DETECTING SOFTWARE ATTACKS ON EMBEDDED IOT DEVICES

Vom Fachbereich Informatik
der Technischen Universität Darmstadt
genehmigte

DISSERTATION

zur Erlangung des akademischen Grades
Doktor-Ingenieur (Dr.-Ing.)
von

SEEMA KUMAR
geboren in Bangalore, India

Referent: Prof. Dr. Max Mühlhäuser
Referent: Prof. Dr. Patrick Eugster

Tag der Einreichung: 6. Dezember 2022
Tag der Prüfung: 23. Januar 2023

Darmstadt 2023

Detecting Software Attacks on Embedded IoT Devices

Submitted doctoral thesis by Seema Kumar

Reviewer: Prof. Dr. Max Mühlhäuser
Reviewer: Prof. Dr. Patrick Eugster

Date of submission: December 6, 2022
Date of defense: January 23, 2023

Year of publication of the dissertation on TUprints: 2023
Darmstadt - D17

Department of Computer Science
Darmstadt, Technische Universität Darmstadt

Please cite this document as:
URI: https://tuprints.ulb.tu-darmstadt.de/id/eprint/22933
URN: urn:nbn:de:tuda-tuprints-229332

This document is provided by TUprints, the e-publishing-service of TU Darmstadt.
https://tuprints.ulb.tu-darmstadt.de
tuprints@ulb.tu-darmstadt.de

Licensed under *CC BY-SA 4.0 International (Attribution-ShareAlike)*
https://creativecommons.org/licenses/by-sa/4.0/deed.en

Dedicated to Poorna

ABSTRACT

Internet of Things (IoT) applications are being rapidly deployed in the context of smart homes, automotive vehicles, smart factories, and many more. In these applications, embedded devices are widely used as sensors, actuators, or edge nodes. The embedded devices operate distinctively on a task or interact with each other to collectively perform certain tasks. In general, increase in Internet-connected *things* has made embedded devices an attractive target for various cyber attacks, where an attacker gains access and control remote devices for malicious activities. These IoT devices could be exploited by an attacker to compromise the security of victim's platform without requiring any physical hardware access.

In order to detect such software attacks and ensure a reliable and trustworthy IoT application, it is crucial to verify that a device is not compromised by malicious software, and also assert correct execution of the program. In the literature, solutions based on remote attestation, anomaly detection, control-flow and data-flow integrity have been proposed to detect software attacks. However, these solutions have limited applicability in terms of target deployments and attack detection, which we inspect thoroughly.

In this dissertation, we propose three solutions to detect software attacks on embedded IoT devices. In particular, we first propose SWARNA, which uses remote attestation to verify a large network of embedded devices and ensure that the application software on the device is not tampered. Verifying the integrity of a software preserves the static properties of a device. To secure the devices from various software attacks, it is imperative to also ensure that the runtime execution of a program is as expected. Therefore, we focus extensively on detecting memory corruption attacks that may occur during the program execution. Furthermore, we propose, SPADE and OPADE, secure program anomaly detection that runs on embedded IoT devices and use deep learning, and machine learning algorithms respectively to detect various runtime software attacks. We evaluate and analyse all the proposed solutions on real embedded hardware and IoT testbeds. We also perform a thorough security analysis to show how the proposed solutions can detect various software attacks.

ZUSAMMENFASSUNG

Internet der Dinge (IoT) Anwendungen werden zunehmend in immer mehr Bereichen eingeführt, wie intelligenten Gebäuden, Kraftfahrzeugen, intelligenten Fabriken als Beispiele neben vielen Anderen. In diesen Anwendungen werden eingebettete Geräte häufig als Sensoren, Aktoren oder Edge Nodes eingesetzt. Diese eingebetteten Geräte arbeiten eigenständig an einer Aufgabe oder interagieren miteinander, um gemeinsam bestimmte Aufgaben zu erfüllen. Die zunehmende Vernetzung von *Dingen* mit dem Internet hat eingebettete Geräte zu einem attraktiven Ziel für verschiedene Cyberangriffe gemacht, bei denen ein Angreifer Zugriff auf entfernte Geräte erhält und diese für bösartige Aktivitäten kontrolliert. Diese IoT-Geräte erleichtern es einem Angreifer, die Plattform des Geschädigten zu kompromittieren, ohne dass ein physischer Hardware-Zugang erforderlich ist.

Um solche Softwareangriffe zu erkennen und sicherzustellen, dass die IoT-Anwendungen wirklich vertrauenswürdig und zuverlässig sind, muss unbedingt überprüft werden, dass die Geräte nicht durch eine bösartige Software kompromittiert wurden und die korrekte Ausführung des Programms sichergestellt ist. In der Literatur wurden Lösungen vorgeschlagen, die auf Remote-Attestierung, Anomalieerkennung, Kontroll- und Datenflussintegrität basieren, um Softwareangriffe zu erkennen. Diese Lösungen sind jedoch nur begrenzt anwendbar, was den Einsatz und die Erkennung von Angriffen betrifft, wie wir gründlich untersuchen.

In dieser Dissertation schlagen wir drei Lösungen zur Erkennung von Software-Angriffen auf eingebettete IoT-Geräte vor. Zunächst stellen wir SWARNA vor, das ein großes Netzwerk von eingebetteten Geräten mithilfe von Remote-Attestierung überprüft und sicherstellt, dass die Anwendungssoftware auf dem Gerät nicht manipuliert wurde. Dabei werden die statischen Eigenschaften des Geräts bewahrt . Um die Geräte vor verschiedenen Softwareangriffen zu schützen, muss unbedingt auch sichergestellt werden, dass die Ausführung eines Programms zur Laufzeit wie erwartet erfolgt. Wir konzentrieren uns daher weitgehend auf die Erkennung von Angriffen auf den Speicher, die während der Programmaus- führung auftreten können. Wir präsentieren SPADE und OPADE, welche eine sichere Erkennung von Programmanomalien ermöglichen, die auf eingebetteten IoT-Geräten laufen und Deep Learning bzw. maschinelle Lernalgorithmen verwenden, um verschiedene Softwareangriffe während der Laufzeit zu erkennen. Wir bewerten und analysieren alle

vorgeschlagenen Lösungen auf normaler eingebetteter Hardware sowie IoT-Testumgebungen. Wir führen auch eine gründliche Sicherheitsanalyse durch, um zu zeigen, wie die vorgeschlagenen Lösungen verschiedene Softwareangriffe erkennen können.

ACKNOWLEDGEMENTS

Foremost, I would like to thank my supervisor Prof. Patrick Eugster for his continued support, feedback and guidance. His dedication and passion for research has highly influenced me during my PhD research.

I wish to express my sincere gratitude to Prof. Max Mühlhäuser for his extended support during my PhD and also his willingness to act as a reviewer. I take this opportunity to also acknowledge the support from the administration team of Distributed Systems Programming Lab and Telecooperation Lab of TU Darmstadt.

I would like to thank my colleagues Marcel Blöcher, Malte Viering, and Patrick Jahnke for their continued support, great conversations and discussions. I would also like to thank Akash Agarwal, Savvas Savvides, Fiona Murphy, Harsha Nandeeshappa, Tzu-Chun Chen (Gina), and Prof. Silvia Santini for their collaborative effort, feedback, and discussions. I would like to express my sincere gratitude towards Prof. Kristian Kersting, Prof. Carsten Binnig, and Prof. Zsolt István to act as committee members.

I thank my parents for their continuous love and support. And last but not least, I am honoured and grateful to my husband Lohith Yagatera for being a huge pillar of support.

CONTENTS

1 Introduction 1
1.1 Contributions . . . 3
1.2 Dissertation Statement . . . 5
1.3 Roadmap . . . 5
2 Background 8
2.1 Attacks on Embedded IoT Devices . . . 8
2.2 Software Attacks . . . 9
2.2.1 Software Tampering Attacks . . . 9
2.2.2 Memory Corruption Attacks . . . 10
2.3 Defense: Remote Attestation . . . 13
3 Literature Review 16
3.1 Detecting Software Tampering Attacks using Remote Attestation 16
3.2 Memory Corruption Attacks Defenses . . . 18
I Software Tampering Attacks
4 SWARNA: Software-based Remote Network Attestation 23
4.1 Introduction . . . 24
4.1.1 IoT Malware . . . 24
4.1.2 Remote Attestation . . . 25
4.1.3 Back to the Future . . . 25
4.1.4 Challenges . . . 26
4.1.5 Contributions and Roadmap . . . 26
4.2 Problem Definition . . . 27
4.2.1 System Model . . . 27
4.2.2 Assumptions . . . 27
4.2.3 Threat Model . . . 28
4.2.4 Problem Statement . . . 28
4.3 Overview . . . 29
4.3.1 Monitoring and Control . . . 29
4.3.2 Attestation Protocols . . . 30
4.4 Deterministic Communication Paths . . . 30
4.4.1 Network Reliability and False Positives . . . 31

4.4.2 TSCH Link-Layer Protocol 31
4.4.3 Generating TSCH Schedules for Attestation 32
4.5 SWARNA Protocols 33
4.5.1 SWARNA-individual 33
4.5.2 SWARNA-aggregate 35
4.5.3 Theoretical Analysis 41
4.6 Implementation 44
4.7 Evaluation 44
4.7.1 Analytical Evaluation vs Simulation Results 45
4.7.2 Empirical Evaluation 45
4.7.3 Use Case: Data Collection 49
4.8 Security Analysis 51
4.8.1 Passive Malware 51
4.8.2 Active Malware 51
4.8.3 DDoS Attacks 53
4.8.4 Modifying Dedicated Slots 53
4.8.5 Non-Persistent and Persistent Malware 53
4.9 Conclusions 53

II Memory Corruption Attacks

5 SPADE: Secure Program Anomaly Detection for Embedded IoT Devices 57
5.1 Introduction 58
5.2 Problem Statement 61
5.2.1 Threat Model 61
5.2.2 Security Goals 63
5.2.3 Trusted Hardware (and) Challenges 64
5.3 Overview and System Architecture 65
5.3.1 Precise Calling Context 65
5.3.2 Main Components 66
5.3.3 Checkpoints 67
5.3.4 Buffer Reader 68
5.3.5 Anomaly Detection 68
5.4 SPADE 69
5.4.1 Program Behavior Modeling using Precise Call Sites . . . 70
5.4.2 Trusted Execution Environment for Anomaly Detection . 72
5.4.3 Tracing 73

5.5 Implementation 74
5.5.1 Traces for Training 75
5.5.2 Neural Network Model 75
5.6 Evaluation 76
5.6.1 Experimental Setup 76
5.6.2 RQ1: Real-world Attack Detection 78
5.6.3 RQ2: Overhead and Trade-offs 81
5.6.4 GRU vs LSTM 83
5.7 Security Analysis 85
5.7.1 Control-oriented Attacks 85
5.7.2 Stealthy Attacks 85
5.7.3 Modifying Checkpoints 86
5.8 Discussion 87
5.9 Conclusion 87
6 OPADE: Online Program Anomaly Detection on Embedded IoT Devices 90
6.1 Introduction 90
6.2 Problem Statement 92
6.2.1 Behavioural Control Anomalies 93
6.2.2 Threat Model 94
6.2.3 Example: A Vulnerable Insulin Pump 95
6.3 OPADE 96
6.3.1 Overview 96
6.3.2 Features for Anomaly Detection 98
6.3.3 Tracing 100
6.3.4 Online Anomaly Detection using HTMs 101
6.4 Implementation 104
6.4.1 Source Instrumentation 104
6.4.2 Hierarchical Temporal Memory Algorithm 105
6.5 Evaluation 105
6.5.1 Experimental Setup 105
6.5.2 RQ1: Systematic Evaluation Program Behaviour Anomalies 107
6.5.3 RQ2: Real-world Attack Detection 108
6.5.4 RQ3: Overhead 111
6.6 Conclusion 113
7 Conclusion 115

7.1 Dissertation Summary 115
7.2 Future Research Directions 116
Bibliography 117
Glossaries 133
List of Figures 134
List of Tables 139

III Appendix

A Curriculum Vitae 144
B Erklärung laut Promotionsordnung 145

1 INTRODUCTION

CHAPTER CONTENTS

1.1 Contributions 3
1.2 Dissertation Statement 5
1.3 Roadmap 5

A rapid deployment of IoT applications has led to increase in Internet connected devices. Embedded devices are widely used in IoT applications such as home automation, automotive, healthcare, as well as in critical infrastructures namely electric power grid, chemical facilities, natural gas and oil distribution. In these applications, embedded devices typically monitor their physical environment to read data from various sensors, process data and make control decisions to act on physical environment. In general, due to the fact that devices are connected to each other and to the Internet has made embedded devices an attractive target for various cyber attacks, where an attacker gains access and control devices for malicious activities. Internet connected devices facilitate an attacker to compromise the victim's platform without requiring any physical hardware access.

Software attacks. Due to the additional cost, energy (battery usage), and limited resources on embedded devices, security is an afterthought, and not incorporated by design. The consequence is an ongoing stream of exploits. An attack on a water treatment plant in 2021 manipulated the quantity of chemical mix [53], and a similar attack was also seen in 2016 [75]. In late 2019, a vulnerability in a communication protocol was exploited to gain unauthorized access to a hotel's in-room robot assistant that could be used to spy on guests. [58]. In 2017, a protocol vulnerability on Philips hue smart lamps was exploited to control the smart street lights of an entire city [94]. In another example, two researchers remotely updated the firmware of a car to gain control of its steering wheel, breaks, and engine [83]. Those are only a few examples of evident attacks that are known to the public. The above attacks either tamper the application binary or corrupt the runtime memory to gain control of the device.

Defense approaches. Remote attestation, control flow integrity (CFI), and program anomaly detection are techniques that have been widely advocated to detect various software attacks. Remote attestation is a prominent approach to remotely verify the integrity of the software running on embedded device [44, 98, 23] or the runtime control flow of the program [3, 104]. In remote attestation, an external trusted entity verifies the IoT devices, and is reactive in nature. CFI is a prevention mechanism that monitors the control flow of a program on a device at execution, ensuring that it follows pre-determined legitimate paths. CFI and remote attestation, however, cannot capture the non-deterministic runtime characteristics of a program (e.g., execution of if-else statements at runtime). Program anomaly detection techniques, on the other hand, use various features to capture the behavior of a program [116, 117]. Even a subtle deviation in the runtime behavior compared to the normal execution is flagged as anomaly.

Challenges. In order to ensure the IoT applications are truly trustworthy and reliable it is imperative to (i) assure that the devices are not compromised by malicious software, (ii) assert correct execution of the embedded control program, and (iii) finally ensure that the proposed attack detection mechanism is itself secured. A wide range of solutions have been proposed to detect software attacks on embedded IoT devices.

Due to the limited resources on embedded devices, the goal of ensuring device security is most often offloaded to a remote entity with unlimited resources (c.f. remote attestation). Remote attestation solutions have been applied for both verifying device software and also to verify correct execution of the program. However, the solutions most often rely on special hardware capabilities [44, 20, 104], or apply to single device setting [98, 4]. With increase in IoT applications where devices are distributed and operate collectively (e.g., smart cities, automated factories), there is a need for *swarm* attestation. Current swarm attestation techniques rely on special hardware capabilities to generate attestation results which makes them inapplicable to both legacy and lower-end constrained devices (class 0 or class 1 devices [19]).

However, implementing a secure software-based swarm attestation is challenging for several reasons. Software-based attestation relies on strict timeout values to generate attestation results, and the non-deterministic behavior of multi-hop wireless networks causes unpredictable communication latency making it difficult to derive the timeout value. Also, malicious relay nodes could tamper with responses of their descendants making it hard to pinpoint benign and malicious nodes in the network.

IoT devices that are slightly less resource constrained, implement solutions that can execute on IoT devices with limited or no dependency on external entities, which make them proactive in detecting attacks at runtime. Anomaly detection is widely deployed on embedded devices which use various features like process information [21], performance counters [2], and Linux system call usage [29, 119] for detecting runtime anomalies. Most of the proposed solutions [2, 119] can detect only changes in high-level execution contexts and is not sensitive to small local variations between two system calls (c.f. control-oriented attacks). Inversely, plain sequence-based approaches are sensitive to only local variations or is order-insensitive [29, 101]. Sequence-based solutions cannot detect attacks that takes illegal-yet-valid control path. Identifying the correct features to trace and designing an anomaly detection scheme to detect both local variations and changes in high-level execution contexts is challenging. It should also be noted that there are very few anomaly detection techniques for embedded devices and a proposed solution has to be efficient in terms of memory requirement and runtime latency.

