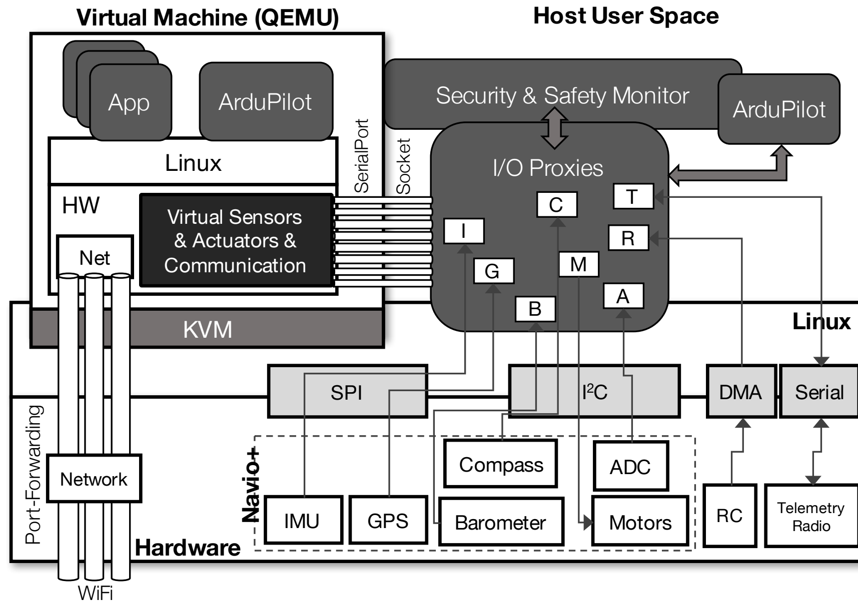


Figure 6.8: VirtualDrone implementation on the prototype.

Table 6.2: The rate and amount of data transfer between the secure and normal 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

maneuver commands to compute correct inputs for the four motor-prop units, which are sent by the actuator ports of the Navio+ board. APM is capable of maintaining the quadcopter's global position provided that a kind of global position estimation of the vehicle can be achieved (e.g., via GPS).

As shown in Figure 6.8, 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.

6.3.4 Virtualization

Sensors: The I/O proxy, shown in Figures 6.3 and 6.8, 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. The following describes how the sensors are virtualized in our prototype implementation, which are also summarized in Table 6.2

- 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). Hence, total 32 bits of data is fed to the virtual machine.

- 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. Hence, at a frequency of 1.67 Hz, a data of 40 bit (8 bit for channel ID and 32 bit for sampled data) is transferred to the virtual machine.

- 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, as explained in Section 6.2.3. Actuations are performed 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).

Host-Guest Communication: We use `QEMU`'s `virtio-serial`[2] for data transfer between the host (i.e., I/O proxy

[2]`https://fedoraproject.org/wiki/Features/VirtioSerial`

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 6.2. 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.[3] 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 6.5), 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).[4] MAVLink contains all interface functions to control the vehicle, monitor states, change parameters, etc. Each MAVLink message's size and frequency vary depending on the message type. The messages typically have small size (the maximum is 263 bytes) and low frequency (a few Hz).

6.4 Experiments

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

6.4.1 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 work, we consider a pessimistic scenario; the attacker can know the HW/SW configuration and even gain a root access to the NCE.

[3] 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.

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

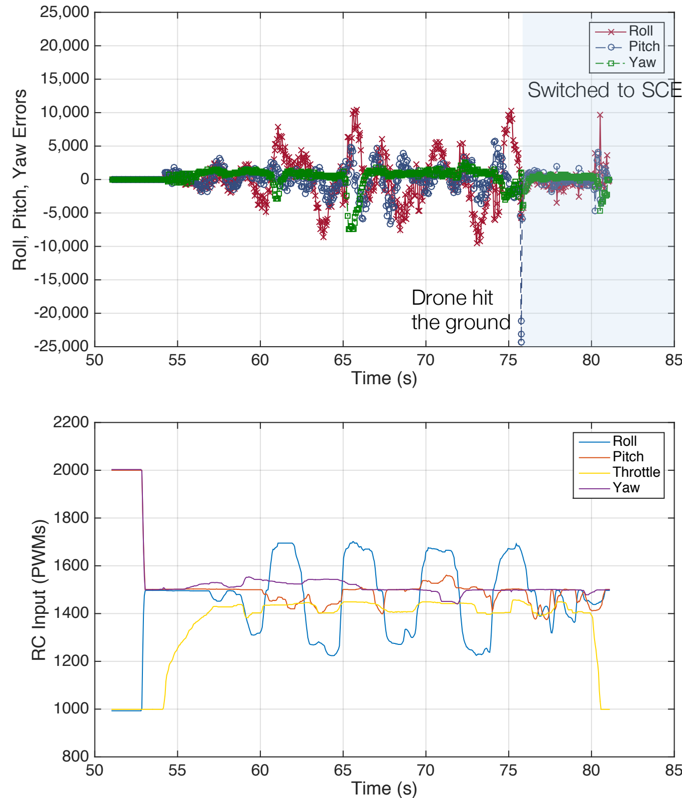


Figure 6.9: The roll, pitch, yaw errors (top) measured during the drone's extreme movements. The bottom plot shows the roll, pitch, throttle, and yaw targets set by the pilot using the remote controller. The errors (especially the pitch error in this particular experiment) grow beyond the threshold when the drone hit the ground.

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 (i) finds the APM autopilot process from the kernel's process list and then (ii) 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

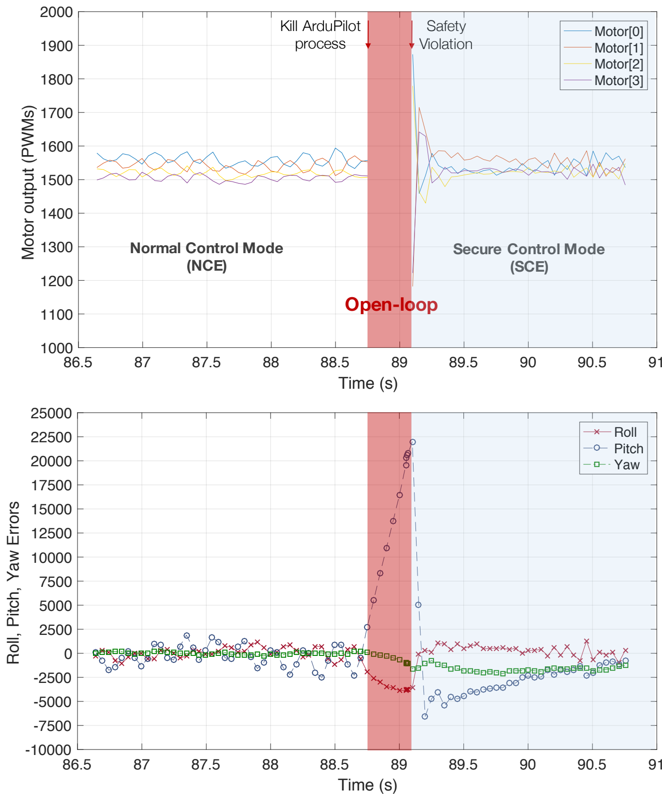


Figure 6.10: The motor outputs (top) and the roll, pitch, yaw errors (bottom) measured at the SCE. The attacker in NCE kills the flight controller, which leads the copter to an open-loop state. The SCE takes the control when the errors grow beyond the thresholds, after which the copter is stabilized.

to monitor the *true physical state* of the vehicle. We use the attitude control performance measured at the SCE.[5] 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 6.9, 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, 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. This approach can also detect other types of attacks that degrade flight performance. Upon a safety violation, the framework switches to the secure control mode.

Figure 6.10 shows the results of this experiment. While the drone was in the virtual control mode, the attacker

[5]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.

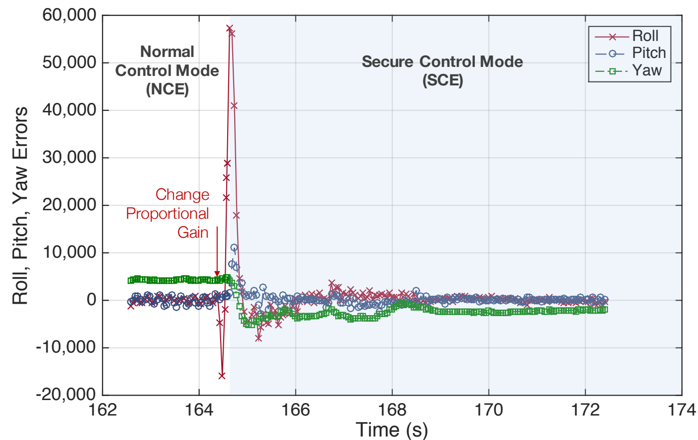


Figure 6.11: The roll, pitch, yaw errors when the proportional gain of the attitude controller is set to an abnormally high value. The safety module switches to the host control mode immediately upon the detection of the unstable physical state.

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 6.10 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.

For the thresholds on the errors, we used $\pm 20,000$ (rad/sec) for roll and pitch, and $\pm 10,000$ (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 6.9 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.

We also tested a scenario when the control parameters are modified during flight. Figure 6.11 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.

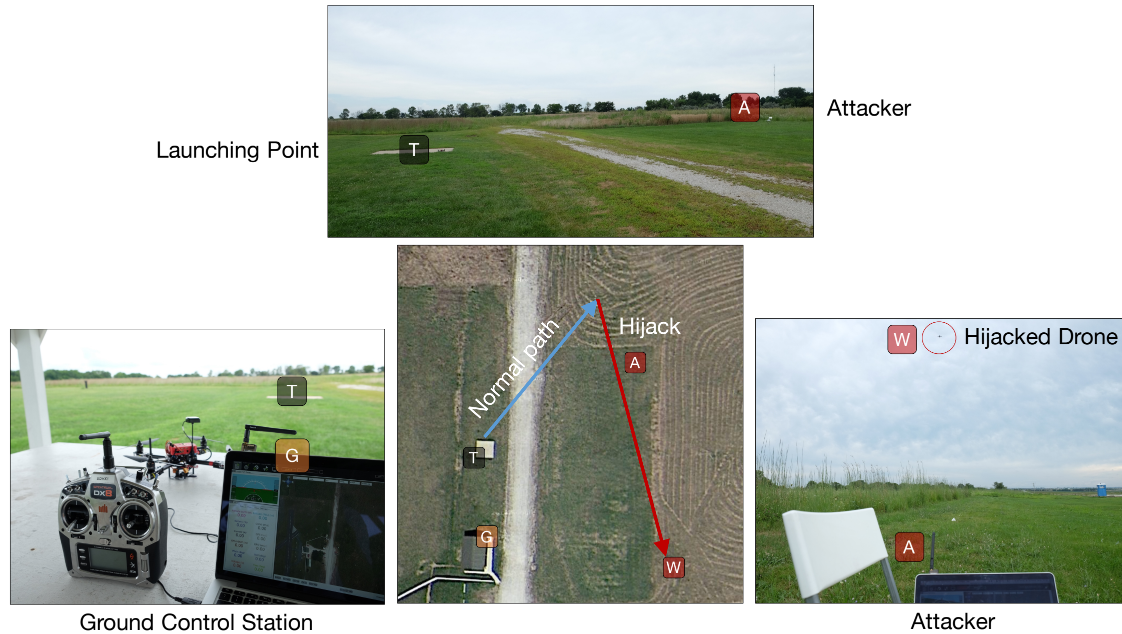


Figure 6.12: The attacker ('A') tries to hijack the drone flying along the normal path and to send it to a new waypoint ('W'). The attacker uses the same telemetry radio as the GCS and the drone, and unmodified APM Planner GCS.

Hijacking

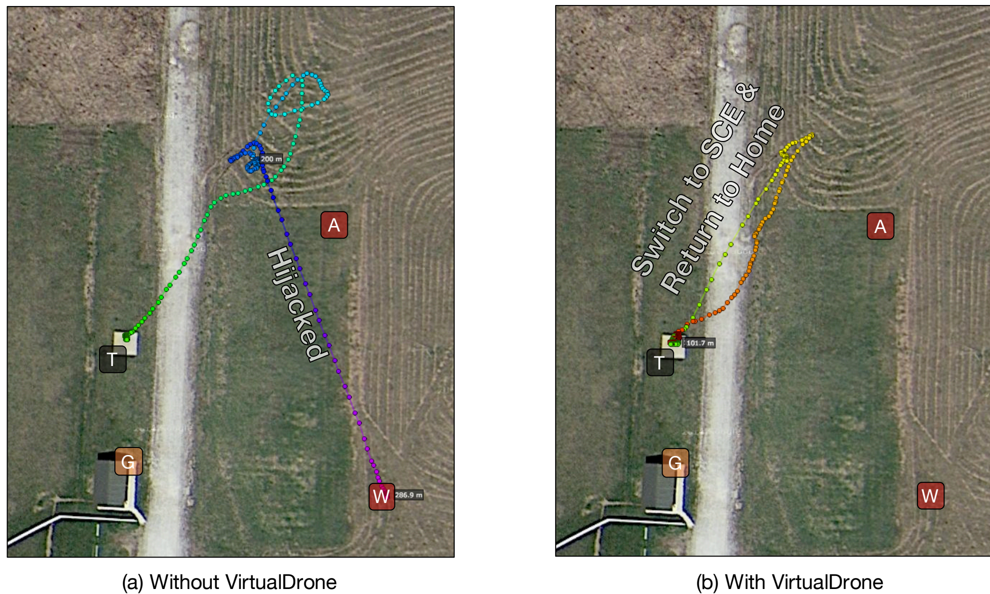
It has been demonstrated that it is possible to hijack a drone by exploiting the MAVLink protocol [30]. The idea is to send a command that sets a new flight plan through the telemetry channel. The telemetry radio pair of the drone and the ground control station (GCS) distinguish themselves from others by their unique NetID[6] 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 (915 MHz) 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.[7] Figure 6.12 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 GCS program (the stock APM Planner 2.0[8] with default settings). Figure 6.13(a) shows the flight trajectory of the drone when it was successfully hijacked by the attacker.