1.1 CONTRIBUTIONS

In this dissertation, we address the aforementioned challenges and present three solutions to secure embedded IoT devices. The first solution is aimed at verifying the IoT devices to detect any attacks that may have tampered the application software. The other two solutions focus on detecting memory corruption attacks that may occur during the execution of a program. More specifically, this dissertation has the following contributions:

- Classification of attacks: Over the past few decades in the literature, we can find various attacks that have been demonstrated on IoT devices. In this dissertation, we define and classify the attacks into various categories that helps us in designing a defense solution focused on an attack class.
- Software-based remote network attestation: To cater to the increasing in scale and number of IoT deployments, we present SoftWAre-based Remote Network Attestation (**SWARNA**), a system to attest swarm of remote IoT devices. SWARNA builds a deterministic communication path enforcing time bounds across a wireless multi-hop IoT network for attestation purposes. Also, SWARNA implements two novel remote swarm attestation protocols: (i) SWARNA-ind to verify the devices individually, and (ii) SWARNA-agg that verifies a network of devices by aggregating the attestation results at intermediate nodes. We implement SWARNA and evaluate attestation times and communication overhead both via IoT testbed and simulation. We also

demonstrate applicability of SWARNA with an IoT periodic data collection application.

- Secure program anomaly detection: In order to detect memory corruption attacks at program runtime, we propose a secure program anomaly detection for embedded IoT devices (**SPADE**). SPADE runs on embedded devices and the anomaly detection is triggered efficiently to detect attacks instantaneously. For anomaly detection, SPADE captures the behavior of a program using function calls with precise caller sites and implements a gated recurrent unit (GRU)-based anomaly detection scheme to detect attacks that modify the execution flow of a program. The function call traces during program execution are extracted from embedded trace macrocell (ETM) [78], an on-board debugging component present on an ARM Cortex processor. Such hardware-based tracing incurs only minimal overhead. We also consider software-based tracing by instrumenting the source code, which incurs higher overhead but can extract many more useful features and does not add hardware constraints. SPADE leverages trusted hardware that provides an isolated execution environment, to defend the proposed anomaly detector against mimicry attacks. SPADE thus combines trusted hardware and software solutions to create secure and trustworthy IoT systems. We evaluate the anomaly detection accuracy through real-world applications, and its static and runtime overhead.
- Online program anomaly detection: We first introduce behavioural control anomaly (BCA), an anomaly seen during program execution that affects the behaviour of the program by modifying one or several control aspects of the program. We further classify the anomalies into: control flow anomaly (CFA), control branch anomaly (CBA), and control intensity anomaly (CIA). We design an online program anomaly detection for embedded IoT devices (**OPADE**). OPADE captures sequence of function calls with its precise calling location, number of times a function call was invoked, and a loop execution cycle count using hardware counters. OPADE implements an hierarchical temporal memory (HTM) based anomaly detection technique that supports sequence and continual learning, and does not require a separate data collection and training like other neural network algorithms. Similar to SPADE, OPADE runs inside trusted execution environment. We evaluate OPADE for its anomaly detection accuracy in detecting the BCAs and also its overhead.

Declaration of Originality. All ideas, models, algorithms, and implementation details described are the results of my work under the supervision of Prof. Patrick Eugster. All systems presented in this dissertation – SWARNA,

SPADE, OPADE– are built from scratch. I was fortunate to collaborate with colleagues and students, who have at times assisted me in extending and evaluating specific prototype components. Furthermore, the co-authors of the publications have contributed in designing the algorithms, models of the implemented systems, and discussions while working on the publications.

SWARNA was published together with Prof. Patrick Eugster and Prof. Silvia Santini who was involved in discussions throughout the work. Akash Agarwal was involved in the discussions along with Prof. Patrick Eugster in designing SPADE, in particular the threshold based anomaly detection technique. The HTM algorithm in OPADE was implemented by Fiona Murphy during her internship supervised by me and Prof. Patrick Eugster.

1.2 DISSERTATION STATEMENT

In this dissertation, we introduce methodologies to build trustworthy and reliable IoT applications by securing embedded IoT devices. We propose a system to collectively verify the integrity of the software running on remote IoT devices and further propose two solutions, which are secure by design, to detect memory corruption attacks that occur during the runtime execution of a program, and detects broader class of attacks on embedded device at runtime compared to state-of-the-art detection techniques.

1.3 ROADMAP

The dissertation is structured as follows: Chapter 2 presents the background information on various attacks on embedded IoT devices and their classification required to understand this dissertation. The background also provides working of remote attestation protocol which is used to verify integrity of devices. Chapter 3 provides the literature review of works closely related to this dissertation.

We then present the dissertation in two parts. Part I focuses on detecting attacks that tamper the application software. Chapter 4 presents a software-based remote network attestation technique that verifies the integrity of the software running on the device.

Part II of the dissertation focuses on detecting memory corruption attacks that may occur during the execution of a program. Chapter 5 presents a GRU-based anomaly detection technique to detect attacks that modify the execution flow of a program. Chapter 6 presents an HTM based online anomaly detec-

tion technique that detects various behavioural control anomalies. Chapter 7 concludes this dissertation and presents future research directions.

2 BACKGROUND

CHAPTER CONTENTS

2.1 Attacks on Embedded IoT Devices 8
2.2 Software Attacks 9
2.2.1 Software Tampering Attacks 9
2.2.2 Memory Corruption Attacks 10
2.3 Defense: Remote Attestation 13

In this chapter, we provide basic information on attacks and essential defense mechanisms vital to understanding this dissertation. First, we broadly classify attacks targeted at embedded IoT devices in Section 2.1. We then describe the software attacks in detail in Section 2.2, as the focus of this dissertation is detecting software attacks. We broadly classify the software attacks into software tampering and memory corruption attacks, which are described in Section 2.2.1 and Section 2.2.2. We also review the most prominent memory corruption attacks: control-oriented and data-oriented attacks. Finally, in Section 2.3, we explain the working of remote attestation, which is used for detecting software attacks.

2.1 ATTACKS ON EMBEDDED IOT DEVICES

An IoT system comprises several smart devices that exchange data through the Internet. In this section, we classify and describe attacks compromising the security of embedded IoT devices.

We broadly classify the attacks on embedded IoT devices into two categories: software attacks and hardware attacks [71]. Figure 1 shows the classification of attacks on embedded IoT. Based on the literature, we classify attacks that compromise only the software of a device, that is, the code and the data residing on the device, as **software attacks**. With software attacks, an attacker can exploit vulnerabilities in the code to take the target device under control which may lead to catastrophic consequences. An attacker can execute a software attack remotely, making it an easy and widespread choice of attack.

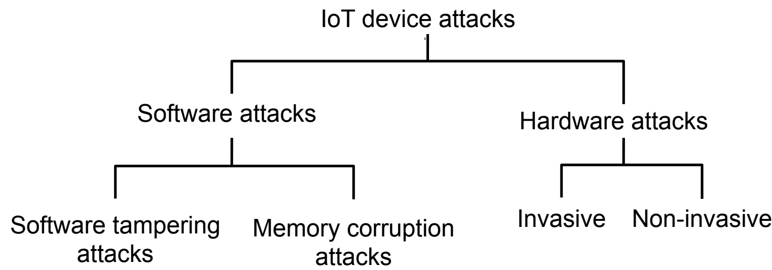


Figure 1: IoT attacks classification

The attacks that capture a device and physically tamper it are classified as **hardware attacks** [61, 71]. Hardware attacks that directly access the internal components of a device using expensive and sophisticated equipment are termed as *invasive attacks*. Hardware attacks that use low-cost electrical engineering tools to extract the cryptographic material stored on the device are termed as *non-invasive attacks*. The famous side-channel attack is an example of a non-invasive attack, which uses electromagnetic radiation, power, or time to extract sensitive data during normal device operations.

In this dissertation, we focus on software attacks and propose several solutions to detect various software attacks. Hence in the following sections, we provide an in-depth description of software attacks.

2.2 SOFTWARE ATTACKS

An attack is defined as a software attack when an adversary modifies the code or data on a device. We further categorize software attacks into software tampering attacks and memory corruption attacks (ref. Figure 1).

2.2.1 *Software Tampering Attacks*

The adversary here is capable of modifying the application binary before or after deployment by exploiting vulnerabilities in the source code or the deployment setting. Along with the application software, the adversary can also modify any configuration settings on the device. However, the application binary or the device configuration typically resides in a device's non-volatile memory (survives device restarts) and does not change during program execution. Hence, we define attacks that only tamper the static properties of a device as software tampering attacks.

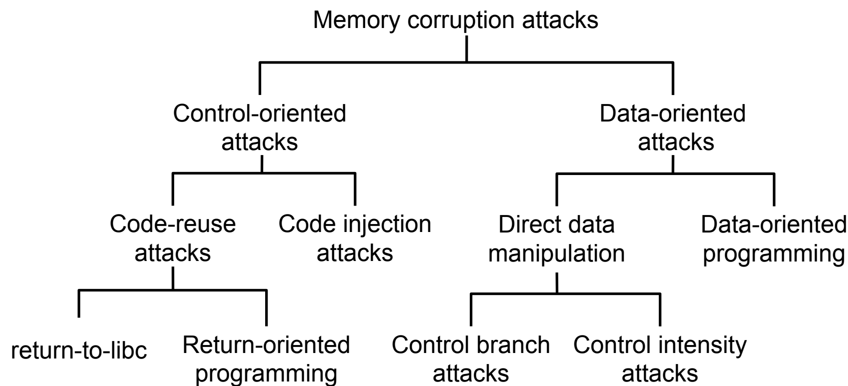


Figure 2: Memory corruption attacks classification

2.2.2 *Memory Corruption Attacks*

During program execution, an application generates various runtime information that is stored in volatile memory (resets on device restarts). Typically in embedded devices, the runtime information is stored on a stack or heap, which resides on the volatile RAM. An attacker can exploit the vulnerabilities in the application software, like buffer overflow, to modify the runtime data of the application. Therefore, we classify attacks that modify the contents of runtime memory as memory corruption attacks. As the memory corruption attacks take place during the execution of an application program, in the literature, memory corruption attacks are also referred to as runtime attacks [28]. The classification of memory corruption attacks is shown in Figure 2.

Based on the data that is modified in the memory, memory corruption attacks can be further classified into control-oriented attacks and data-oriented attacks. Figure 3 shows how the control and data flow look during various memory corruption attacks. We further explain the memory corruption attacks in detail.

2.2.2.1 *Control-oriented attacks*

In control-oriented attacks, the adversary modifies the control information (e.g., return address, code pointer) of a program that resides in memory during program execution. The aim of an adversary is to divert the intended execution flow of a program. By subverting the control flow, the adversary can trigger one or several malicious actions, such as accessing sensitive data, illegally executing a program, or injecting malware.

Control-oriented attacks [34] can be broadly classified into (1) code injection and (2) code-reuse attacks. In code injection, an adversary first exploits a

vulnerability to inject malicious executable code into the application's address space. As a next step, the adversary uses the same or finds another exploit to redirect the control flow of a program to execute the injected malicious code. The control flow of a program during the attack is shown in attack 1 in Figure 3. The figure shows that the malicious code is placed in the runtime stack, and the current return address of a program is altered to point to the malicious code location, which is further executed. Executing a code injection attack on Harvard architecture, where the code and data memories are separated, is not straightforward and requires expert skills [48]. The Harvard architecture is the typical CPU design in embedded systems.

In contrast to code injection attacks, code-reuse attacks do not require injecting malicious code into the application's address space. Instead, in code-reuse attacks, an adversary redirects the control flow of a program to execute an already existing code block of the program. For example, as shown in Figure 3, in attack 2, the return address is modified to redirect the control flow of a program to a current location in the program memory. If the attacker can execute a function that performs critical tasks, then the effect of such an attack is disastrous.

In order to redirect the control flow of a program, the adversary tampers the code pointer. The most common way to hijack the control flow is to change the return address of a function stored on the program stack to a new address location. Function pointers can also be modified to hijack the control flow of the program. The target address can point to a critical function that an adversary can obtain by reverse engineering the program binary using a debugger.

2.2.2.2 *Data-oriented attacks*

Data-oriented attacks modify the benign behavior of a program by altering the non-control data (data variables that do not contain any address information). The adversary modifies the internal data variables or data pointers during the execution of a program without violating the control flow integrity.

In the literature, several types of attacks corrupt the non-control data to alter the program's behavior. Data-oriented attacks [28] can be categorized into (1) direct data manipulation and (2) data-oriented programming. In direct data manipulation, an adversary directly changes the non-control data variable to achieve the required malicious goal. Here the attacker should know the precise address of the target variable. Binary analysis or deriving the randomized address stored in the memory can be used to obtain the target address of a variable. The attacker then exploits a program vulnerability to overwrite a data variable's memory location. In Figure 3, a direct data manipulation attack

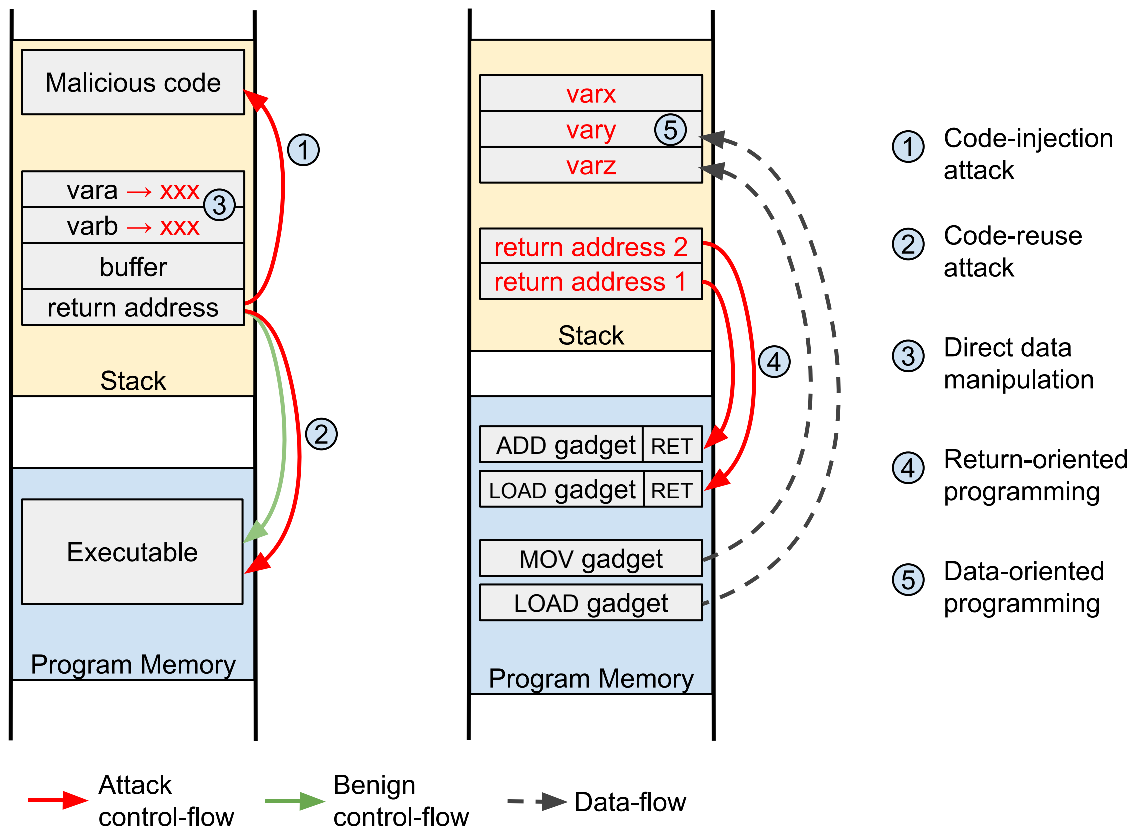


Figure 3: Memory corruption attacks description. The figure shows how the control-flow and data-flow is altered during various memory corruption attacks.

shows that the non-control data variables stored on the stack are altered to a malicious value.

Based on the impact of an attack, we classify direct data manipulation into control branch and control intensity attacks (cf. Figure 2). When an adversary modifies critical decision-making variables at runtime to illegally execute a control branch, we term such an attack a control branch attack. Control branch attacks alter the benign control flow path of a program without violating the control flow integrity of a program, i.e., the attack does not introduce any illegal control flow. If the adversary corrupts a data variable to alter the amount of control operations is called a control intensity attack. An example of such an attack is modifying the number of loop iterations to dispense large amounts of chemicals illegally.

Data-oriented programming attack is a systematic technique of generating expressive non-control data exploits. The attack involves finding gadgets and chaining them in arbitrary sequences using a gadget dispatcher. The dispatcher is usually a loop [59] because an attacker can execute a sequence of instructions several times to achieve the desired effect within a loop.

2.3 DEFENSE: REMOTE ATTESTATION

Remote attestation [98, 3] is a security mechanism used by a trusted entity to verify the integrity of a remote untrusted device. Due to the limited resources on embedded devices, using an external trusted entity in the IoT architecture is typical. Remote attestation uses a challenge-response protocol to validate the internal state of a remote device. In remote attestation, a trusted entity called the *verifier* issues a challenge to an untrusted *prover*. The prover generates a response to the challenge and sends it to the verifier to prove its legitimate state. It is generally assumed that the verifier is aware of all the possible correct states of the prover. The challenge is generated such that a compromised prover cannot produce a valid response.

The Figure 4 summarizes the working of remote attestation. First, the verifier creates a unique challenge and sends an attestation request to a remote prover. Next, the prover executes an attestation routine to calculate and send an attestation response back to the verifier. Finally, the verifier compares the state of the prover received in the attestation response with the expected state of the device. If the results match, the verifier declares the remote device to be trusted; otherwise, the remote device remains untrusted.

Remote attestation is broadly classified into three categories:

- Hardware-based attestation: Hardware-based attestation techniques rely on tamper-resistant components like trusted platform module (TPM) or

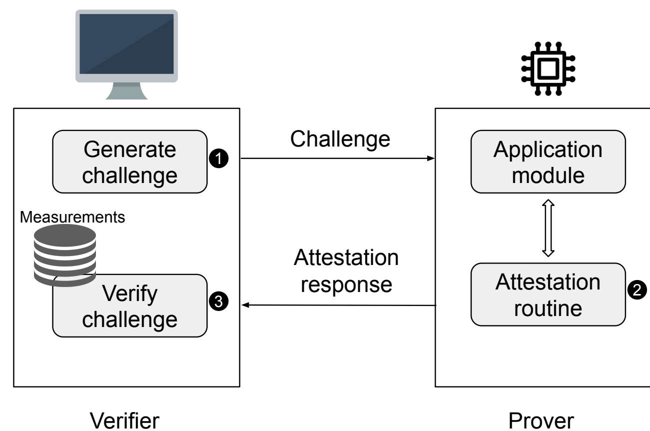


Figure 4: Overview of Remote Attestation

physical unclonable functions (PUF). The attestation response is calculated using tamper-proof hardware; therefore, the verifier trusts the attestation response.

- Software-based attestation: Software-based attestation does not require any trusted hardware; instead, it relies on strict time restrictions. When the verifier sends an attestation request, it expects the prover to generate a response within the given time bounds. If the prover cannot generate a valid response within the given time, then the prover is assumed to be compromised. Software-based attestation depends on the fact that, given sufficient time, an adversary can move the malware present on the device and still generate a valid response. However, with an added restriction of time guarantees, the verifier can successfully verify the integrity of a prover.
- Hybrid attestation: Attestation techniques that do not require tamper-resistant hardware but still depend on specific hardware for generating attestation results is categorized as hybrid attestation. In the literature, micro-controllers are modified to restrict access to certain memory regions for attestation, or specific types of ROMs are used.

Remote attestation is used to attest a single prover or a collection of devices, which is called swarm attestation [23]. In the literature, we find remote attestation has been used to verify the software integrity of devices, and also runtime execution of the application program [3, 104]. We review various remote attestation techniques in the literature in Section 3.1.

3 LITERATURE REVIEW

CHAPTER CONTENTS

3.1 Detecting Software Tampering Attacks using Remote Attestation 16
3.2 Memory Corruption Attacks Defenses 18

In this chapter, we review the literature works related to this dissertation. We first present the related works that verify the static properties of an embedded device through remote attestation in Section 3.1. In Section 3.2, we review the defense mechanisms proposed to detect various memory corruption attacks.

3.1 DETECTING SOFTWARE TAMPERING ATTACKS USING REMOTE ATTESTATION

Remote attestation is popularly used to detect attacks that alter a device's static properties, such as the program binary and software configuration. Remote attestation implements a challenge-response protocol that enables a trusted *verifier* to verify the integrity of the software running on a remote embedded device.

We classify the remote attestation works into two dimensions - (1) attestation approach and (2) attestation scope.

Approach - Software vs Hardware. Early remote attestation approaches were mostly software-based with the focus being to implement an optimized and secure routine to calculate a memory checksum (e.g., [98, 4]). That is, on attestation request, a device computes the checksum of attestable memory, and the verifier accepts a device's response only if it is received within a predefined time frame. One of the reasons software-based remote attestation has been criticized for its difficulty in implementing a simple, time-optimal checksum function. Castelluccia et al. [25] perform several attacks on SWATT [98] to show the difficulty of implementing a time-optimal checksum routine for software attestation. After refutations the authors state [49] that software attestation can be secure if the verifier communicates directly with the device over a single hop and data memory is included in the attestation measurement. Later, Armknecht

et al. [12] present a formal generic security framework to implement such time-optimal attestation routines. However, the checksum function or the attestation protocol may have to be slightly modified based on device architecture. Li et al. [76] propose pre-filling data memory with pseudo-random values known to the verifier before attestation to detect malware in unused memory. SμV [9] presents a software-based security architecture that provides memory isolation using selective software virtualisation and machine-level code verification.

Based on limitations of software-based remote attestation, trusted hardware like TPMs are exploited to perform hardware-based remote attestation [105]. Due to the relatively high overhead and complexity involved in such hardware-based solutions, several hybrid secure architectures for IoT have been proposed. SMART [44] stores the attestation routine and secret keys in ROM and modifies the micro-controller to restrict access to the code in ROM. A hybrid remote attestation with formally verified security guarantees is proposed in VRASED [87]. TyTAN [20] and Sancus [85] are other secure hardware architectures that provide remote attestation features and can run several separate programs in isolation. HAtt [7] leverages physical unclonable functions to attest the memory of a remote IoT device iteratively in blocks instead of the entire memory at once, as done in existing software and hybrid attestation techniques.

Scope - Nodes vs Swarms. There has been an increase in IoT malware used as bots for large-scale distributed denial-of-service (DDoS) attacks [72]. Since in IoT applications devices are interconnected to form a network, several researchers have thus considered remote attestation for entire swarms. Building on SMART, LISA [23] stores the attestation routine, a pre-computed checksum, and secret keys in secure memory. On attestation request, a memory checksum is computed upon and compared against pre-computed values. Identifiers of benign nodes are combined and sent in a response packet whose size increases with network size. In contrast, SWARNA aggregates responses by $\oplus$-ing the received checksums, yielding constant packet sizes. SANA [8] introduces a novel optimistic aggregate signature (OAS) which uses multi-signature and aggregate signature schemes. With OAS, a correct node signs a default message sent by verifier and a malicious node signs the hash of its program memory. SANA was implemented on secure architectures, SMART and TyTAN. While LISA, SANA, and SWARNA use aggregation as a technique to minimize communication overhead, they differ greatly in what is aggregated. US-AID [60] is applicable to dynamic topologies. Nodes periodically attest their neighbors, to identify software attacks and use periodic heartbeat messages to identify their physical presence. By combining continuous attestations and heartbeat messages, US-AID identifies both software and physical attacks. ESDRA [74]

takes a distributed approach where neighboring nodes directly attest provers and hence an IoT swarm.

As stated, previous works on software-based remote attestation require a *strict timing guarantee* for successful operation, making software-based attestation challenging for multi-hop networks. Seshadri et al. [97] propose an expanding ring method for multi-hop networks where the sink attests one-hop nodes and these nodes attest their neighbors. The focus is to implement optimal checksum calculation routine and not efficient swarm attestation protocol.

3.2 MEMORY CORRUPTION ATTACKS DEFENSES

In this section, we present the related works in identifying various memory corruption attacks, specifically control-oriented and data-oriented attacks targeted at an embedded device.

Program anomaly detection. Using machine learning algorithms to capture program runtime information has proved to yield effective behavioral models to detect deviations (program anomalies) during runtime. Such a technique is orthogonal to integrity techniques – that solely focus on static analysis – in that it is designed to offer quantitative behavior such as distinguishing path frequencies that might be indicative of run-time program misuses.

Forrest et al. first proposed program anomaly detection using system call traces [47]. Initial approaches used hidden markov models (HMMs) [114] and finite state automata (FSA) [95] to analyze information flows quantitatively using frequencies of system calls. STILO [117] and CMarkov [116] introduced static program analysis to initialize HMMs and use 1-level calling context respectively. These approaches primarily focus on the identification of local anomalies by inspecting short call segments. They are therefore suitable for only control-oriented attacks that exhibit deviations in short traces. Also, HMMs tend to converge to local optima, hampering prediction performance. The use of HMMs for predicting the next executed call and characterize normal executions has limitations. Firstly, it can captures only limited call history to model the next call. Therefore, it loses valuable execution information for informed predictions. Secondly, HMMs tend to converge to local optima, hampering prediction performance.

Long-span anomaly detection (LAD) [101] captures long-term behavior of executions using co-occurrence and frequency analysis. But it is order-insensitive and fails to jointly learn the semantics of individual function calls and their interactions appearing in the trace sequences. Therefore, it cannot detect aberrant path attacks where the attack trace resembles a normal trace with just a

few missing sequences (cf. Table 3). In contrast, our proposed scheme leverages machine learning models to analyze long traces. We show empirically that even moderate-length traces can effectively detect control-oriented and aberrant path attacks (data-oriented attacks).

Anomaly detection on embedded devices. Several features of embedded devices have been explored in identifying various types of anomalies on lightweight devices. HADES-IoT [21] prevents executing new processes that do not belong to a known whitelist of benign processes. However, this technique does not defend against control-oriented attacks. Abbas et al. [2] use hardware performance counters to train benign and anomalous applications using a support vector machine. However, the hardware counters used are not available on several lightweight embedded devices like Cortex-M. Yoon et al. [119] cluster system calls to identify patterns in system call frequency distributions using k-means clustering. Monitoring frequencies of system calls can detect only changes in high-level execution contexts and is not sensitive to small local variations between two system calls (as in control-oriented attacks). Inversely, plain sequence-based approaches are sensitive to only local variations and cannot detect aberrant path attacks. A fine-grained context-sensitive sequence-based approach can detect a large set of attacks including control-oriented and aberrant path attacks which we demonstrated in this paper. Orpheus [29] builds an event-aware finite state automaton (eFSA) model to detect runtime data-oriented attacks that change Linux system call usage and cannot detect attacks on bare-metal devices.

Alternate defense approaches. In this paper, we propose an on-device attack detection solution for resource-constrained IoT devices using a program anomaly detection algorithm. Most popular alternative techniques to program anomaly detection to secure devices against control-oriented attacks include remote attestation [3] and CFI [1]. In remote attestation, a trusted external entity verifies a remote IoT device for its software integrity. C-FLAT [2] remotely verifies the control flow using remote attestation technique. Along with control flow attestation, OAT [104] verifies the integrity of critical data. Recent works on remote attestation use secure hardware and swarm attestation for IoT networks [24]. Remote attestation requires an external trusted verifier to periodically verify IoT devices, and is reactive in nature. Unlike remote attestation and anomaly detection, CFI is a prevention mechanism, CFI works on the device, monitoring the control flow of a program on a device at execution, ensuring that it follows pre-determined legitimate paths. CFI is widely studied for general-purpose systems and is shown to incur very high processing overhead as the program regularly checks every control flow step [1]. CFI has

been attempted on resource-constrained devices using a pure software-based approach [32], exhibiting significant runtime overhead compared to hardware-based CFI techniques [33]. Also, CFI in general addresses only control-oriented attacks. Hasan et al. [55] use ARM TrustZone on Cortex-A to verify all actuator commands before forwarding them to the peripheral. The approach can only partially detect control-oriented attacks, and fails to detect aberrant path attacks. Data flow integrity [26] defends against specific data-oriented attacks by instrumenting all memory-accessing instructions, which however incurs a very high runtime overhead.

Secure machine learning. ARM TrustZone on Cortex-A has been used to securely execute machine learning algorithms and ensure user data privacy [16], providing the classification result to the application in the non-secure region. DeepLog [37] uses long short term memory (LSTM) to identify anomalies in system logs generated during normal execution of an underlying Linux system. In contrast to these works, the objective of proposed approaches in this thesis is to detect anomalies during the execution of a user program in the non-secure region. Several privacy-preserving neural network inference approaches have been proposed [93, 81, 51, 68]. The main objective of these works is similarly to ensure the privacy of the user's input provided to the machine learning model. Another focus is to secure the model structure and parameters from users. Inference in these works is typically performed in the cloud. In general, these works focus on preserving user data privacy, whereas we use neural networks to provide integrity of a program's control flow. Also, they focus on securing *convolutional* neural networks that cannot remember long-term sequences and hence cannot be efficiently used for anomaly detection.

Neural networks on embedded devices. Several authors implement neural network inference engines on tiny micro-controllers for specific applications other than anomaly detection [121, 100, 90]. Nyamukuru et al. [90] show that the `soft-sign` activation function is computationally more efficient compared to the `sigmoid` and `tanh` activation functions used in standard GRUs. DeepIoT [118] compresses deep neural networks by over 90%, thereby reducing execution time by over 72%. DeepIoT and the `soft-sign` activation function could be used to reduce the static and runtime overhead of the machine learning models used by the proposed techniques. DeepIoT reduces the number of hidden elements between the layers by choosing an optimal dropout probability for each layer.

Part I

SOFTWARE TAMPERING ATTACKS

4 SWARNA: SOFTWARE-BASED REMOTE NETWORK ATTESTATION

CHAPTER CONTENTS

4.1 Introduction . . . 24
4.1.1 IoT Malware . . . 24
4.1.2 Remote Attestation . . . 25
4.1.3 Back to the Future . . . 25
4.1.4 Challenges . . . 26
4.1.5 Contributions and Roadmap . . . 26
4.2 Problem Definition . . . 27
4.2.1 System Model . . . 27
4.2.2 Assumptions . . . 27
4.2.3 Threat Model . . . 28
4.2.4 Problem Statement . . . 28
4.3 Overview . . . 29
4.3.1 Monitoring and Control . . . 29
4.3.2 Attestation Protocols . . . 30
4.4 Deterministic Communication Paths . . . 30
4.4.1 Network Reliability and False Positives . . . 31
4.4.2 TSCH Link-Layer Protocol . . . 31
4.4.3 Generating TSCH Schedules for Attestation . . . 32
4.5 SWARNA Protocols . . . 33
4.5.1 SWARNA-individual . . . 33
4.5.2 SWARNA-aggregate . . . 35
4.5.3 Theoretical Analysis . . . 41
4.6 Implementation . . . 44
4.7 Evaluation . . . 44
4.7.1 Analytical Evaluation vs Simulation Results . . . 45
4.7.2 Empirical Evaluation . . . 45
4.7.3 Use Case: Data Collection . . . 49

4.8 Security Analysis 51
4.8.1 Passive Malware 51
4.8.2 Active Malware 51
4.8.3 DDoS Attacks 53
4.8.4 Modifying Dedicated Slots 53
4.8.5 Non-Persistent and Persistent Malware 53
4.9 Conclusions 53

4.1 INTRODUCTION

Embedded devices in Internet of Things (IoT) applications interact with each other to collectively perform certain tasks. Examples include *smart* cities, homes, and industries. Increase in internet-connected *things* has made them attractive targets for various malware attacks. Securing data transmission and guaranteeing the integrity of the devices are crucial for the dependability of IoT applications.

4.1.1 *IoT Malware*

Several incidents demonstrate the catastrophic effect of remote malware injection in large-scale attacks. E.g., a protocol vulnerability was exploited to update Philips hue smart lamps with malicious firmware, allowing the attacker to control the smart lights of an entire city [94]. In another example, two researchers remotely updated the firmware of a car to gain control of its steering wheel, breaks, and engine [83]. Francillon et al. describe an attack to compromise devices based on Harvard architecture [48]. Most of these malware attacks exploit software vulnerabilities to remain on the devices undetected and survive device restarts.

Due to limited resources and weak security of embedded devices, IoT systems have become a preferred target for attacks. A 2018 survey by Gartner reports that nearly 20% of organizations worldwide experienced at least one IoT-based attack in three years [65].

4.1.2 *Remote Attestation*

Remote attestation is a prominent approach to detect presence of malware on a remote embedded device by verifying the integrity of its software [105, 44, 98, 23]. In remote attestation, a trusted entity called the *verifier* issues a challenge to a remote device to prove the correct state of the device's program memory. If the response to the challenge meets the verifier's expectation, the remote device is declared trusted. There are different variants of remote attestation. *Hardware-based* remote attestation requires trusted hardware like TPMs to generate attestation results [105]. Due to the relatively high cost and complexity of hardware-based solutions, several *hybrid* architectures for IoT have been proposed that restrict access to certain memory regions by modifying the micro-controller [44, 20]. (Pure) *software-based* solutions require no additional trusted hardware of any kind [98, 4]. A device must however respond within a *strict timeout*, or attackers could compute the correct response to the challenge.

Early attestation schemes focused on single device settings, where the verifier can *directly* communicate with devices. Following the increase in IoT applications operating collectively, attention has shifted to attesting entire *networks*, a.k.a. *swarm attestation* [23]. E.g., SANA [8], LISA [23], and ESDRA [74] are hardware-based swarm attestation protocols.