[6] We use radio that run the SiK firmware (`https://github.com/Dronecode/SiK`). In the SiK firmware, every aerial vehicle and ground station have assigned NetIDs. A NetID pair is used to simulate the binding of a specific ground station and a specific aerial vehicle.

[7] It is possible to find out the NetID used between the GCS and a target vehicle by modifying the firmware [30]. An attacker can even disable the NetID-pair checking routine in the SiK firmware source code to intercept any MAVlink messages via a radio system running at a known frequency band.

[8] `http://ardupilot.org/planner2/index.html`



(a) Without VirtualDrone (b) With VirtualDrone

Figure 6.13: (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.

While the drone itself cannot detect such a hijack attempt, the GCS can detect it because of *unexpected message exchange* initiated by the attacker. In order for a GCS to set waypoints for a vehicle, a series of MAVLink messages are exchanged. Figure 6.14(a) shows the MAVLink waypoint protocol.[9] 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 6.14(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.

Hence, we used MAVProxy[10] as the legitimate GCS and added the functionality that detects such unexpected messages. Upon a detection, the GCS commands the drone to return to where it was launched, as shown in Figure 6.13(b).

[9] `http://qgroundcontrol.org/mavlink/waypoint_protocol`
[10] `https://github.com/ArduPilot/MAVProxy`

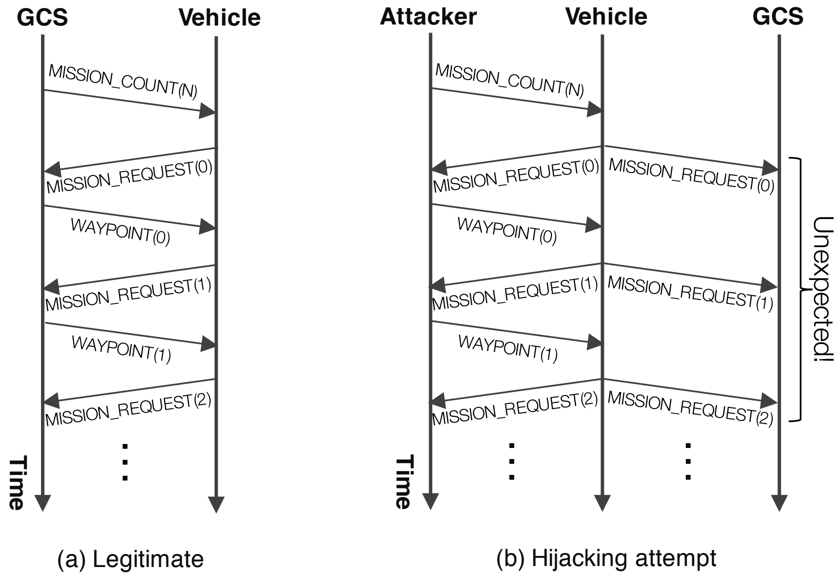


Figure 6.14: (a) MAVLink messages exchanged between the GCS and the vehicle for waypoint setup. (b) The legitimate GCS can detect the attacker's update on the vehicles route as the vehicle's responses are unexpected.

This takes advantage of the VirtualDrone's *virtual telemetry* explained in Section 6.2.3; 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 6.2.4, 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 a software simulator) to the GCS so that it looks as if it is flying on the planned path when it is not. 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.

Disabling 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 in-memory values.

```
ae65 0000 1000 0000 5746 0040 57f0 17c0
4f5d cb10 0000 01dc 0500 a062 f017 c058
5dcb 1000 0001 dc05 0080 68f0 17c0 4d5d   Waypoint 2
cb10 0000 01dc 0500 2069 f017 0037 5dcb   Waypoint 3
1000 0001 dc05 0080 5cf0 1700 4e5d cb10   Waypoint 4
0100 01dc 0500 205f f017 004f 5dcb 1000
0001 e803 0080 6ef0 1740 695d cb10 0000
01e8 0300 a034 f017 c045 5dcb 1000 0001
e803 00c0 45f0 1700 865d cb10 0000 01e8
0300 805d f017 c04d 5dcb 1000 0001 d007
0040 13f0 1740 8f5d cb10 0000 01d0 0700
2035 f017 c09c 5dcb 0000 0000 0000 0000
```

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 00a0 40d8 fb08 0001 0301 0000 00c8   Geo-fence enabled
4200 0096 4300 0000 4000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
```

Figure 6.15: 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.

```
nce_control@NCE:~$ sudo insmod attack_geofence.ko

Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:Found ArduCopter.elf [1898]

Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:Disabling geo fence

Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:88d6af80, 40a00000 8fbd8 10301 42c80000

Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:88d6af80, 40a00000 8fbd8 10300 42c80000
nce_control@NCE:~$
Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:Changing Waypoint 3

Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:88d83579, 17f06920 cb5d3700 1000010 800005dc

Message from syslogd@NCE at Oct 13 22:28:04 ...
 kernel:88d83579, 17f05d80 cb5d2e40 1000010 800005dc
```

Figure 6.16: The rootkit that disables the geo-fence and modifies the flight plan by manipulating the APM's memory.

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 finds the APM process, disables the geo-fence during a flight, and finally sends it into a no-fly zone by modifying the corresponding values *in the memory* – (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. 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 6.15 shows the memory dump of the waypoint list and geo-fence enable flag.

Our quadcopter drone was planned to fly through a path (0-1-2-3) in an autonomous mode as shown in Figure 6.17. During the flight, the attacker logged into the system (the NCE) through WiFi and launched the rootkit when the drone was flying toward Waypoint 2. Figure 6.16 shows how these values are modified by the rootkit. The new Waypoint

Figure 6.17: 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.

3 set by the rootkit is located inside a no-fly zone. 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 6.17.

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 presented 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 6.18 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

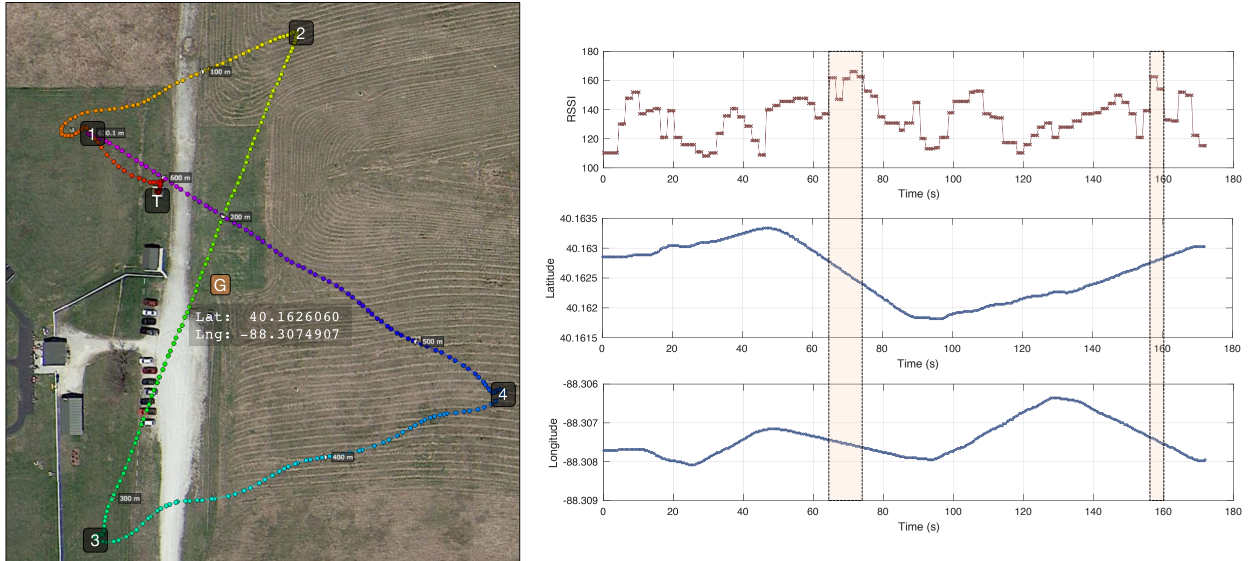


Figure 6.18: The drone flies through T-1-2-3-4-1. The graphs represent the RSSI (top), the latitude (middle), and the longitude (bottom). An attacker can estimate the location of the GCS by correlating the RSSI and the GPS information.

(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. Although an installation or invocation of rootkit does not immediately harm the system's safety, it is desirable to switch to the secure control mode as a preventive action. 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 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,

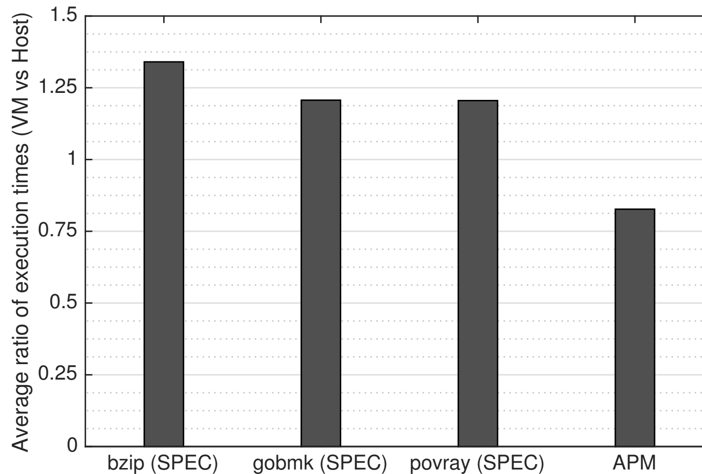


Figure 6.19: Average ratio of execution time when running on VM to the case of the host.

we were able to detect a known rootkit, `modhide1` [17], that hijacks open system 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` [15].

6.4.2 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 [106, 102, 108]. 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.

Timing Analysis

Virtualization typically imposes performance overhead, which may degrade the flight control performance. Hence, we measured the execution times of the APM when it runs directly on the host (using the stock APM) and on the virtual machine. Figure 6.19 shows the result. As a reference, we ran three benchmark applications randomly chosen from the SPEC2006 benchmark suite [72] and measured the execution times on the VM and the host. We executed each of

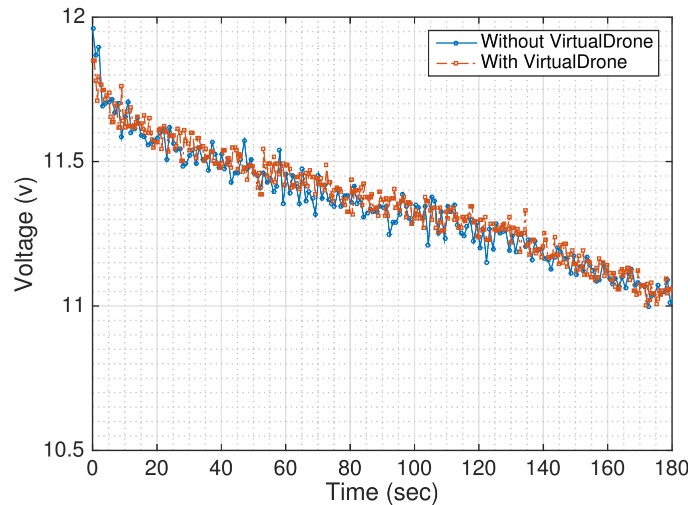


Figure 6.20: Voltage drop at the battery for 3-minutes of hovering with and without the VirtualDrone framework.

the benchmarks 10 times for each setup. Each benchmark takes minutes to execute once. For the APM, we measured 10,000 execution times. As can be seen from the left three bars in the plot, the average increases in the execution times due to virtualization are around $20-35\%$. However, the result of APM shows that its execution time actually decreases (about 17%) when running on the VM. This was mainly because it does not perform any real I/O directly with the sensors and actuators. Hence, the main control loop is less interfered with I/O threads whose only task is to read from virtual serial ports through which sensor data are fed by the SCE. The overheads could have been much higher if the hardware-assisted virtualization (KVM) were not available.

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.[11] We flew the drone without the VirtualDrone framework first, and then with the VirtualDrone framework.

Figure 6.20 shows the voltage drop at the battery for 3 minutes. The results show 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, and it ranged between

[11] It could fly longer, but we limited the time in order to avoid over-discharge of the battery.

114 Watts and 118 Watts in both cases.[12] Hence, the power consumption by any on-board computing in flight 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.

6.5 Conclusion

In this chapter, 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 software environment. Our prototype implementation requires a minimal effort to build the framework on an off-the-shelf computing board with open-source software stack. Through case studies, we demonstrated that the framework provides an integrated platform to handle various security threats that would otherwise have required potentially conflicting requirements. The framework also provides an experimental platform on which one can implement and evaluate the cyber-space behavior monitoring and intrusion/anomaly detection techniques (e.g., timing, memory, etc.) presented in the previous chapters.

Nevertheless, there still remains challenges for practical use of the framework. We used a full-featured autopilot for both the 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, integrating a high-assurance controller into the VirtualDrone framework would be an interesting extension.

[12] Even though the pilot tried to stabilize the drone at the fixed location in both flights, the manual control inevitably cause variances in drone movements. This could cause variances in the power consumption.

Chapter 7

Conclusion and Future Work

In this thesis, we presented hardware and software architectures for secure and dependable cyber-physical systems. Through certain modifications at the processor architecture and operating system levels, these frameworks not only make the process of monitoring application-level or system-wide behaviors efficient, but also enable trusted monitoring. Based on these architectures, we demonstrated novel uses of statistical learning techniques to model normal behaviors of CPS and to detect malicious activities. We also developed a virtualization-based attack-resilient architecture for safety-critical CPS.

As future CPS continues to expand the frontiers of automation, more challenges to security and safety will be emerging. The results presented in this dissertation open up interesting research problems for future study. One research direction could be a *multi-dimensional* behavior analysis. Each of the behavior monitoring techniques proposed in this dissertation looked at a certain behavior signal separately. The advances in modern computing technology enable adversaries to use sophisticated learning techniques to mimic such a low-dimensional cyber-space behavior. By combining multiple behavior signals, e.g., timing behavior with system call frequency distribution as briefly discussed in Section 4.5.5, we can significantly limit what attackers can do because it becomes much more difficult for them to stay within the normal behavior boundaries defined by the high-dimensional behavioral space. Similarly, we can also incorporate physical environmental factors into cyber-space behavior and vice versa. The physical behavior, although driven by the cyber component, is governed by the *law of physics*. Hence, by learning how physical behavior should react to particular dynamics of the cyber component, or vice versa, we can (a) detect a compromised state of the cyber component from physical behavior observation, and also (b) predict the physical consequence of a cyber-space behavior. This can enforce a strong security invariant on the system because the cyber and physical behaviors should be *mutually-consistent*, therefore significantly thwarting the attacker's attempt to imitate mutually-consistent behavior.