4.1.3 *Back to the Future*

Miniaturization of hardware [42] and recent deployment of low-power long range communication technologies like LPWAN and LoRA [82] enable new IoT applications with tiny devices connected to the Internet (e.g., smart clothing, freight monitoring, disposable IoT). It remains a challenge to enable trusted hardware on such low(est)-end constrained IoT devices; for devices where the hardware might be available, even if individually small, the incurred costs can substantially increase for large-scale deployments.

An average of 127 IoT devices are connected to the Internet every second and the total number of devices is expected to reach 41 billion by 2027 [86]. While hardware-based remote attestation offers the highest dependability, most fielded IoT devices (still) use conventional processors without trusted hardware. For example, Philips Lighting alone installed 44 million connected light points in 2017 and 2018 which are not equipped with trusted hardware [102]. (As a comparison point, ARM launched its trusted hardware in 2003 [31].) In order to secure the billions of legacy IoT devices and next generation IoT applications, it is necessary to (re)investigate solutions for devices without trusted hardware.

4.1.4 *Challenges*

However, implementing secure software-based swarm attestation is challenging for several reasons: **(C1)** to prevent the attacker from getting sufficient time to forge an acceptable response, the attestation routine must be time-optimal; **(C2)** the non-deterministic behavior of multi-hop wireless networks causes unpredictable communication latency, which in turn makes it difficult to derive the timeout value required by software-based remote attestation; **(C3)** malicious relay nodes could tamper with responses of their descendants making it hard to pinpoint correct and malicious nodes in the network. Several authors (e.g., [98, 4, 12]) provide guidelines and solutions addressing **C1**, however challenges **C2** and **C3** have not yet been tackled in the literature.

4.1.5 *Contributions and Roadmap*

To overcome these challenges, we make several contributions. After further defining the problem addressed, we:

1. design deterministic communication paths in Section 4.4 for attestation purpose enforcing time bounds across multi-hop IoT networks (cf. **C2**).
2. introduce two novel remote swarm attestation protocols SWARNA-ind and SWARNA-agg in Section 4.5 (cf. **C3**), and analytically assess their respective total attestation times and communication overheads, exhibiting tradeoffs.
3. in Section 4.6, we implement SWARNA-agg and SWARNA-ind on Contiki OS using open standard IoT protocols allowing interoperability, easy deployment and integration of SWARNA in real-world applications.
4. empirically evaluate attestation times and communication overheads via both IoT testbed [5] and simulation in Section 4.7, and confirming our analytical assessment. For attesting a 30 nodes network on the testbed SWARNA-ind takes around 6s and SWARNA-agg takes between 1.5s to 8.2s depending on the number of malicious nodes in the network. We also show that SWARNA maintains constant payload size whereas it increases linearly with network size for SANA [8] and LISA [23].
5. demonstrate applicability of SWARNA with an IoT periodic data collection application in Section 4.7. Testbed results show that attestation has only minimal impact on the application's packet delivery ratio (0.4% drop) with upto 4% false positive rates and 0 false negatives.
6. analyze security of SWARNA in Section 4.8.

Section 4.9 draws final conclusions.

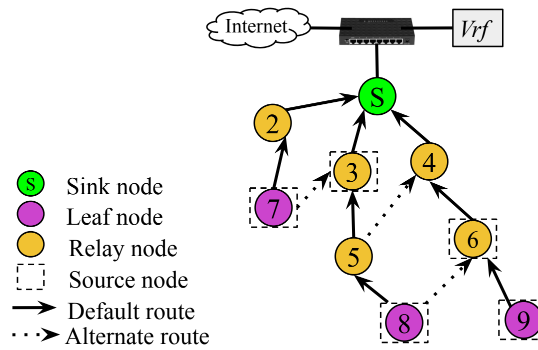


Figure 5: Example topology with *leaf nodes*, *source nodes*, *relay nodes* and a trusted *sink node*. The verifier is a trusted external entity communicating with nodes through the sink.

4.2 PROBLEM DEFINITION

This section presents the system and threat model considered, and also the assumptions and problem addressed.

4.2.1 *System Model*

We consider an IoT scenario where a network (for instance a wireless sensor network) of multi-hop nodes are connected to the Internet through a *sink node* (cf. Figure 5) with *source nodes* generating data, and *relay nodes* forwarding the data. Nodes without any child node are *leaf nodes*. The verifier is a trusted entity external to the network that sends attestation requests and verifies respective responses. The network setting described corresponds to use-cases like smart city or industrial IoT; the system model is also in line with models considered in the literature of swarm attestation. However, existing swarm attestation techniques assume trusted hardware on all devices in the network [23, 8], whose integrity has to be verified. In this chapter, we focus on proposing an efficient swarm software-based remote attestation protocol.

4.2.2 *Assumptions*

We make a few assumptions common to software-based single device attestation works (cf. Section 3.1). We assume that a public-key infrastructure suitable for embedded IoT devices (e.g., [112, 23]) is in place and nodes in the network can use the public key of the verifier to verify signed attestation requests. We assume that the verifier is aware of the architecture, software state (e.g., firmware

version, initial configuration), and CPU model of the IoT devices (if devices are homogeneous, it is sufficient for the verifier to store the configuration of a single IoT device). Such information can be obtained during deployment. We assume that the sink node is trusted and communication between devices in the network and the outside world passes through it [97].

4.2.3 *Threat Model*

Wang et al. [112] classify non-physical attacks along three dimensions. With respect to those dimensions, the attacks considered in this chapter are as follows:

1. Outsider vs insider attacks: We consider *outsider* attacks since several types of IoT malware were injected remotely to control devices and launch large-scale DDoS attacks [72]. The compromised devices can collude with each other to evade detection. To handle insider attacks (impersonation and proxy attacks), nodes typically require secret keys which software-based solutions cannot securely store [12]. However, additional mechanisms can be incorporated along with remote attestation to handle insider attacks [80].
2. Passive vs active attacks: We consider both *active* and *passive* malware attacks. Active malware can modify/drop any packets received or generated. In contrast, passive malware does not interfere in the protocol operation. The attacker can also use protocol messages to launch DDoS attacks in the network and render the system unresponsive.
3. Mote-class vs laptop-class devices: The attacker can use either a *mote-* or *laptop-class* device to infect the devices in the network with malicious software as long as the used device is not part of the network (cf. inside attacker).

4.2.4 *Problem Statement*

We consider an IoT system where an attacker can compromise several devices by injecting malicious code into them using vulnerabilities in the software running on the devices. A trusted verifier wants to collectively verify the integrity of the IoT devices. Our objective is to cater to the current needs that are applicable to legacy devices and next generation IoT applications without trusted hardware or modified hardware.

We are interested in practical solutions, i.e., solutions that perform well with respect to the following metrics:

Overall communication overhead: The total number of packets transmitted in the network during remote attestation execution. A high communication over-

head has a direct impact on the energy consumption [46]. Hence, an effective attestation protocol should have low communication overhead.

Overall attestation time: The time taken to attest all the nodes in the network. It is the total time duration between the verifier sending an attestation request to the first node in the network and the response received from the last node.

4.3 OVERVIEW

In this section, we provide an overview of the proposed system – SWARNA– SoftWAre-based Remote Network Attestation and thereby how the system addresses the challenges in designing a software-based swarm attestation protocol.

4.3.1 *Monitoring and Control*

SWARNA, a pure software-based solution for attesting swarms of IoT devices proposed in this chapter, does not depend on any trusted hardware for generating attestation results. Instead, SWARNA uses the checksum of the device calculated by traversing the memory in pseudo-random pattern [98] (cf. **C1**). The checksum must be received at the verifier within strict time bounds to identify the malicious node. We thus build a deterministic communication path enforcing time bounds across a wireless multi-hop IoT network for attestation purposes (cf. **C2**). The communication path is built by a centralized component, *monitoring & control unit* as shown in Figure 6a. Monitoring & control unit has similar capabilities as the verifier and the two components can be deployed on a single or separate machines. Routing information transmitted to the nodes are signed by monitoring & control unit and hence a malicious node cannot deliberately change routing paths used for attestation. The communication path built by monitoring & control unit can co-exist with other paths constructed by any distributed or centralized routing schemes. However, a malicious relay node can still drop/delay/modify results of descendant nodes to distract from its own incorrect state. If the verifier does not receive an attestation response, it is difficult to determine whether there was a packet drop due to poor channel condition or a malicious activity. A solution for swarm attestation must be able to eventually verify all devices even in lossy networks, i.e., losses should not affect security but only efficiency.

4.3.2 *Attestation Protocols*

We present two software-based attestation protocols to verify remote IoT devices (cf. **C3**). Existing over-the-air reprogramming methods [99] can update the devices to use SWARNA.

4.3.2.1 *SWARNA-individual*

In SWARNA-individual (SWARNA-ind), the verifier attests devices individually which can be several hops away from the verifier. Each trigger for attestation verifies all the nodes and incurs high cost in terms of attestation time and communication overhead. We thus propose SWARNA-aggregate (SWARNA-agg).

4.3.2.2 *SWARNA-aggregate*

SWARNA-agg aggregates the attestation responses at intermediate nodes minimizing attestation time and communication overhead. SWARNA-agg works in two phases to deal with false suspicions or indecision caused due to aggregation. The second phase is invoked only on suspected nodes, if there are any. The overhead increases with the number of malicious nodes depending on the network topology. This makes SWARNA-agg more suitable for networks with fewer malicious nodes or when attestation is performed very often as part of network maintenance. Evaluation results show SWARNA-agg performs better than SWARNA-ind as long as the number of malicious nodes are below 50% and 70% for higher and lower transmission powers respectively. Aggregation reduces attestation time and communication overhead, hence, in medium to large IoT deployments like smart grid, agriculture monitoring, or industrial IoT application, SWARNA-agg can be deployed. For small networks like smart home where devices are usually directly reachable from the sink SWARNA-ind can be used to verify the integrity of devices.

4.4 DETERMINISTIC COMMUNICATION PATHS

We discuss the impact of network reliability on identifying malicious nodes and mechanisms for creating deterministic communication paths.

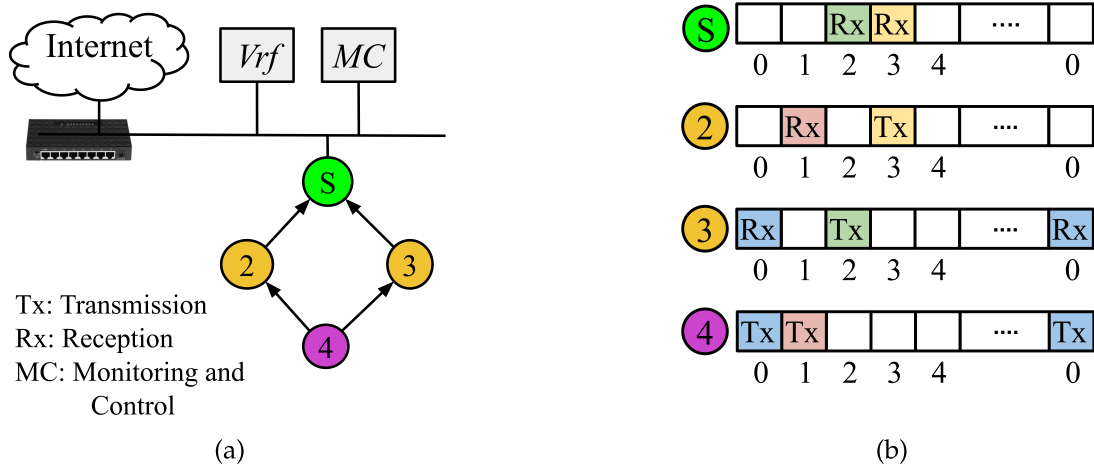


Figure 6: (a): A simple 4 node network (1 sink, 1 leaf, and 2 relay nodes), the verifier, and monitoring & control unit. (b): Example of TSCH schedules created by monitoring & control unit for the network shown in a.

4.4.1 *Network Reliability and False Positives*

A malicious node can drop packets of itself or its descendants. However, it is not possible to determine remotely whether a missing packet is due to malicious activity or just poor channel conditions. In order to identify a benign node during attestation, the verifier should receive the attestation response from the node, lest that node be suspected eventually. Under poor channel conditions, many false positives will occur. A highly reliable communication protocol can minimize these false positives.

Dedicated slots for communication can vastly improve the determinism in wireless settings. Centrally coordinated slot allocation mechanisms can eliminate the influence of an attacker in modifying the dedicated slots. However, such techniques incur very high overhead in dynamic wireless networks. Hence — for attestation purposes only — we allocate *centrally coordinated dedicated slots based on knowledge of physical topology*. For application traffic and all other purposes a distributed routing protocol can be used.

4.4.2 *TSCH Link-Layer Protocol*

We consider IEEE 802.15.4 Time Slotted Channel Hopping (TSCH) link-layer protocol [62], shown to achieve time-synchonized communication with reliability over 99.99% in real-world deployments [41, 40]. Several alternatives for

time-slotted communication in IEEE 802.11 networks [15] exist. TSCH being a widely accepted open-source standard simplifies deployment.

In a TSCH network, time is divided into *timeslots* which are combined to form a *slotframe* repeating continuously over time. Communication between nodes is orchestrated by a *schedule* which defines actions (transmit, receive, or sleep) to be performed in a particular timeslot. Figure 6 shows an example of dedicated timeslots for communication. In the example, node 4 transmits to node 3 in timeslot 0 and to node 2 in timeslot 1. The standard defines several mechanisms for creating and maintaining schedules [107].

Algorithm 1: Creating dedicated TSCH schedules on monitoring & control unit

Variables: Set of all nodes in network $nodes$
Schedule for a node N, $S_N[len]$

```
1 task CREATE_SCHEDULES()
2     foreach N ∈ nodes do
3         ts ← 0
4         foreach c ∈ CHILDREN(N) do
5             if S_N[ts] = FREE ∧ S_c[ts] = FREE then
6                 S_N[ts] ← Rx
7                 S_c[ts] ← Tx
8             ts ← ts + 1
```

4.4.3 *Generating TSCH Schedules for Attestation*

For attestation, SWARNA uses predetermined TSCH schedules — called *hard cells* in TSCH terminology — which are initialized and maintained by monitoring & control unit in a secure environment using cryptographic signatures and verification. Algorithm 1 shows a mechanism used in SWARNA to create dedicated TSCH schedules. Hop count is used as a metric to derive children and parent nodes. The physical topological information is considered for generating the schedules and not the communication topology. The communication topology can change drastically during the network lifetime. However, the physical node reachability can also change due to external noise, but less frequently. Such changes are learned through infrequent SETUP messages. Once the schedules are created, monitoring & control unit transmits the signed messages with schedules to each node in the network. TSCH allows several slotframes to operate simultaneously. Application traffic can use attestation

timeslots, a different scheduling mechanism like Orchestra [41], or Competition [52] that describes a simple approach for routing using a single shared timeslot.

With dedicated schedules and time synchronization, the end-to-end network reliability achieved is higher compared to contention-based networks. A highly reliable network aids in identifying malicious nodes in a vulnerable network with fewer false positives, that leads to unnecessary reprogramming of devices affecting the application's performance. However, poor reliability, inversely, does not incur false negatives in detecting malicious nodes – SWARNA identifies all the malicious nodes in the network (Figure 4.7.3).

4.5 SWARNA PROTOCOLS

In this section we present SWARNA, our approach to remote attestation of swarms of IoT devices. We propose two variants of SWARNA – SWARNA-individual and SWARNA-aggregate. Both rely on program memory checksum calculation. Several software-based remote attestation techniques in the literature provide algorithms for calculating such checksums, e.g., [98, 4, 12]. We use the state-of-the-art block-based pseudo-random memory traversal technique of AbuHmed et al. [4].

We use the following notations. $\{e_1, \ldots, e_k\}$ represents a set with elements $e_1 \ldots e_k$, $[f_1, \ldots, f_k]$ represents a tuple, e.g., a message m, containing fields $f_1 \ldots f_k$. A message m signed by the verifier with secret key vrf is represented as m_{vrf} and $|X|$ is the cardinality of a set X. Table 1 summarizes the notations used and the functions available to SWARNA.

4.5.1 *SWARNA-individual*

SWARNA-individual — SWARNA-ind for short — works by attesting devices individually which can be several hops away from the verifier. In SWARNA-ind, the verifier generates a signed attestation request $attReq$ for a given node N as

$$[N, nonce, seq, \text{REQ}]_{vrf} .$$

$nonce$ is a random value generated for N and seq indicates the attestation instance. The verifier sends $attReq$ to N and records the time as T^{start}. N accepts $attReq$ only if it is signed by the verifier and seq is greater than the values it has seen before. N uses the $nonce$ to calculate the checksum cs of its program memory and generates an attestation response $attResp$ as

$$[N, cs, \text{RESP}]$$

Notation	Description
the verifier	Verifier
$attReq$	Request generated by the verifier to initiate attestation process
$attResp$	Attestation response
m_{vrf}	Message m signed by the verifier's secret key vrf
N	Node ID
cs	Memory checksum of a node
$nonce$	Used in calculating the cs. Sent in $attReq$
seq	Indicates the instance of attestation process
T^{accept}	Time for a node to respond with $attResp$ for an $attReq$
T^{out}	Timeout after which received $attResp$ are aggregated
Node state	**Description**
Correct	Node whose software is in the expected state and that can generate a correct checksum cs within T^{accept}
Malicious	Node compromised by an attacker that cannot generate a correct checksum cs within T^{accept}
Suspected	Node suspected to be malicious
Function name	**Description**
CHKSUM($nonce$)	Returns checksum of attestable memory calculated for challenge $nonce$
VERIFY_REQ(m)	Returns TRUE iff m is signed by the verifier
MSG_TYPE(m)	Returns $type$ of m. $type$ is REQ or RESP for SWARNA-ind, and REQ_PI, RESP_PI, REQ_PII, or RESP_PII for SWARNA-agg. SETUP identifies network setup requests
CURR_TIME()	Returns current system time
SEND(m, N)	Unicasts message m to single node N
BCAST(*msg*, $\mathcal{N}$)	Broadcasts message m to nodes $\mathcal{N}$={$N_1 \ldots N_k$ }
SINK_ID	Yields identifier of the sink
DEF_PARENT(N)	Returns default parent of node N
PARENTS(N)	Returns all immediate parent nodes of N
CHILDREN(N)	Returns all immediate child nodes of N
DESCEND(N)	Returns all nodes for which N is a relay
FUNC_ENTERED(m)	For receiving a message m
DEREF(m, *type*)	Decomposes m based on $type$ to individual fields

Table 1: Generic notation, node states, and standard functions.

and forwards it to the verifier through its default path. The verifier records the time when it receives attResp as T^{end}. Algorithm 2 shows the detailed algorithm for N.

attResp is accepted *only* if following conditions are met:

1. $T^{end} - T^{start} \leqslant T^{accept}$, where $T^{end} - T^{start}$ includes time to transmit and verify attReq, calculate cs, and transmit attResp.
2. The received cs matches the cs generated by the verifier on its site using the same nonce sent to N in attReq

If N responds with a wrong checksum or does not reply within T^{accept}, the verifier can re-program N using over-the-air re-programming techniques [99]. Re-programming is required to ensure that an intermediate node does not modify the response of any descendant nodes. A malicious node triggering such an operation will be ignored by a remote node due to signature verification failure. The verifier repeats the above-mentioned procedure by first verifying nodes at one hop away from sink, then nodes at two hops etc.

Algorithm 2: SWARNA-ind on node N

```
Variables: Current sequence number curSeq
1 upon FUNC_ENTERED(m)
2   if VERIFY_REQ(m) ∧ MSG_TYPE(m) = REQ then
3     [N, nonce, seq] ← DEREF(m,REQ)
4     if seq > curSeq then
5       curSeq ← seq
6       cs ← CHKSUM(nonce)
7       attResp ← [N, cs, RESP]
8       SEND(attResp, SINK_ID)
```

4.5.2 *SWARNA-aggregate*

SWARNA-ind attests nodes individually which is suboptimal in terms of communication overhead and attestation time in many scenarios. SWARNA-aggregate — SWARNA-agg for short — attempts to overcome the limitations of SWARNA-ind without compromising on the security guarantees.

In short, SWARNA-agg works in two phases to identify all malicious nodes in the network. The first phase (Phase I) follows a bottom-up approach and marks all the nodes in the network as correct, malicious, or suspected. At the end of Phase I, if all the nodes are identified as correct or malicious then the protocol terminates. If not, in order to mark all suspected nodes as either correct

or malicious, a second phase (Phase II) is triggered for the suspected nodes following a top-down approach. Figure 7 shows an overview of the protocol used in SWARNA-agg.

Algorithm 3: Calculating T^{out} on node N

```
Variables: Set of child nodes child
           Time to attest a 1-hop node (configured) T^attest
           Number of hops from the sink hop
           TSCH timeslot duration T^slot
1  upon FUNC_ENTERED(m)
2    if MSG_TYPE(m) = SETUP ∧ |CHILDREN(N)| = 0 then
3      T^out ← 0
4      resp ← [N, hop, SETUP]
5      BCAST(resp, PARENTS(N))
6    else if MSG_TYPE(m) = SETUP ∧ |CHILDREN(N)| > 0 then
7      [N_i, hop_i] ← DEREF(m, SETUP)
8      child ← child ∪ {N_i}
9      k ← |child|
10     if k = |CHILDREN(N)| then
11       T^out ← MAX(hop_1, ..., hop_k) × T^attest × T^slot
12       resp ← [N, hop, SETUP]
13       BCAST(resp, PARENTS(N))
```

4.5.2.1 *Aggregation and aggregation timeout*

In order to reduce communication overhead, a relay node in SWARNA-agg aggregates the checksums received in the responses from its child nodes and forwards a single response representing the software states of all its descendant nodes. If the verifier receives a wrong aggregated result, identifying malicious nodes from the aggregated result is generally impossible. To address this, each node transmits the response to all of its parent nodes instead of just the default parent. Figure 5 shows an example of alternate paths available for each node. This redundancy is exploited to better identify malicious nodes. If there are several malicious nodes in a given sub-tree, the degree of redundancy required to uniquely identify the malicious nodes is equal to the number of malicious nodes.

Each node is configured with an aggregation timeout T^{out}, which represents the maximum time it takes for an attResp from a farthest descendant node to

reach the node. A simple methodology to calculate T^{out} on a node is shown in Algorithm 3. Time to attest a 1-hop node, T^{attest}, is a configurable parameter which remains constant for a given hardware.

Algorithm 4: SWARNA-agg on leaf node N

```
Variables: Flag PIDone ← FALSE for Phase I completion
           Current sequence number curSeq
1  upon FUNC_ENTERED(m)
2    if VERIFY_REQ(m) ∧ MSG_TYPE(m) = REQ_PI then
3      [nonce, seq] ← DEREF(m, REQ_PI)
4      if seq > curSeq then
5        curSeq ← seq
6        cs ← CHKSUM(nonce)
7        attResp ← [N, cs, RESP_PI]
8        BCAST(attResp, PARENTS(N))
9        PIDone ← TRUE
10   else if PIDone ∧ MSG_TYPE(m) = REQ_PII then
11     attResp ← [N, cs, RESP_PII]
12     SEND(attResp, SINK_ID)
13     PIDone ← FALSE
```

4.5.2.2 *Phase I - bottom-up swarm attestation*

Phase I follows a bottom-up approach where attReq is processed from leaf nodes through the relay nodes up to the sink. The verifier generates a signed attReq in Phase I as

$$[\mathtt{nonce}, \mathtt{seq}, \textsc{req_pi}]_{vrf}$$

where nonce is a random value for attesting the network and seq indicates the attestation instance. The verifier sends attReq to the sink node which is broadcast to its neighbors. When a relay node receives attReq, it records the local time as T^{req} and further broadcasts attReq unaltered. Algorithm 4 shows the algorithm for leaf nodes. When leaf node N receives attReq, it verifies the signature and processes attReq. N uses the nonce to compute the checksum of its program memory. Leaf node N generates attResp in Phase I as

$$[N, cs, \textsc{resp_pi}]$$

and broadcasts it to every node for which N is a child node. attResp is transmitted to all parent nodes to allow the verifier to unequivocally identify the malicious nodes (cf. Section 4.5.2.1).

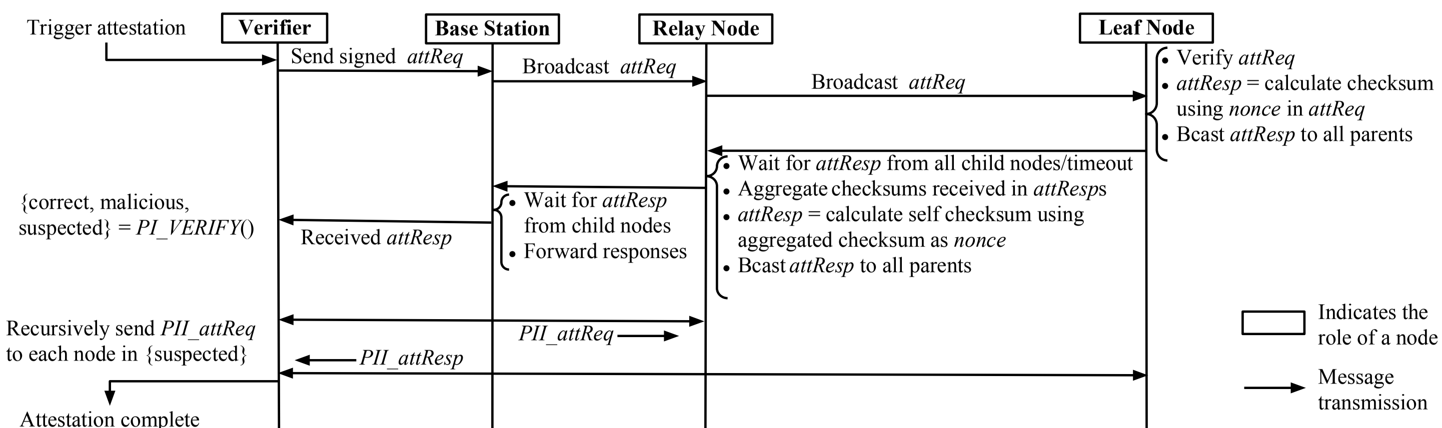


Figure 7: Overview of of SWARNA-agg. An external process or an administrator triggers attestation. Time proceeds from top to bottom. On receiving the trigger, the verifier initiates Phase I (bottom-up approach) of the protocol by sending a signed `attReq`, which is broadcast in the network. The `attResps` are aggregated along the path from leaves to the base station. On receiving aggregated `attResps`, the verifier executes Phase I verification to identify correct, malicious, and suspected nodes. If there are any suspected nodes, Phase II (recursive top-down approach) is initiated to mark all suspected nodes as either malicious or correct.

Algorithm 5: SWARNA-agg on relay node N

```
Variables: Set of checksums from children csChild
           Flag PIDone ← FALSE for Phase I completion
           Aggregation timeout timer ← 0
1  upon FUNC_ENTERED(m)
2    if MSG_TYPE(m) = REQ_PI then
3      T^req ← CURR_TIME()
4      BCAST(m, CHILDREN(N))
5    else if MSG_TYPE(m) = RESP_PI then
6      T^resp ← CURR_TIME()
7      if (T^resp − T^req) > T^reject then
8        raise "unsolicited RESP" alert
9        return
10     [N_i, cs_i] ← DEREF(m, RESP_PI)
11     if (N_i ∈ CHILDREN(N)) then
12       csChild ← csChild ∪ {cs_i}
13       k ← |csChild|
14       if timer = 0 then
15         timer ← CURR_TIME()
16     else
17       return
18     attime ← CURR_TIME() − timer
19     if k = |CHILDREN(N)| ∨ attime > T^out then
20       aggcs ← cs_1 ⊕ ... ⊕ cs_k
21       cs ← CHKSUM(aggcs)
22       attResp ← [N, cs ⊕ aggcs, RESP_PI]
23       BCAST(attResp, PARENTS(N))
24       PIDone ← TRUE
25       timer ← 0
26   else if PIDone ∧ MSG_TYPE(m) = REQ_PII then
27     attResp ← [N, cs, {cs_1, ..., cs_k}, RESP_PII]
28     SEND(attResp, SINK_ID)
29     PIDone ← FALSE
```

The algorithm for intermediate relay nodes is given in Algorithm 5. Consider a relay node N with k child nodes. N processes $attResp$ only if it has seen an $attReq$ in latest T^{reject} time duration, otherwise it raises an unsolicited message received as an action to curb DDoS-attacks where the nodes are tricked into performing attestation (line 8). T^{reject} is generally set to twice T^{out} with some additional constant time for calculating the checksum. N waits for responses from all its k child nodes $N_1 \dots N_k$ or until T^{out}— derived during network setup — expires. N aggregates the received checksums to use it as a nonce to calculate its own cs (line 20). Relay node N generates $attResp$ in Phase I as

$$[N, cs \oplus aggcs, \textsc{resp_pi}]$$

where $\oplus$ represents an XOR operation, and then broadcasts $attResp$ to all its parent nodes (line 21 – line 23). The sink forwards the response packets to the verifier for verification. The number of response packets the sink receives is at most equal to the number of its child nodes.

Verification. The verifier receives $attResp$ for a node from multiple unique paths, if available. As discussed in Section 4.5.2.1 verifier uses redundant paths available in the network to mark unambiguously correct and malicious nodes, and remaining nodes for which a decision cannot be made due to insufficient redundancy are marked as suspected (cf. Table 1). The verifier triggers a Phase II top-down attestation to mark the suspected nodes as unambiguously either correct or malicious.

4.5.2.3 *Phase II - top-down swarm attestation*

In Phase II, the verifier recursively performs top-down attestation on the suspected nodes. The verifier unicasts $attReq$ to each suspected node in increasing order of its hop distance as

$$[N, \textsc{req_pii}]$$

N processes Phase II $attReq$ only if it has completed Phase I and generates a response $attResp$ as

$$[N, cs, \{cs_1 \dots cs_k\}, \textsc{resp_pii}]$$

where $\{cs_1 \dots cs_k\}$ are the checksums received from the child nodes of N, $\{N_1 \dots N_k\}$.

Verification. The recursive algorithm used by the verifier to perform top-down attestation is given in Algorithm 6. It distinguishes three cases – a node N reported either

i) the correct checksum cs: In this case the verifier simply marks N and all its descendants until the leaf node as correct (line 15). N aggregates the measurements received from its child nodes within acceptable time and uses it as $nonce$ to calculate its own measurement, so if N reports a correct checksum cs then its child nodes must be correct. Recursively all their children must also be correct.

ii) the wrong checksum cs but N is correct: This happens when a descendant node is malicious and generated a wrong cs or failed to generate a checksum within acceptable time. N will report a measurement calculated using a wrong nonce causing it to be suspected by the verifier. In this case, the verifier aggregates $cs_1, \dots, cs_k$ reported by N in Phase II $attResp$ to re-calculate the checksum (line 20); the checksum is used to verify if N (line 24) or any of its descendants (line 30) are malicious. It is necessary for the verifier to execute the attestation routine without which it is impossible to locate the malicious node.

iii) the wrong checksum cs and N is malicious: the re-calculated cs by the verifier for N (line 20) will not match cs reported by N. The verifier marks N as malicious and further verifies N's child nodes recursively. (line 25 – line 30).

4.5.3 *Theoretical Analysis*

We theoretically compare communication overhead and attestation time of SWARNA-agg and SWARNA-ind. Consider a tree with depth d, branching factor w and a total of n nodes. We assume the following definitions will remain constant for a network:

- T^{sigv} is signature verification time,
- T^{cs} is the time for calculating checksum,
- l is the communication latency per hop which is sum of queuing delay and time to transmit a packet

4.5.3.1 *Complexity of SWARNA-ind*

Attestation time. SWARNA-ind attests each device individually in the increasing order of its *depth* from the sink:

$$T^{att} = n \times \left(2 \times (l \times d) + T^{cs} + T^{sigv}\right) \quad (1)$$

Algorithm 6: SWARNA-agg Phase II verification on the verifier

Input : Set of suspected nodes from Phase I $suspect$
Variables : Node ID sets $correct$, $malicious$
Received/expected checksum at the verifier rcs, ecs
Number of child nodes k
Receiver timeout T^{recv}

```
1  task PHASE_II_VERIFY(suspect)
2    foreach N ∈ suspect do
3      if N ∉ correct then // Can change handling prev.
4        [newC, newM] ← REC_VERIFY(N, suspect)
5        correct ← correct ∪ newC
6        malicious ← malicious ∪ newM

7  function REC_VERIFY(N, suspect)
8    corr, mal ← ∅
9    SEND(N, REQ_PII)
10   timer ← CURR_TIME()
11   wait until FUNC_ENTERED(m) ∨ (CURR_TIME() - timer) > T^recv
12   if m = NULL then
13     return [corr, mal]
14   [N, rcs, {rcs_1, ..., rcs_k}] ← DEREF(m, RESP_PII)
15   if rcs = ecs then
16     corr ← corr ∪ {N} ∪ DESCEND(N)
17   else if k = 0 then
18     mal ← mal ∪ {N}
19   else
20     cs ← CHKSUM(rcs_1 ⊕ ... ⊕ rcs_k)
21     if rcs = cs then
22       corr ← corr ∪ {N}
23     else
24       mal ← mal ∪ {N}
25     foreach N_c ∈ CHILDREN(N) do
26       if N_c ∈ suspect then
27         if rcs_c = ecs_c then
28           corr ← corr ∪ {N_c} ∪ DESCEND(N_c)
29         else
30           [corr, mal] ← REC_VERIFY(N_c, suspect)
31   return [corr, mal]
```

Since $n = (1 + 2 + \ldots + 2^d)$, we have $d = \log n$. Therefore, for SWARNA-ind the complexity of T^{att} is $O(n \log n)$.