Cyber-physical systems are becoming more autonomous. They learn from environment, make intelligent decisions, and change their actions as the environment changes. Because of this *self-modifying behavior*, defining their normal behavior models now becomes much more difficult, if not impossible. In the current practices, we collect observational data by running the target system in normal situations, apply a learning technique, and then draw an empirical boundary of the normal behaviors. This will not work, especially for autonomous CPS, because a security

decision is made solely based on such empirical evidences. Hence, we need a systematic way to define their normal behavior boundaries. In the physical world, the law of physics and control theories enable deriving a mathematical model of normal physical behavior, i.e., *safety envelopes* (or *stability envelopes*). For future CPS, we need such a mechanism that can tell us whether the current cyber-space state is within secure boundaries. Having such a mathematical behavior model makes it possible to draw a sharp boundary between normal and malicious cyber-space states, and to detect zero-day attacks through a *mathematical reasoning process*, not based on unclear empirical evidences. To realize this, however, system techniques are crucial, because cyber-space behaviors rarely follow precise models, as opposed to physical-space behavior that follow the law of physics. Hence, it is a necessary direction of research to develop system techniques that turn true cyber-space behaviors into ones that can be mathematically modeled and represented. For example, let us consider the timing behavior model presented in Section 2.3. The timing behaviors of (even) real-time applications cannot be modeled by a precise statistical model as discussed. This necessitated a threshold test based on an empirical probability distribution of execution times. However, through an architectural support or a run-time level instrumentation and control, or an advanced compiler technique, we can turn such a complex temporal behavior into one that can be represented by a *parametric* statistics, for example, a Gaussian distribution. With such a precise mathematical behavior model, the detection process now becomes, for example, a statistical hypothesis test, not a threshold test. Furthermore, this will help reduce false classifications such as false positives or negatives, which are costly in cyber-physical systems.

Another interesting line of research is to make system behaviors more sensitive to security attacks. One way could be randomizing or modulating normal behaviors in a *provable* way, e.g., scheduling randomization [160]. We can also embed certain signatures in system behaviors, i.e., *behavior watermarking*, with the help of architecture or operating system supports. With these increased uncertainties, attackers will have to cope with the increased system dynamics. This will require attackers to use more resources such as time and memory to learn the system, find vulnerabilities, and launch attacks, which increases the level of complexity for would-be attackers and also the chance to detect such suspicious behavior. Lastly, it is also important to devise metrics to be able to quantify the security or privacy of a given system design, application sets, particular configuration, and so on.

References

[1] AMBA Specifications. http://www.arm.com/products/system-ip/amba-specifications.php.

[2] ArduPilot Autopilot Suite. http://www.ardupilot.org/.

[3] ARM Architecture virtualization extension. https://www.arm.com/products/processors/technologies/virtualization-extensions.php.

[4] ARM Cortex-A9 Processor. http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[5] Autosar 4.0. http://www.autosar.org/index.php?p=3&up=1&uup=0.

[6] BCM2836 ARM-local peripherals. https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2836/QA7_rev3.4.pdf.

[7] Buildroot. http://git.buildroot.net/buildroot/.

[8] e500mc Core Reference Manual. http://cache.freescale.com/files/32bit/doc/ref_manual/E500MCRM.pdf.

[9] EEMBC AutoBench Suite. http://www.eembc.org.

[10] Emlid NAVIO+. https://docs.emlid.com/navio/.

[11] Freescale QorIQ P4080 Processor. http://www.freescale.com/webapp/sps/site/prod_summary.jsp?code=P4080.

[12] Freescale's Embedded Hypervisor for QorIQ P4 Series Communications Platform. http://cache.freescale.com/files/32bit/doc/white_paper/EMBEDDED_HYPERVISOR.pdf?fsrch=1&sr=2.

[13] Hijacking system calls with loadable kernel modules. http://r00tkit.me/?p=46.

[14] LEON3 Processor. http://www.gaisler.com/index.php/products/processors/leon3.

[15] LibVMI. http://libvmi.com/.

[16] Linux/ARM shellcode - Disable ASLR Security. http://shell-storm.org/shellcode/files/shellcode-669.php.

[17] modhide1 Rootkit. http://packetstormsecurity.com/files/favorite/24880/.

[18] Motion. http://www.lavrsen.dk/foswiki/bin/view/Motion/WebHome.

[19] QEMU. http://wiki.qemu.org/.

[20] Raspberry PI 2 Model B. https://www.raspberrypi.org/products/raspberry-pi-2-model-b/.

[21] SANS Institute - Kernel Rootkits. http://www.sans.org/reading-room/whitepapers/threats/kernel-rootkits-449.

[22] Simulink. http://www.mathworks.com/products/simulink.

[23] SNARE: System iNtrusion Analysis and Reporting Environments. http://http://www.intersectalliance.com/.

[24] Suterusu Rootkit. http://poppopret.org/2013/01/07.

[25] Versatile Express Product Family. http://www.arm.com/products/tools/development-boards/versatile-express.

[26] Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit. http://www.xilinx.com/products/boards-and-kits/ek-z7-zc702-g.html.

[27] Cyber-attack concerns raised over Boeing 787 chip's 'back door'. *The Guardian*, May 2012. https://www.theguardian.com/technology/2012/may/29/cyber-attack-concerns-boeing-chip.

[28] 3DR's Solo Drone Boasts Dual Linux Computers Running Dronecode, Apr 2015. https://www.linux.com/news/3drs-solo-drone-boasts-dual-linux-computers-running-dronecode.

[29] A hacker developed Maldrone, the first malware for drones. *Security Affairs*, Jan 2015. http://securityaffairs.co/wordpress/32767/hacking/maldrone-malware-for-drones.html.

[30] Hijacking drones with a mavlink exploit. Oct 2015. http://diydrones.com/profiles/blogs/hijacking-quadcopters-with-a-mavlink-exploit.

[31] Jeep Hacking 101. *IEEE Spectrum*, Aug 2015. http://spectrum.ieee.org/cars-that-think/transportation/systems/jeep-hacking-101.

[32] Qualcomm Goes Ubuntu for Drone Reference Platform, Sep 2015. https://www.linux.com/news/qualcomm-goes-ubuntu-drone-reference-platform.

[33] Watch GPS Attacks That Can Kill DJI Drones Or Bypass White House Ban. *Forbes*, Aug 2015. http://www.forbes.com/sites/thomasbrewster/2015/08/08/qihoo-hacks-drone-gps/#26431a2853fe.

[34] Intel Aero Compute Board, 2016. http://www.intel.com/content/www/us/en/technology-innovation/aerial-technology-overview.html.

[35] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13(1):4:1–4:40, Nov. 2009.

[36] A. Aiyer, K. P. Pyun, Y. zong Huang, D. B. OBrien, and R. M. Gray. Lloyd clustering of gauss mixture models for image compression and classification. *Signal Processing: Image Communication*, 20(5):459 – 485, 2005.