Communication overhead. In the worst case a node may transmit n requests and n response packets making the complexity $O(n^2)$.

4.5.3.2 *Complexity of SWARNA-agg*

Attestation time. In a time-synchronized tree, all leaf nodes at same depth from the sink will receive $attReq$ simultaneously and calculate their checksums by end of T^{cs} . The nodes at same depth will compete with $w - 1$ sibling nodes for transmission. Hence, time taken to transmit $attResp$ by a node with w child nodes is $T^{cs} + w \times l$. The attestation time for Phase I is thus as follows:

$$T^{att_PI} = \underbrace{l \times d}_{attReq} + \underbrace{(T^{cs} + w \times l) \times d}_{attResp} + T^{sigv} \tag{2}$$

Therefore, the complexity of T^{att_PI} will be $O(\log n)$.

The attestation time for Phase II, T^{att_PII}, depends on the number of suspected nodes and their depth from the sink. In the worst case where all n nodes are suspected, a Phase II $attReq$ is sent to n nodes in the network and Phase II $attResp$ is received. (cf. Section 4.5.2.3). The maximum attestation time of Phase II is thus:

$$T^{max_PII} = n \times \big(2 \times (l \times d)\big) \tag{3}$$

From Equation 2 and Equation 3, we have

$$T^{att} = T^{att_PI} + T^{att_PII} \text{with } 0 \leqslant T^{att_PII} \leqslant T^{max_PII}$$

Communication overhead. To generate a signed $attReq$ we consider a 256 bit elliptic curve digital signature [66] that generates a 32B output. The payload size of $attReq$ is 38B (1B seq, 4B $nonce$, 1B label and 32B signature). $attResp$ contains a 1B label, 1B node ID (for up to 255 nodes network) and 8B checksum, making it a 10B packet. Considering the header size of the protocols shown in the protocol stack in Figure 8, the maximum payload of a IEEE 802.15.4 frame is 102B [62]. Hence, our requests and responses are transmitted without any fragmentation. In Phase I of SWARNA-agg, every node in the network will transmit one request and one response packet and Phase II transmissions depend on the number of suspected nodes. Therefore, the complexity of communication overhead will be $O(n + s^2)$ where s is the number of suspected nodes.

SWARNA
Transport - UDP
Network - RPL (IPv6)
MAC - TSCH
PHY - IEEE 802.15.4

Figure 8: Protocol/software stack on IoT devices.

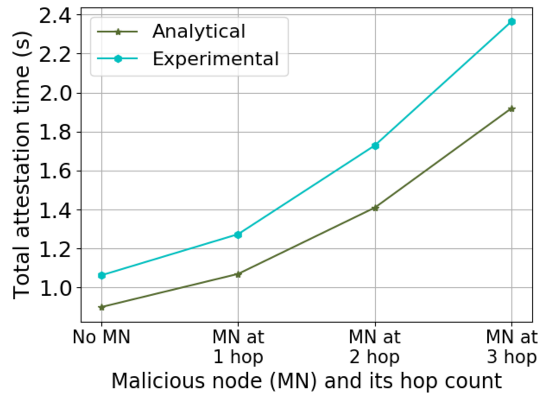


Figure 9: Analytical vs simulation results for a perfect binary tree with maximum depth 3 for SWARNA-agg.

4.6 IMPLEMENTATION

IoT nodes are implemented on Contiki, an open-source OS for resource-constrained embedded devices and IoT [39]. Contiki code is written in C and the verifier in Python.

The protocol stack of a Contiki node is shown in Figure 8. IEEE 802.15.4-2015 describes physical and MAC layer protocols for resource-constrained devices [62]. TSCH is used in all our experiments and monitoring & control unit creates TSCH schedules for each node (cf. 4.4.2). In order to generate a network topology we use IPv6 Routing Protocol for Low Power Lossy Networks (RPL) [6], which forms a directed acyclic graph rooted at the sink. RPL computes a rank for each node using a metric. For a given node, neighboring nodes with lower rank are parent nodes, and nodes with higher rank are child nodes. We use hop count as a metric for RPL.

SWARNA is implemented as an UDP application. In SWARNA-ind, `attReq` for nodes at the same hop level are sent in parallel. Only after verifying these nodes, higher hop distance nodes are attested. In SWARNA-agg, Phase II `attReq` for 1-hop nodes are sent in parallel and hence all sub-trees rooted at 1-hop nodes are processed in parallel.

4.7 EVALUATION

We evaluate SWARNA on a testbed and simulator, compare it to analytical results and related work.

4.7.1 *Analytical Evaluation vs Simulation Results*

We compared the analytical with simulation results for a tree with branching factor $w = 2$ and depth $d = 3$. l is the sum of queuing delay T^q and per hop transmission time T^{tx}. We use T^{tx}= 10ms, the default length of a timeslot used in TSCH frame [40]. T^q of 110ms is used for a slotframe length of 11. For this comparison we ignore $T^{cs}$and T^{sigv}, as they remain constant for a node type. Figure 9 shows the analytical and simulation results for attestation time which are an average of 50 runs. The slope of the simulation results closely matches with the analytical results. With 0 malicious nodes, only Phase I of SWARNA-agg will be executed. The observed difference between the two techniques is due to the T^q. During experiments each packet experiences a different T^q, whereas a constant T^q was considered for analytical results.

4.7.2 *Empirical Evaluation*

We present the experimental setup followed by the performance of SWARNA and comparison of SWARNA with the state-of-the-art swarm attestation techniques.

4.7.2.1 *Experimental setup on testbed*

We evaluate SWARNA on the FIT IoT-LAB testbed at the Grenoble site where the nodes are deployed in a typical office environment [5]. The site also has significant noise due to the Wi-Fi access points. We choose 30 nodes such that multi-hop topologies can be formed. We vary the transmission power (3dBm, -9dBm, -17dBm) to generate different topologies with 2, 3 and 4 hop networks. Nodes start with a single slotframe containing one shared timeslot (contention based) as described in 6TiSCH minimal configuration [40]. Application and control traffic are transmitted over this shared timeslot. After a certain amount of network settling time we initiate the attestation process. As a first step, monitoring & control unit sends a request to each node to add a slotframe for attestation with TSCH schedule generated by monitoring & control unit (cf. Figure 6).

We use M3 nodes on the testbed for evaluation. These have an ARM Cortex-M3 micro-controller operating at 72MHz. Calculating the memory checksum for 128KB of memory takes 0.3s, and verifying a signature that uses 256bits elliptic curve digital signature takes 0.01s.

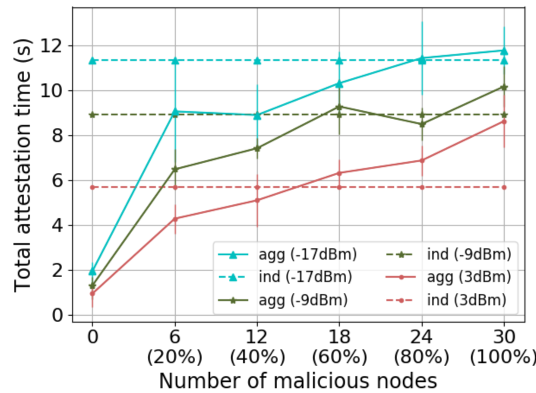


Figure 10: Attestation time for a 30 nodes network on IoT-LAB testbed, with varying numbers of malicious nodes and transmission powers. Networks of depth 2, 3 and 4 were generated with 3, -9 and -17dBm transmission power. A trade-off between x-axis and y-axis for same transmission ranges can be seen at the intersection of the lines. Up to 50%-80% of malicious nodes in the network, SWARNA-agg has lower attestation time

4.7.2.2 *Overall attestation time*

The performance of SWARNA-agg depends on the number of malicious nodes and their hop counts. Hence, for a given topology we randomly choose a varying number of malicious nodes. We repeat the experiment for 10× and calculate the mean and variance with 95% confidence interval.

Malicious nodes and transmission ranges. Figure 10 shows attestation time for SWARNA-ind and SWARNA-agg. Attestation time of SWARNA-ind does not change with the number of malicious nodes as it attests the nodes individually. If there are malicious nodes in the network and SWARNA-agg cannot make a definite decision in Phase I, Phase II is executed on the suspected nodes, leading to an increase in attestation time. In Phase I, nodes at the same hop distance from the sink node calculate their memory checksums at roughly the same time and also, broadcast `attReq` and `attResp`. As a consequence, Phase I has less impact on attestation time performance compared to Phase II. In Phase II, the request is sent to each suspected node. After a certain point attestation time of SWARNA-agg exceeds that of SWARNA-ind because Phase I of SWARNA-agg does not detect any node as malicious and invokes Phase II on all the nodes with additional, little, time wasted in Phase I.

We also increase the transmission power of each node which increases the number of neighbors, thereby reducing the number of hops to the sink node.

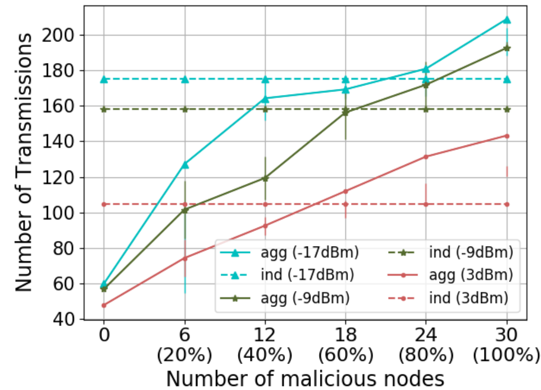


Figure 11: Communication overhead for a 30 nodes network in IoT-LAB testbed, with varying numbers of malicious nodes and transmission powers. Networks of depth 2, 3 and 4 were generated with 3, -9 and -17dBm transmission power. A trade-off between x-axis and y-axis can be seen at the intersection of the lines for same transmission ranges. For up to 50%-75% of malicious nodes in the network, SWARNA-agg has lower overall communication overhead (CO).

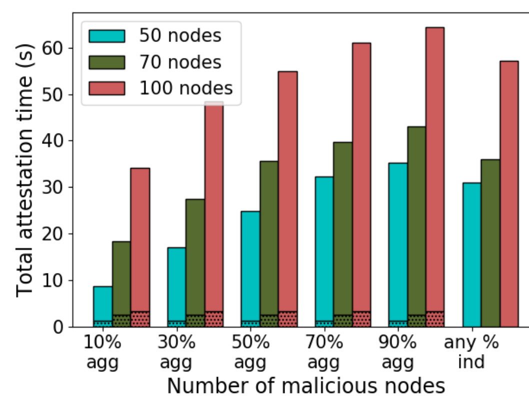


Figure 12: Attestation time for different network sizes in the Cooja simulator. For each bar, the dotted region represents the time taken by Phase I of SWARNA-agg, and the remaining time is spent on Phase II. In SWARNA-ind varying the number of malicious nodes has an effect on the overall attestation time (AT), unlike for SWARNA-agg.

Hence, we see that attestation time decreases as the number of hops to the sink decreases.

Network size and breakdown. In order to show the performance of SWARNA as the network size increases, we run simulations in the Cooja simulator [92], due to unavailability of large numbers of nodes for long durations. Cooja emulates the Contiki nodes at hardware level. Figure 12 shows the breakdown of time spent on Phase I and Phase II with SWARNA-agg. SWARNA-agg with 10% of malicious nodes in the network takes about 40% - 60% less time for attesting the entire network compared to SWARNA-ind.

4.7.2.3 *Overall communication overhead*

Figure 11 shows SWARNA-agg has less communication overhead when fewer nodes are malicious, as Phase II is invoked fewer times and in Phase I every node transmits a single aggregate response. Lower communication overhead implies lower energy consumption. It has been clearly shown [46] that aggregation reduces energy spent by the nodes, as energy required for aggregation operations is less compared to communication (Tx + Rx).

4.7.2.4 *Choice: SWARNA-ind vs SWARNA-agg*

With lower communication overhead and attestation time, SWARNA-agg represents a more energy-efficient choice when there are fewer expected malicious nodes. Earlier work in swarm attestation has considered less than 1% of malicious nodes in evaluation [8]. However, it is obviously difficult to obtain the knowledge of how many nodes will be compromised ahead of time. In general, network attestation is triggered periodically with varying frequency, e.g., once an hour, once a day. Depending on the attestation frequency one can make a choice of using SWARNA-agg or SWARNA-ind. However, SWARNA-agg is suitable for most use cases considering the usual behavior of IoT malware – typically it is installed on a victim device by the attacker and is used as a bot to launch large-scale attacks days, weeks and sometimes years later only.

4.7.2.5 *SWARNA and state-of-the-art swarm attestation*

Next we compare SWARNA with state-of-the-art techniques for IoT swarm attestation – LISA [23] and SANA [8]. Both work only on secure hardware. In terms of attestation time SANA reports 2.5s for attesting a million devices in its target setting and LISA-α reports in the best case 0.5s for attesting a 40 nodes random network, both on a simulator. Raspberry Pis operating at more than

	LISA-α	LISA-S	SANA	SWARNA-ind	SWARNA-agg
CO	$O(n^2)$	$O(n)$	$O(n)$	$O(n^2)$	$O(n+s^2)$
AttReq	43	47	$20g+78$	$H+39$	$H+38$
AttResp	79	$47+4y$	$32+32w$ + $20m$	$H+10$	$H+10$

H = 54B which includes IPv6, UDP, and MAC headers; s is the number of suspected nodes; y is the number of descendants of a node transmitting the message [23]; w and m are the number of distinct OAS public keys and software configurations of malicious nodes respectively; g is the number of benign software configurations [8].

Table 2: Overall communication overhead (CO) and message size (bytes) of swarm remote attestation techniques for a network of n nodes.

700MHz were considered in case of LISA and a proprietary research platform in SANA for emulating cryptographic operations. SWARNA in best case takes 1.6s for attesting a 30 nodes network. Note however that in contrast to other swarm attestation techniques, we evaluated the checksum calculation on a low-end IoT device with only 72MHz CPU. We gauge communication overhead via communication complexity and message sizes. Table 2 shows the communication complexity that includes the number of $attReq$ and $attResp$ transmitted and message sizes of various approaches. The complexity of SWARNA is comparable to previous approaches and provides a similar performance without special hardware. Also, in case of SANA and LISA-α, the message sizes depend on neighboring nodes and descendant nodes respectively which can vary with the network size. SWARNA, instead, has constant payload size.

4.7.3 *Use Case: Data Collection*

In typical IoT use cases like smart grids, network monitoring, or industrial IoT, periodic data collection for further processing is a common task. In this section, we implement a simple UDP data collection application and perform periodic attestation to study its effect on application performance. We evaluate the use case on FIT IoT-LAB testbed on a 22 nodes network with 12 source nodes generating UDP packets every 30s destined to the sink. 5 nodes (20%) are infected with a passive malware. Two TSCH slotframes are added, one for application and control traffic based on 6TiSCH minimal and the second

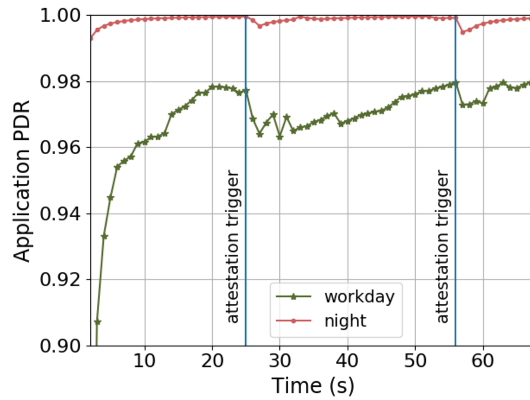


Figure 13: PDR of a data collection application for a 22 nodes 3 hop network in the IoT-LAB testbed with 12 source and 9 relay nodes. In a noisy environment, application PDR drops up to 1.4% and drops by 0.4% when noise is minimal during attestation. PDR recovers faster when noise is minimal.

slotframe for attestation packets (cf. Section 4.4). SWARNA-agg is used for attestation.

Impact of attestation on application. Figure 13 shows the packet delivery ratio (PDR), the ratio of number of packets received at the sink and the number of packets sent by all source nodes. An attReq is sent at 24 and 56 minutes. The effect of attestation on application PDR is evaluated under two different settings: (1) During a working day where the external noise is high due to experiments running on other neighboring nodes, actively used access points and also maybe people movement as the chosen testbed is deployed in the office setup. (2) At night when there is minimal activity and external noise. The application PDR during working day drops by 1.4% compared to just 0.4% at night. Application traffic using contention-based communication is affected by the noise compared to attestation packets using dedicated timeslots. Also, attestation packets have priority over application packets.

False positives due to packet loss. A malicious relay node can drop packets from its child nodes or drop its own attestation packets. However, the verifier cannot differentiate these packet drops from the packets lost due to poor channel conditions. Hence, the nodes from which the packets are not received are suspected as malicious. In this experiment, we evaluate the rate of false positives, where a benign node is suspected due to poor channel conditions. The verifier sends a Phase II attReq to all the nodes and calculates the number of attResp packets received within acceptable time. We carry out the experiment on a working day when external noise is high and also at night where the noise

is minimal. The results are an average over 10 trials. The timeout value for a node N is calculated as:

$$T^{accept} = \text{round trip time from } verifier \text{ to sink } + (2 \times \text{hop count of N} \times \text{slot frame length}) + T^{attest}$$

At night, there were no benign nodes that were falsely identified as malicious (0 false positive rate) and during a work day a 1-hop relay node was falsely identified as malicious due to delayed packet reception at the verifier (4.5% false positive rate). There were no false negatives i.e., no malicious node remained undetected. Attestation packets have dedicated timeslots (Section 4.4) with high priority and no contention and hence we see very few attestation packets lost resulting in low false positive rate. Our observations are in line with literature showing 99.99% reliability of TSCH even under interference and in large networks [41, 40], confirming our design.

4.8 SECURITY ANALYSIS

SWARNA secures against network attacks mentioned in Section 4.2.3 as follows:

4.8.1 *Passive Malware*

Passive malware responds to $attReq$ from the verifier and does not interfere with the attestation. The presence of malware results in an unexpected memory checksum which can be detected by comparing the individual $attResp$ with the expected value on the verifier in case of SWARNA-ind. SWARNA-agg will detect a malicious node either in Phase I or Phase II depending on depth and number of unique redundant paths between the node and sink (line 18 and line 24 in Algorithm 6).

4.8.2 *Active Malware*

Active malware can perform a number of activities discussed in turn in the following:

Modifying $attResp$ **received from descendant nodes.** A descendant of a malicious node which does not have an alternate correct path is suspected along with the malicious node in Phase I of SWARNA-agg. During Phase II the verifier will re-generate the checksum on its site with the nonce the malicious node claims to have received (line 20 in Algorithm 6). Consider the expected

checksum is cs and checksum of the node with malware is mcs. For the malicious node to prove $mcs = cs$, it should identify a nonce that would output cs as a result of pseudo-random memory traversal. Generating such a nonce requires time and high processing power which is impractical for an IoT device. Hence, the malicious node reports mcs and the verifier will identify it as malicious.

Delaying/dropping packets from descendant nodes. Delayed packets are not accepted by parent nodes and are dropped. Assume that in Figure 5, node 6 is malicious and drops all packets from itself and its child nodes. In Phase II, node 6's parent, node 4 reports to the verifier that no response from node 6 was received in Phase I. In SWARNA-ind nodes at a one hop distance from the base station are attested and fixed before attesting nodes at a hop distance greater than one.

The attestation process has no means to distinguish a packet dropped due to poor channel condition or due to malicious activity and will thus suspect the node that generated the lost packet. This ensures that malicious nodes causing packet drops are always suspected and the false positive rate for a given application is discussed in Figure 4.7.3

Timing attacks. Due to varying communication latency or checksum calculation delay, the attacker can utilize any additional time to calculate the correct checksum. The checksum is calculated in pseudo-random fashion and hence the attacker cannot know in advance the memory location that will be read (Section 4.5). However, an active malware can pause checksum calculation at every memory read and if the memory location points to malware location, move the malware code to a temporary location in data memory and back.

The standard deviation to calculate the checksum of 128K code memory on a FIT-IoT LAB testbed node for 50 runs is 0.08ms. However, just read and compare operations on a 128K memory requires at least 62ms (3 cycles for single read and compare [10]). Hence, such an attack is not possible on SWARNA. The communication latency is calculated based on slot frame length and hop count. Any packet that takes more time is detected as malicious. We observed false positive rates of 0 and 4.5% under minimal and high external noise conditions respectively Figure 4.7.3.

Replay attacks. A nonce is added to $attReq$ and used in the process of calculating memory checksums. In SWARNA-agg a relay node uses the combination of checksums received from its child nodes as nonce. These nonces thwart replay attacks.

Disabling SWARNA. SWARNA can be disabled by disabling network communication or by modifying the firmware. Such attempts can be detected by the verifier through an absence of response to attReq or a wrong checksum in attResp respectively.

4.8.3 *DDoS Attacks*

A malicious node can trigger fake attReq or attResp to launch a DDoS attack, attempting to make the network unresponsive. SWARNA-ind drops attReq not signed by the verifier. In SWARNA-agg a fake attReq is broadcast until it reaches a leaf node, which drops it. A node raises an alarm if it receives an attResp without having seen a corresponding attReq recently (line 8 in Algorithm 5).

4.8.4 *Modifying Dedicated Slots*

A node accepts only signed requests from the verifier, hence any attempt to change the attestation schedule will be rejected. However, it is possible that a malware performs a malicious activity between two attestation rounds and removes itself. Such time-of-check to time-of-use attacks are also applicable to attestation schemes using trusted hardware [23, 8] and remain an open problem.

4.8.5 *Non-Persistent and Persistent Malware*

A non-persistent malware is stored on non-volatile memory of a device. Rebooting the infected device will remove the malware, however, the vulnerability exploited to infect the device can still be used to re-install the malware. SWARNA performs attestation on code memory, which usually is a volatile memory [25]. Hence, SWARNA cannot detect attacks that are non-persistent in nature. A well-known example of such a malware is Mirai [72]. SWARNA is aimed at detecting types of malware that are persistent and remains on the device forever. Several real-world attacks that install persistent malware on IoT devices and the catastrophic effects it has had are discussed in Section 4.1.1.

4.9 CONCLUSIONS

Remote attestation (remote attestation) determines the trustworthiness of resource-constrained devices. Attesting entire swarms rather than single de-

vices is required for IoT deployments. Existing work on swarm attestation however requires trusted hardware for remote attestation. These efforts do little to improve the situation of the many existing legacy deployments without such hardware or next generation IoT applications with low-end devices where it is challenging to provide trusted hardware.

This paper presented, to the best of our knowledge, the first, pure software-based swarm remote attestation technique. By using IEEE 802.15.4 time-slotted MAC protocol we can overcome the main limitations of existing software-based solutions. We described two protocol variants — SWARNA-ind and SWARNA-agg— showed their feasibility on an IoT testbed, and investigated trade-offs between the two. SWARNA-agg performs better than SWARNA-ind in terms of attestation time and communication overhead with less than 50%-70% malicious nodes in the network, making SWARNA-agg a good fit for frequent periodic remote attestation. Also, SWARNA maintains constant payload size in contrast to existing hardware-based swarm remote attestation works where it increases linearly with network size. We also demonstrate feasibility of SWARNA with a periodic data collection application, showing only minimal impact on the application's packet delivery ratio (0.4% drop).

Part II

MEMORY CORRUPTION ATTACKS

5 SPADE: SECURE PROGRAM ANOMALY DETECTION FOR EMBEDDED IOT DEVICES

CHAPTER CONTENTS

5.1 Introduction . . . 58
5.2 Problem Statement . . . 61
5.2.1 Threat Model . . . 61
5.2.2 Security Goals . . . 63
5.2.3 Trusted Hardware (and) Challenges . . . 64
5.3 Overview and System Architecture . . . 65
5.3.1 Precise Calling Context . . . 65
5.3.2 Main Components . . . 66
5.3.3 Checkpoints . . . 67
5.3.4 Buffer Reader . . . 68
5.3.5 Anomaly Detection . . . 68
5.4 SPADE . . . 69
5.4.1 Program Behavior Modeling using Precise Call Sites . . . 70
5.4.2 Trusted Execution Environment for Anomaly Detection . 72
5.4.3 Tracing . . . 73
5.5 Implementation . . . 74
5.5.1 Traces for Training . . . 75
5.5.2 Neural Network Model . . . 75
5.6 Evaluation . . . 76
5.6.1 Experimental Setup . . . 76
5.6.2 RQ1: Real-world Attack Detection . . . 78
5.6.3 RQ2: Overhead and Trade-offs . . . 81
5.6.4 GRU vs LSTM . . . 83
5.7 Security Analysis . . . 85
5.7.1 Control-oriented Attacks . . . 85
5.7.2 Stealthy Attacks . . . 85
5.7.3 Modifying Checkpoints . . . 86

5.8 Discussion 87
5.9 Conclusion 87

5.1 INTRODUCTION

To ensure the reliability of IoT applications, it is imperative to assert correct execution of *control programs* running on embedded IoT devices. Otherwise attackers can exploit vulnerabilities in such programs to completely alter their behavior.

Runtime software attacks. Due to the additional cost, energy (battery usage), and development effort involved in securing embedded control programs, security is, alas, still often an afterthought, and not incorporated by design. The consequence is an ongoing stream of exploits. An attack on a water treatment plant in 2016 manipulated the quantity of chemical mix [75], and a similar attack was seen again in 2021 [53]. In late 2019, a vulnerability in a communication protocol was exploited to gain unauthorized access to a hotel's in-room robot assistant that could be used to spy on guests. [58]. In 2017, a protocol vulnerability on Philips hue smart lamps was exploited to control the smart street lights of an entire city [94]. Those are only few examples of evident attacks that are known to the public.

Defense approaches. *Control-oriented attacks* [29] typically modify return addresses to redirect the control flow to an attacker's desired location, thereby violating the control flow graph (CFG) of a program. Control flow integrity [1], remote attestation [3], and program anomaly detection [117] are techniques that have been advocated to detect control-oriented attacks. Control flow integrity and remote attestation, however, cannot capture the non-deterministic runtime characteristics of a program (e.g., execution of `if-else` statements at runtime). Program anomaly detection techniques, on the other hand, use various features to capture the behavior of a program [116, 117]. Any subtle deviation in the runtime behavior compared to the normal execution is flagged as anomaly.

Stealthy attacks. However, an attacker can exploit vulnerabilities to modify decision making variables in order to manipulate control flow indirectly. Such *aberrant path attacks* [101] (cf. *data-oriented attacks* [29]) do not violate the CFG of a program, and hence existing techniques to identify control-oriented attacks are ineffective in detecting them. When a detection technique is in place,

Figure 14: IoT device classes from ARM Cortex family in decreasing order (left to right) of their sizes and processing capabilities, and respective application examples.

the attacker can also attempt to imitate the normal behavior of a program in a *mimicry attack* [111] (not part of data-oriented attacks), to evade detection mechanisms. The above two types of *stealthy attacks* are challenging to detect, as they exhibit behaviors closely resembling normal program behavior. Existing program anomaly detection approaches typically consider either short sequences of function calls or pair-wise function calls to capture 1-level calling context [116], making them sensitive to only local variations, and vulnerable to certain control-oriented attacks (detailed in Section 5.3.1). In addition, to derive the features [29, 116, 119], these solutions rely on Linux-based tools like `strace` and `ltrace`, which cannot be used on the many embedded devices executing freeRTOS, ContikiOS, and mbedOS, or running in bare-metal fashion [54]. A study in 2019 showed that 35% of embedded systems projects did not use any OS, and when an OS was used only 21% used embedded Linux [43]. Moreover, existing anomaly detection solutions assume that the detection scheme, tracing mechanism, and the embedded software are not tampered with, which is unrealistic (cf. stealing machine learning models [109]).

SPADE. In this chapter, we propose a first of its kind secure program anomaly detection for embedded IoT devices (SPADE). SPADE captures the behavior of a program using function calls with precise caller sites. We exploit ETM [78], an on-board debugging component present on an ARM Cortex processor, for extracting function call traces at program execution. Such hardware-based tracing incurs only minimal overhead. We also consider software-based tracing by instrumenting the source code, which incurs higher overhead but can extract many more useful features and does add hardware constraints.

To identify aberrant path attacks, SPADE captures long-term traces. A gated recurrent unit (GRU) [30] neural network is used to model program behavior. GRUs have the ability to remember short-term as well as long-term depen-

dencies. SPADE leverages trusted hardware providing an isolated execution environment to defend the proposed anomaly detector against mimicry attacks. SPADE thus combines trusted hardware and software solutions to create secure and trustworthy IoT systems. Note the emphasis on the combination, as trusted hardware does not solve all problems trivially. Minimizing code deployed on such hardware is essential, just like minimizing the number of context switches between secure and non-secure regions, which SPADE achieves by strategically minimizing the triggers to anomaly detection. SPADE is implemented on tiny embedded devices based on the most resource-constrained processor Cortex-M, making our solution amenable to all classes of IoT devices (see Figure 14 for classes of IoT devices with their computational capabilities).

To the best of our knowledge, SPADE is the first program anomaly detection solution to exhaustively detect stealthy attacks and to use deep learning on the most constrained devices for detection; it also detects all control-flow attacks.

Contributions and roadmap. In summary this chapter has the following contributions:

- a first of its kind *secure* program anomaly detection technique SPADE for detecting control-oriented and stealthy attacks that include aberrant path and mimicry attacks. In particular our design encompasses the following:
  - a GRU-based anomaly detection scheme to detect control-oriented and aberrant path attacks;
  - a novel hardware-based tracing technique for collecting the trace features for anomaly detection using the ETM, thereby introducing minimal overhead;
  - a software-based tracing technique using source code instrumentation that provides flexibility in choosing the granularity of the traces and also the trace features.
- the implementation of SPADE for embedded devices with Cortex-M processor, the most resource-constrained processor available, showing broad applicability of our solution (cf. Figure 14).
- an evaluation of SPADE's anomaly detection accuracy through real-world applications, and its static and runtime overhead. The results show that our GRU-based anomaly detection scheme can detect anomalies with 100% accuracy and only 0-0.5% false positives. SPADE incurs 11.3% average code size overhead with software-based tracing and only 2% with hardware-based. We also show how SPADE can detect various attacks.

The rest of the chapter is organized as follows. Section 5.2 presents the threat model considered, security goals, and challenges for secure program anomaly

detection. Section 5.3 gives an overview of SPADE by presenting its architecture. Section 5.4 details the design of our proposed solution. Section 5.5 presents implementation details. Section 5.6 evaluates SPADE on various case study applications. Section 5.7 discusses how SPADE can identify the attacks described in our threat model. We discuss our design and implementation choices in Section 5.8. Finally, Section 5.9 concludes with possible future research directions.

5.2 PROBLEM STATEMENT

We present our threat model, security goals, and the challenges involved in achieving secure program anomaly detection.

5.2.1 *Threat Model*

We discuss the attacker abilities assumed, followed by attacks using them, and finally illustrate those attacks.

Attacker abilities. We assume that an attacker can exploit vulnerabilities present in an embedded application to launch software exploits, thereby gaining access to devices. We also assume that there are no internal threats in the development setting (e.g., during training). These basic assumptions are common to program anomaly detection works [29, 119, 3].

Attacks. Control-oriented attacks modify return addresses to redirect the control flow of a program to a location in existing or newly injected code, introducing illegal control flows. A stealthy attack, where an attacker can exploit a vulnerability, e.g., to change an authentication variable to access critical tasks, does not introduce any illegal control flow. Aberrant path attacks, first coined by Shu et al. [101], group several attacks that indirectly affect the control flow of a program without affecting the integrity of control paths. Cheng et al. consider these as data-oriented attacks [29]. However, a solution for detecting the above attacks is futile without being resilient to mimicry attacks [115] (not part of data-oriented attacks). Aberrant path and mimicry attacks are *stealthy* in nature, making their detection challenging.

Attacks illustration. In order to illustrate how attackers can execute control-oriented, aberrant path, and mimicry attacks we use the open syringe application [3] as shown in Listing 5.1. The application controls the bolus of a syringe by the number of units specified over a serial port by an authenticated user.

```
char cmd[20] = {0};
int cmd_ready = 0;

void bolus(int direction) { ... }

void process_cmd() {
  if(cmd[0] == '+') bolus(PUSH);
  else if(cmd[0] == '-') bolus(PULL);
}

void config() {
  char buffer[20] = {0};
  int authenticated = 0;

  while(serial_available()) {
    char in_char = (char)serial_read();
    if(in_char == '\n') {
      if(auth_password(buffer)) {
        authenticated = 1;
        serial_write("...");
      }
      break;
    } else { buffer += in_char; }
  }
  if(authenticated) {
    cmd = read_cmd();
    cmd_ready = 1;
    ... /*any other critical tasks*/
  }
}

void loop_pass() {
  config();
  if(cmd_ready) process_cmd();
}
```

Listing 5.1: Code snippet of *open syringe* application [3] (slightly modified to demonstrate all the attacks). The authentication is performed similar to the SSH protocol.

Table 3: Sample of traces under normal conditions, during a control-oriented attack, aberrant path attack, and mimicry attack.