[37] T. W. Anderson and D. A. Darling. Asymptotic theory of certain "goodness of fit" criteria based on stochastic processes. *The Annals of Mathematical Statistics*, 23(2):193–212, 1952.

[38] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *Proceedings of the International Conference on Real-Time and Network Systems*, May 2006.

[39] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[40] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *Proceedings of the ACM SIGCOMM Workshop on Internet Measurment*, 2002.

[41] C. Beecks, A. Ivanescu, S. Kirchhoff, and T. Seidl. Modeling image similarity by gaussian mixture models and the signature quadratic form distance. In *Proceedings of the IEEE International Conference on Computer Vision*, 2011.

[42] R. Begleiter, R. El-yaniv, and G. Yona. On prediction using variable order markov models. *Journal of Artificial Intelligence Research*, 22:385–421, 2004.

[43] J. Bigham, D. Gamez, and N. Lu. Safeguarding scada systems with anomaly detection. In *Proceedings of the International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security*, 2003.

[44] D. M. Blei. Probabilistic topic models. *Commun. ACM*, 55(4):77–84, Apr. 2012.

[45] S. Bratus, M. E. Locasto, A. Ramaswamy, and S. W. Smith. Vm-based security overkill: A lament for applied systems security research. In *Proceedings of the Workshop on New Security Paradigms*, pages 51–60, 2010.

[46] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *Proceedings of the ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, 2011.

[47] V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection for discrete sequences: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 24(5):823–839, 2012.

[48] S. Chen, B. Falsafi, P. B. Gibbons, M. Kozuch, T. C. Mowry, R. Teodorescu, A. Ailamaki, L. Fix, G. R. Ganger, B. Lin, and S. W. Schlosser. Log-based architectures for general-purpose monitoring of deployed code. In *Proceedings of the workshop on Architectural and system support for improving software dependability*, 2006.

[49] S. Chen, M. Kozuch, T. Strigkos, B. Falsafi, P. B. Gibbons, T. C. Mowry, V. Ramachandran, O. Ruwase, M. Ryan, and E. Vlachos. Flexible hardware acceleration for instruction-grain program monitoring. In *Proceedings of the International Symposium on Computer Architecture*, 2008.

[50] S. Cheung, B. Dutertre, M. Fong, U. Lindqvist, K. Skinner, and A. Valdes. Using Model-based Intrusion Detection for SCADA Networks. In *Proceedings of the SCADA Security Scientific Symposium*, 2007.

[51] J. Choi and E. Amir. Lifted relational variational inference. In *Proceedings of the Conference on Uncertainty in Artificial Intelligence*, 2012.

[52] J. Criswell, N. Dautenhahn, and V. Adve. Kcofi: Complete control-flow integrity for commodity operating system kernels. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2014.

[53] J. Criswell, N. Dautenhahn, and V. Adve. Virtual ghost: Protecting applications from hostile operating systems. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[54] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society, Series B*, 39(1):1–38, 1977.

[55] D. Y. Deng, D. Lo, G. Malysa, S. Schneider, and G. E. Suh. Flexible and efficient instruction-grained run-time monitoring using on-chip reconfigurable fabric. In *Proceedings of the IEEE/ACM International Symposium on Microarchitecture*, pages 137–148, 2010.

[56] D. E. Denning. An intrusion-detection model. *IEEE Trans. Softw. Eng.*, 13(2):222–232, Feb. 1987.

[57] B. Dolan-Gavitt, T. Leek, J. Hodosh, and W. Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the ACM Conference on Computer Communications Security*, 2013.

[58] P. Düssel, C. Gehl, P. Laskov, J.-U. Buß er, C. Störmann, and J. Kästner. Cyber-critical infrastructure protection using real-time payload-based anomaly detection. In *Proceedings of the International Conference on Critical Information Infrastructures Security*, 2010.

[59] D. Eastlake, 3rd and P. Jones. Us secure hash algorithm 1 (sha1), 2001.

[60] V. A. Epanechnikov. Non-parametric estimation of a multivariate probability density. *Theory Probab. Appl*, 14(1):153–158, 1969.

[61] E. Eskin. Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of DARPA Information Survivabilty Conference and Exposition II*, 2001.

[62] M. A. T. Figueiredo and A. Jain. Unsupervised learning of finite mixture models. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 24(3):381–396, 2002.

[63] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1996.

[64] D. D. Gajski, S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded system design: modeling, synthesis and verification*. Springer Science & Business Media, 2009.

[65] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *Proceedings of Network and Distributed Systems Security Symposium*, 2003.

[66] I. Garitano, R. Uribeetxeberria, and U. Zurutuza. A review of scada anomaly detection systems. volume 87 of *Advances in Soft Computing*, pages 357–366, 2011.

[67] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent os kernels. In *Proceedings of the USENIX Conference on Operating Systems Design and Implementation*, 2016.

[68] Y. Gu, A. McCallum, and D. Towsley. Detecting anomalies in network traffic using maximum entropy estimation. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement*, 2005.

[69] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. Mibench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE Annual Workshop on Workload Characterization*, 2001.

[70] D. Hadiosmanović, D. Bolzoni, S. Etalle, and P. H. Hartel. Challenges and opportunities in securing industrial control systems. In *Proceedings of the IEEE Workshop on Complexity in Engineering*, 2012.

[71] M. Handley, V. Paxson, and C. Kreibich. Network intrusion detection: evasion, traffic normalization, and end-to-end protocol semantics. In *Proceedings of the conference on USENIX Security Symposium*, 2001.

[72] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Comput. Archit. News*, 34(4):1–17, Sept. 2006.

[73] J. L. Henning. SPEC CPU2006 memory footprint. *SIGARCH Comput. Archit. News*, 35(1):84–89, Mar. 2007.

[74] G. E. Hinton, S. Osindero, and Y.-W. Teh. A fast learning algorithm for deep belief nets. *Neural Comput.*, 18(7):1527–1554, 2006.

[75] A. Ho, M. Fetterman, C. Clark, A. Warfield, and S. Hand. Practical taint-based protection using demand emulation. In *Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems*, 2006.

[76] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *J. Comput. Secur.*, 6(3):151–180, 1998.

[77] W. Hu, Y. Liao, and V. R. Vemuri. Robust anomaly detection using support vector machines. In *Proceedings of the International Conference on Machine Learning*, 2003.

[78] W.-M. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: an effective technique for vliw and superscalar compilation. *J. Supercomput.*, 7:229–248, 1993.

[79] D. Jang, H. Lee, M. Kim, D. Kim, D. Kim, and B. B. Kang. Atra: Address translation redirection attack against hardware-based external monitors. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2014.

[80] A. Y. Javaid, W. Sun, V. K. Devabhaktuni, and M. Alam. Cyber security threat analysis and modeling of an unmanned aerial vehicle system. In *Proceedings of the IEEE Conference on Technologies for Homeland Security*, 2012.

[81] S. Jha, K. Tan, and R. A. Maxion. Markov chains, classifiers, and intrusion detection. In *Proceedings of the IEEE workshop on Computer Security Foundations*, 2001.

[82] X. Jiang, X. Wang, and D. Xu. Stealthy malware detection through vmm-based "out-of-the-box" semantic view reconstruction. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[83] T. T. Johnson, S. Bak, M. Caccamo, and L. Sha. Real-time reachability for verified simplex design. *ACM Trans. Embed. Comput. Syst.*, 15(2):26:1–26:27, Feb. 2016.

[84] I. Jolliffe. *Principal Component Analysis*. Springer Series in Statistics, 2002.

[85] M. C. Jones, J. S. Marron, and S. J. Sheather. A brief survey of bandwidth selection for density estimation. *Journal of the American Statistical Association*, 91(433):401–407, 1996.

[86] S. T. Jones, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau. Antfarm: Tracking processes in a virtual machine environment. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[87] H. Kannan, M. Dalton, and C. Kozyrakis. Decoupling dynamic information flow tracking with a dedicated coprocessor. In *Proceedings of the IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 105–114, 2009.

[88] A. J. Kerns, D. P. Shepard, J. A. Bhatti, , and T. E. Humphreys. Unmanned Aircraft Capture and Control Via GPS Spoofing. *Journal of Field Robotics*, 31, 2014.

[89] E. M. Knorr and R. T. Ng. A unified notion of outliers: Properties and computation. In *Proceedings of the International Conference on Knowledge Discovery and Data Mining*, 1997.

[90] P. Koeberl, S. Schulz, A.-R. Sadeghi, and V. Varadharajan. Trustlite: A security architecture for tiny embedded devices. In *Proceedings of the European Conference on Computer Systems*, 2014.

[91] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage. Experimental security analysis of a modern automobile. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 447–462, 2010.

[92] Y. Kwon, A. M. Dunn, M. Z. Lee, O. S. Hofmann, Y. Xu, and E. Witchel. Sego: Pervasive trusted metadata for efficiently verified untrusted system services. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2016.

[93] B. Lickly, I. Liu, S. Kim, H. D. Patel, S. A. Edwards, and E. A. Lee. Predictable programming on a precision timed architecture. In *Proceedings of International Conference on Compilers, Architecture, and Synthesis from Embedded Systems*, 2008.

[94] A. Likas, N. Vlassis, and J. J. Verbeek. The global k-means clustering algorithm. *Pattern Recognition*, 36(2):451 – 461, 2003.

[95] J. Liu, W. Huang, B. Abali, and D. K. Panda. High performance vmm-bypass i/o in virtual machines. In *Proceedings of the USENIX Annual Technical Conference*, 2006.

[96] S. Lloyd. Least squares quantization in PCM. *IEEE Transactions on Information Theory*, 28(2):129–137, Mar. 1982.

[97] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 7(4):381–395, 2010.

[98] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hllberg, J. Hgberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, 2002.

[99] P. C. Mahalanobis. On the generalized distance in statistics. *Proceedings of the National Institute of Sciences (Calcutta)*, 2:49–55, 1936.

[100] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the workshop on New security paradigms*, 2000.

[101] J. M. McCune, Y. Li, N. Qu, Z. Zhou, A. Datta, V. Gligor, and A. Perrig. Trustvisor: Efficient tcb reduction and attestation. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[102] Y. Mo and B. Sinopoli. Secure estimation in the presence of integrity attacks. *IEEE Transactions on Automatic Control*, 60(4):1145–1151, 2015.

[103] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo. S3A: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems. In *Proceedings of the ACM International Conference on High Confidence Networked Systems*, 2013.

[104] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomalous system call detection. *ACM Trans. Inf. Syst. Secur.*, 9(1):61–93, Feb. 2006.

[105] J. Noorman, P. Agten, W. Daniels, R. Strackx, A. Van Herrewege, C. Huygens, B. Preneel, I. Verbauwhede, and F. Piessens. Sancus: Low-cost trustworthy extensible networked devices with a zero-software trusted computing base. In *Proceedings of the USENIX Conference on Security*, 2013.

[106] M. Pajic, J. Weimer, N. Bezzo, P. Tabuada, O. Sokolsky, I. Lee, and G. J. Pappas. Robustness of attack-resilient state estimators. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*, 2014.

[107] M. Paolieri, E. Quiñones, F. J. Cazorla, G. Bernat, and M. Valero. Hardware support for wcet analysis of hard real-time multicore systems. In *Proceedings of IEEE/ACM International Symposium on Computer Architecture*, 2009.

[108] J. Park, R. Ivanov, J. Weimer, M. Pajic, and I. Lee. Sensor attack detection in the presence of transient faults. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*, 2015.

[109] P. Parkinson. Safety, security and multicore. In *Proceedings of Safety-Critical Systems Symposium*, 2011.

[110] G. Parmer and R. Wes. Hijack: Taking control of cots systems for real-time user-level services. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2007.

[111] E. Parzen. On estimation of a probability density function and mode. *The Annals of Mathematical Statistics*, 33(3):1065–1076, 1962.

[112] B. D. Payne, M. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. In *Proceedings of Annual Computer Security Applications Conference*, 2007.

[113] B. D. Payne, M. Carbone, M. Sharif, and W. Lee. Lares: An architecture for secure active monitoring using virtualization. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2008.

[114] S. Peisert, M. Bishop, S. Karin, and K. Marzullo. Analysis of computer intrusions using sequences of function calls. *IEEE Trans. Dependable Secur. Comput.*, 4(2):137–150, 2007.

[115] G. Pék, A. Lanzi, A. Srivastava, D. Balzarotti, A. Francillon, and C. Neumann. On the feasibility of software attacks on commodity virtual machine monitors via direct device assignment. In *Proceedings of the ACM Symposium on Information, Computer and Communications Security*, 2014.

[116] A. G. Pennington, J. D. Strunk, J. L. Griffin, C. A. N. Soules, G. R. Goodson, and G. R. Ganger. Storage-based intrusion detection: Watching storage activity for suspicious behavior. In *Proceedings of the Conference on USENIX Security Symposium*, 2003.

[117] H. Permuter, J. Francos, and I. Jermyn. A study of gaussian mixture models of color and texture features for image classification and segmentation. *Pattern Recognition*, 39(4):695 – 706, 2006.

[118] N. L. Petroni, Jr., T. Fraser, J. Molina, and W. A. Arbaugh. Copilot - a coprocessor-based kernel runtime integrity monitor. In *Proceedings of the Conference on USENIX Security Symposium*, 2004.

[119] N. L. Petroni, Jr. and M. Hicks. Automated detection of persistent kernel control-flow attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[120] J. Pfoh, C. Schneider, and C. Eckert. Nitro: hardware-based system call tracing for virtual machines. In *Proceedings of the International conference on Advances in information and computer security*, 2011.

[121] plaguez. Weakening the linux kernel. *Phrack*, 8(52), 1998.

[122] J.-S. Pleban, R. Band, and R. Creutzburg. Hacking and securing the ar.drone 2.0 quadcopter: investigations for improving the security of a toy. In *Proceedings of the SPIE Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications*, 2014.

[123] R. Sailer, X. Zhang, T. Jaeger, and L. Van Doorn. Design and implementation of a tcg-based integrity measurement architecture. In *Proceedings of the USENIX Security Symposium*, 2004.

[124] M. Salem, M. Crowley, and S. Fischmeister. Anomaly detection using inter-arrival curves for real-time systems. In *Proceedings of the Euromicro Conference on Real-Time Systems*, 2016.

[125] N. Santos, H. Raj, S. Saroiu, and A. Wolman. Using arm trustzone to build a trusted language runtime for mobile applications. In *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, 2014.

[126] M. Schoeberl and P. Puschner. Is chip-multiprocessing the end of real-time scheduling? In *Proceedings of International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.

[127] A. Seshadri, M. Luk, N. Qu, and A. Perrig. Secvisor: A tiny hypervisor to provide lifetime kernel code integrity for commodity oses. In *Proceedings of the ACM Symposium on Operating Systems Principles*, 2007.

[128] L. Sha. Using simplicity to control complexity. *IEEE Softw.*, 18(4):20–28, 2001.

[129] H. Shacham. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[130] M. I. Sharif, W. Lee, W. Cui, and A. Lanzi. Secure in-vm monitoring using hardware virtualization. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[131] D. Shepard, J. Bhatti, and T. Humphreys. Drone hack: Spoofing attack demonstration on a civilian unmanned aerial vehicle. *GPS World*, Aug 2012.

[132] W. Shi, H.-H. S. Lee, L. Falk, and M. Ghosh. An integrated framework for dependable and revivable architectures using multicore processors. In *Proceedings of the International Symposium on Computer Architecture*, 2006.

[133] C. Sinclair, L. Pierce, and S. Matzner. An application of machine learning to network intrusion detection. In *Proceedings of the Computer Security Applications Conference*, 1999.

[134] L. Sirovich and M. Kirby. Low-dimensional procedure for the characterization of human faces. *Journal of the Optical Society of America A*, 4(3):519–524, 1987.

[135] R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the IEEE Symposium on Security and Privacy*, 2010.

[136] Y. Son, H. Shin, D. Kim, Y. Park, J. Noh, K. Choi, J. Choi, and Y. Kim. Rocking drones with intentional sound noise on gyroscopic sensors. In *Proceedings of the USENIX Conference on Security Symposium*, 2015.

[137] C. SPARC International, Inc. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., 1992.

[138] D. Srinivasan, Z. Wang, X. Jiang, and D. Xu. Process out-grafting: An efficient "out-of-vm" approach for fine-grained process execution monitoring. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[139] R. Strackx, F. Piessens, and B. Preneel. Efficient isolation of trusted subsystems in embedded systems. In *Security and Privacy in Communication Networks*, volume 50 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pages 344–361. 2010.

[140] P. Sun, S. Chawla, and B. Arunasalam. Mining for outliers in sequential databases. In *the SIAM International Conference on Data Mining*, 2006.

[141] J. Szefer, E. Keller, R. B. Lee, and J. Rexford. Eliminating the hypervisor attack surface for a more secure cloud. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[142] H. Teso. Aicraft hacking. In *Proceedings of the Annual HITB Security Conference in Europe*, 2013.

[143] N. O. Tippenhauer, C. Pöpper, K. B. Rasmussen, and S. Capkun. On the requirements for successful gps spoofing attacks. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2011.

[144] M. Turk and A. Pentland. Face recognition using eigenfaces. In *Proceedings of the IEEE Coference on Computer Vision and Pattern Recognition*, 1991.

[145] US-CERT. ICSA-10-272-01: Primary stuxnet indicators. Aug. 2010.

[146] A. Valdes and S. Cheung. Communication pattern anomaly detection in process control system. In *IEEE International Conference on Technologies for Homeland Security*, 2009.

[147] D. Wagner and P. Soto. Mimicry attacks on host-based intrusion detection systems. In *Proceedings of the ACM conference on Computer and communications security*, 2002.

[148] X. Wang, N. Hovakimyan, and L. Sha. L1simplex: Fault-tolerant control of cyber-physical systems. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*, 2013.

[149] Z. Wang, X. Jiang, W. Cui, and P. Ning. Countering kernel rootkits with lightweight hook protection. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2009.

[150] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusion using system calls: alternative data models. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1999.

[151] J. Wei, B. Payne, J. Giffin, and C. Pu. Soft-timer driven transient kernel control flow attacks and defense. In *Proceedings of the Annual Computer Security Applications Conference*, 2008.

[152] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[153] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves. Implementing embedded security on dual-virtual-cpu systems. *IEEE Des. Test*, 24(6):582–591, Nov. 2007.

[154] R. Wojtczuk. Subverting the xen hypervisor - xen 0wning trilogy part i. *Black Hat USA*, 2008.

[155] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the ACM Conference on Computer and Communications Security*, 2007.

[156] M.-K. Yoon, M. Christodorescu, L. Sha, and S. Mohan. The DragonBeam Framework: Hardware-protected security modules for in-place intrusion detection. In *Proceedings of the ACM International Systems and Storage Conference*, 2016.

[157] M.-K. Yoon and G. Ciocarlie. Communication pattern monitoring: Improving the utility of anomaly detection for industrial control systems. In *NDSS Workshop on Security of Emerging Networking Technologies*, 2014.

[158] M.-K. Yoon, J.-E. Kim, and L. Sha. Optimizing tunable wcet with shared resource allocation and arbitration in hard real-time multicore systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, 2011.

[159] M.-K. Yoon, B. Liu, N. Hovakimyan, and L. Sha. VirtualDrone: Virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*, 2017.

[160] M.-K. Yoon, S. Mohan, C.-Y. Chen, and L. Sha. TaskShuffler: A schedule randomization protocol for obfuscation against timing inference attacks in real-time systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2016.

[161] M.-K. Yoon, S. Mohan, J. Choi, M. Christodorescu, and L. Sha. Learning execution contexts from system call distribution for anomaly detection in smart embedded system. In *Proceedings of the ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, 2017.

[162] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha. SecureCore: A multicore-based intrusion detection architecture for real-time embedded systems. In *Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium*, 2013.

[163] M.-K. Yoon, S. Mohan, J. Choi, and L. Sha. Memory Heat Map: Anomaly detection in real-time embedded systems using memory behavior. In *Proceedings of the ACM/EDAC/IEEE Design Automation Conference*, 2015.

[164] M. M. Z. Zadeh, M. Salem, N. Kumar, G. Cutulenco, and S. Fischmeister. SiPTA: Signal processing for trace-based anomaly detection. In *Proceedings of the International Conference on Embedded Software*, 2014.

[165] Y. Zhou, X. Wang, Y. Chen, and Z. Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2014.

[166] B. Zhu and S. Sastry. SCADA-specific Intrusion Detection/Prevention Systems: A Survey and Taxonomy. In *Proceedings of the 1st Workshop on Secure Control Systems*, 2010.

[167] S. Zhuravlev, S. Blagodurov, and A. Fedorova. Addressing shared resource contention in multicore processors via scheduling. In *Proceedings of International Conference on Architectural Support for Programming Languages and Operating Systems*, 2010.

[168] C. Zimmer, B. Bhatt, F. Mueller, and S. Mohan. Time-based intrusion detection in cyber-physical systems. In *Proceedings of the ACM/IEEE International Conference on Cyber-Physical Systems*, 2010.