Normal - auth	Normal - fail	Control-oriented attack	Aberrant path attack	Mimicry attack
loop_pass->config	loop_pass->config	loop_pass->config	loop_pass->config	loop_pass->config
config->serial_avail	config->serial_avail	config->serial_avail	config->serial_avail	config->serial_avail
config->serial_read	config->serial_read	config->serial_read	config->serial_read	config->serial_read
lines 2 - 3 repeated	*lines 2 - 3 repeated*	*lines 2 - 3 repeated*	*lines 2 - 3 repeated*	*lines 2 - 3 repeated*
config->auth_password	config->auth_password	config->auth_password	config->auth_password	config->auth_password
config->serial_write	loop_pass->config	config->bolus		config->serial_write
config->read_cmd	...	...	config->read_cmd	config->read_cmd
loop_pass->process_cmd	...	...	loop_pass->process_cmd	loop_pass->process_cmd
process_cmd->bolus	...	...	process_cmd->bolus	process_cmd->bolus
...	...	...	...	...

Table 3 shows sequences of collected traces during normal execution of both successful and failed authentication for approaches that consider two-tuple information (also called 1-level calling context); a function call with its caller function is represented as `caller->callee`. A control-oriented attack exploits a buffer overflow vulnerability (Line 12–Line 17) to overwrite the return address of `config` and jump to `bolus`, executing critical commands that impact the actuators. Such an attack introduces an illegal control path (`config->bolus`) during runtime. A (non-control) data-oriented attack modifies the decision-making variable `authenticated` to execute critical functions and control the syringe without successful authentication (Line 26–Line 28). Critically, as shown in Table 3, aberrant path attacks do not introduce any illegal control path.

Securing the tracing mechanism and the detection process is crucial; without this, a proposed solution becomes itself vulnerable to mimicry attacks and an attacker can easily evade aberrant path detection. Table 3 shows how the tracing mechanism can be tampered with to include a missing sequence (`config->serial_write`) and mimic the normal behavior of a program. Xu et al. [115] discuss various ways of carrying out a mimicry attack.

5.2.2 *Security Goals*

Concretely, considering the above threat model, the three main security guarantees provided by this work are as follows:

G1 Detection of control-oriented attacks that modify the return address of a vulnerable function.

G2 Detection of aberrant-path attacks (cf. data-oriented attacks) tampering with predicate-use variables to execute incompatible branches.

G3 Protection against mimicry attacks attempting to evade detection by tampering with tracing mechanisms to introduce missing sequences.

While **G1** and **G2** are achieved using our novel anomaly detection technique at runtime, **G3**. leverages trusted hardware.

5.2.3 *Trusted Hardware (and) Challenges*

To isolate user application code and SPADE code we leverage ARM TrustZone for Cortex-M [11]. Due to several attacks on IoT devices in recent times, trusted hardware has become a norm for carrying out critical operations. Most devices in the Cortex-A category of processors have inbuilt trust anchors. In 2018, ARM released TrustZone for the Cortex-M family [89].

However, there are several non-trivial challenges that need to be addressed when using trusted hardware, and the challenges only get more stringent for tiny devices with a Cortex-M processor. Also, additional effort is required in implementing detection algorithms, such as the one based on the neural networks we introduce shortly, for Cortex-M compared to Cortex-A, not only due to tighter resource constraints, but also due to non-availability of several libraries.

Precisely, we identify the following challenges implementing a secure program anomaly detection for tiny IoT devices:

C1 The detection process, user application code, runtime tracing mechanisms, and the generated traces *have to be secured*. Attackers otherwise can simply modify any of these to circumvent the detection.

C2 Straightforwardly placing all the code in the secure region of trusted hardware is not an option. The secure code base *has to be kept minimal* (i) due to limited secure memory [84, 45], and (ii) to ensure no vulnerabilities are introduced unknowingly, which otherwise can provide illegal access to the secure region.

C3 Switching between secure and non-secure regions of trusted hardware *has to be minimal* as each switch incurs overhead.

Besides the above challenges we identify the following challenges in designing an actual detection scheme:

C4 The anomaly detection scheme must be able to detect all the attacks described in Section 5.2.1 *with acceptable false positive and false negative rates*.

```
1  void config() {
2    ...
3    while(serial_available()) { ... }
4
5    if(authenticated) {
6      cmd = read_cmd();
7      if(cmd[0] == '+') bolus(PUSH);
8      else if(cmd[0] == '-') bolus(PULL);
9      ...
10   }
11 }
```

Listing 5.2: Modified *open syringe* application to demonstrate the need for tracing precise call location.

C5 In order to implement an anomaly detection scheme for tiny embedded devices, the anomaly detector and any model it relies on have to be *efficient in terms of space and runtime latency*.

As foreshadowed, in this paper we consider neural networks to accurately detect anomalies. This makes challenges C4 and C5 particularly stringent for embedded devices.

5.3 OVERVIEW AND SYSTEM ARCHITECTURE

In this section, we present the system architecture of SPADE, and tracing features.

5.3.1 *Precise Calling Context*

We extract various features from the running application in order to detect attacks at runtime. SPADE captures short- and long-term patterns with *precise call sites* as a context for anomaly detection (C4). We further explain the attacks that can go undetected when just calling functions are considered as context [29, 119, 116] and how SPADE can detect such an attack using precise call location.

For the program in Listing 5.1, a 1-level calling context (as considered in [116]) can detect a control-oriented attack as the attack introduces an illegal control path `config->bolus` (`process_cmd->bolus` is the legal control path). Consider a developer writing the same application differently by eliminating the `process_cmd` function and including its features within `config` as shown in Listing 5.2. The 1-level calling context trace for the normal authenticated case

and during control-oriented attack will be the same (`config->bolus`), and hence the control-oriented attack goes undetected. Capturing precise calling location can however identify such an attack. In the normal authenticated case, the `bolus` function will be invoked from Line 7 or Line 8 of Listing 5.2, whereas during a control-oriented attack the same function will be invoked from Line 11 of (return address will be changed to point to `bolus` function).

We consider two different types of tracing, *hardware*(-based) tracing and *software*(-based) tracing, which exhibit a trade-off between (a) performance vs (b) flexibility and availability: Hardware tracing is performed by leveraging a built-in circular embedded trace buffer (ETB) available on several ARM embedded devices, which provides instruction tracing with minimal overhead. Software tracing is performed by instrumenting the source code to trace various features, providing flexibility in type and granularity of features collected.

More precisely, SPADE uses the following tracing features extracted for anomaly detection from a given control program.

Definition 1 (Context-sensitive function call) *A context-sensitive function call* $f : l \to f'$ *indicates that a function* f' *is invoked from location* l *which is within the scope of function* f*. Here* l *is a precise call site which is captured as the context.*

Definition 2 (Call sequence) *A sequence of context-sensitive calls is denoted* $X = (x_1, x_2, ..., x_n)$ *where* $x_i = f_i : l_i \to f'_i$*.*

5.3.2 *Main Components*

SPADE consists of several software modules running on a target IoT device. Figure 15 shows SPADE's architecture. ARM TrustZone isolates processes, peripherals, and memory regions into secure (green; right) and non-secure regions (red; left). Importantly, SPADE runs in the secure region (**C1**) and the underlying operating system (if any) and user applications run in the non-secure region (**C2**).

SPADE further contains a *buffer reader* that reads required partial traces, from the ETB or from a buffer implemented in software, when indicated by the *anomaly detector* module. The latter module runs a neural network inference engine that checks the runtime function traces against a trained neural network model. The anomaly detector is triggered by *checkpoints* in the user application. A *watchdog* keeps track of checkpoint triggers, and invokes the anomaly detector if the module suspects there has been a suppression of triggers.

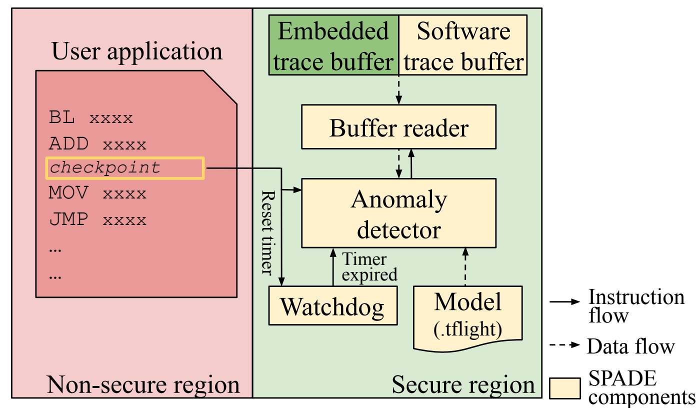


Figure 15: System architecture overview.

5.3.3 *Checkpoints*

To facilitate on-device anomaly detection, the detection module has to be invoked several times during program execution. We achieve this by introducing checkpoints after the execution of critical tasks that affect the operation of the embedded software (e.g., actuators, serial reads, or network packet reads). However, designing a generic technique to identify such logical checkpoints is not plausible because there are several sensors, actuators, and a large set of APIs from different libraries/vendors to access the sensors and actuators. Therefore, we introduce a simple intuitive API,

```
int ADCheckpoint()
```

that a programmer can leverage to declare a checkpoint; multiple checkpoints at any granularity can be introduced. Providing APIs to collect information from a program is a standard approach followed in automotive open system architecture (AUTOSAR) [14], and related literature (e.g., [104]).

A checkpoint invokes the anomaly detector, which reads the relevant traces to check for an anomaly. However, an attacker may modify the application to remove or alter the location of a checkpoint to evade detection. To protect against such checkpoint corruptions, the execution of a checkpoint also notifies the watchdog that monitors its executions. The processing of a watchdog is discussed shortly. Finally, the detection process itself introduces overhead (reading traces, runtime inference, anomaly detection) that can affect the real-time performance of user application running in the non-secure region. The introduction of checkpoints for the program running in the non-secure region alleviates overheads by minimizing the context switches between secure and

non-secure regions for anomaly detection (C3). In summary, checkpoints are used to trigger anomaly detection for the three following reasons:

1. Processing large traces at a time introduces noticeable lag in user application responsiveness because the anomaly detector can hold the processor for a longer duration.
2. The hardware trace buffer leveraged or the software buffer used for anomaly detection is limited in size, hence waiting till the end of a program (or single execution) may cause buffer overrun, leading to missed trace samples.
3. The time spent in anomaly detection can be minimized by tracing only critical tasks instead of the entire program.

5.3.4 *Buffer Reader*

When hardware tracing is enabled, the buffer reader module implements the protocol of ARM's CoreSight library [78] to access the trace information from the ETB. The ETB stores the trace information in a predefined format (see Section 5.4.3) that cannot be directly used for anomaly detection. The buffer reader module processes the trace information to extract the required trace features. If software tracing is enabled, function traces of the instrumented program running in the non-secure region are collected and stored in the secure region to prevent any unauthorized access. Hence, when an instrumented function is executed, there is a context switch into the secure region to write the trace into the software buffer. The buffer reader waits for a trigger and reads N trace entries from the current pointer in the software or hardware buffer (latest N).

5.3.5 *Anomaly Detection*

Modeling and inference. In order to detect control-oriented and aberrant path attacks on the device, we build a model of program behavior using a GRU that captures short- and long-term dependencies. To train the GRU model before deployment, we extract the program traces from the embedded device to an external general-purpose system either by hardware or software tracing mechanisms. For hardware tracing, the execution traces are extracted from the on-chip debug and trace tools of the ARM CoreSight. With software tracing, the instrumented binary prints the execution traces through a serial port. The trained model is stored in the secure region, used by the anomaly detector at runtime for detecting program anomalies. When the anomaly detector is

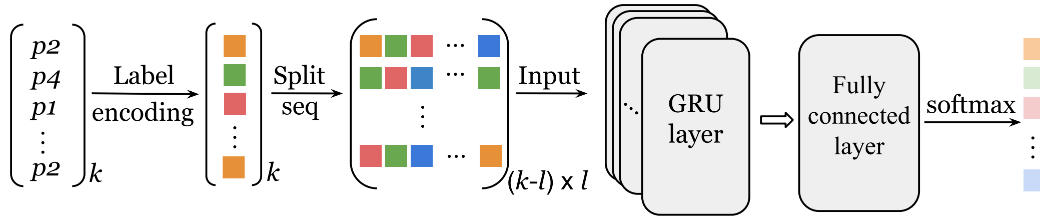


Figure 16: GRU network architecture used for program anomaly detection. The intensity of the color in the output vector indicates the probability of the occurrence.

Table 4: Different encoding techniques.

Encoding technique	Prediction accuracy (%)	Model size (KB)	Single inference (ms)
Label	95.2	15.05	3.1
One-hot	95.3	25	4.7
Embeddings	97.3	34.27	6.3

triggered, the module reads latest N traces from the buffer reader, and encodes them to the input structure needed by the GRU model that predicts the $(N+1)^{th}$ trace entry using the trained model. The predicted output is compared to a pre-derived threshold to check for an anomaly.

Watchdog. As the checkpoints are non-secure, an attacker could modify the user application or execute a control-oriented attack to bypass anomaly detector. To overcome this, we introduce a watchdog, a timer in the *secure* region which is reset each time the anomaly detector is triggered. On expiration, it invokes the anomaly detector unconditionally, thereby preventing an attacker to bypass detection. The timer value is application-dependent and chosen as the largest value between consecutive checkpoints.

5.4 SPADE

This section details our program anomaly detection scheme, including how it is secured, and our tracing mechanisms.

5.4.1 *Program Behavior Modeling using Precise Call Sites*

We capture the behavior of a program and design an anomaly detection scheme based on a language learning model. Our scheme can jointly learn the semantics of individual function calls and their interactions appearing in the trace sequences, whilst existing solutions are limited to coarse-grained features and pairwise function call patterns. As already highlighted in Section 5.3.1, the latter kind of patterns is insufficient to detect control-oriented attacks.

Our anomaly detection approach directly addresses these limitations by capturing short- and long-term patterns with precise function invocation locations. As shown in Table 3, though each entry in an aberrant path attack reflects a legal control path, the sequence as a whole is illegal. In the example, `config->serial_write` is skipped and the `read_cmd` function is directly invoked, which differentiates between malign and benign sequences. In order to detect such an attack it is important to preserve the ordering of function call occurrences. We use recurrent neural networks (RNNs) to capture short- and long-term patterns and also to achieve order sensitivity. Before further detailing our modeling and anomaly detection techniques, we provide a brief background on RNNs.

Background on recurrent neural networks (RNNs). RNNs efficiently model sequential data. The value of a hidden state is determined using the current input and the value of a hidden state at a previous time step; hence, subsequent outputs contain information of previous states. Vanilla RNNs suffer from vanishing gradient problems when modeling long inputs. GRUs, just like long short term memories (LSTMs), solve this issue by using various gates in each memory cell to decide what to remember and forget at that cell.

Language model based anomaly detection. GRUs have a simpler structure and can reduce the number of parameters needed to train. For this reason, in SPADE, we choose GRUs as primary RNN units for our feature prediction model. In Section 5.6.4 we compare LSTMs vs GRUs for anomaly detection in terms of accuracy and overhead.

Figure 16 illustrates the architecture of our GRU model (**C5**). The model estimates the probability distribution of the $(N+1)^{th}$ context-sensitive call given the previous N calls. For a given user application, let S be the set of possible context-sensitive function calls of length k. Label encoding is performed for every element in S. We choose label encoding as it results in smaller model size and runtime inference compared to other alternative encoding techniques as shown in Tab. 4.

Let $X = (x_1, x_2, \ldots, x_l)$ denote the input call sequence, where $x_i \in S$ and l is the input sequence length. The set of sequences is used at the input layer, and fed to the model. The GRU unit has an internal state in the hidden layer and a state update is performed recurrently at each time step t as follows:

$$z_t = \sigma(W_z X_t + U_z h_{t-1}) \qquad r_t = \sigma(W_r X_t + U_r h_{t-1})$$
$$\tilde{h}_t = \phi(W_h X_t + U_h(r_t \odot h_{t-1})) \qquad h_t = z_t h_{t-1} + (1 - z_t)\tilde{h}_t$$

where σ is a sigmoid function, ϕ is the rectified linear unit, and W, U are the learnable parameters. A softmax layer is used as the output to estimate normalized probabilities of next calls in the sequence.

The sequence probability is estimated using a chain rule as:

$$P(X) = \prod_{i=1}^{l} P(x_i | x_{1:i-1}) \tag{4}$$

We train this GRU-based language model over normal call sequence data; the training sequences are generated by considering sliding windows of predefined lengths over the execution traces. The model is trained using a probabilistic loss function such as cross-entropy, to obtain the probability distribution of the estimation instead of a single value. The cross-entropy loss is given as:

$$loss = -\sum_{i=1}^{l} \sum_{j=1}^{k} x_{i,j} \times \log(P(\hat{x}_{i,j})) \tag{5}$$

where $\hat{x}_{i,j}$ is the predicted output for the given input $x_{i,j}$. The training is performed to minimize $loss$.

During inference, the GRU model gathers trace sequences of a predefined length N (same as the length of window used during training) from the buffer reader as input, and encodes them using label encoding. We predict the next call – i.e., the $(N+1)^{th}$ call – in this sequence using the trained language model as $P(x_{N+1}|x_{1:N})$. This yields a vector of normalized probabilities $(P_1, \ldots, P_k)$ as predictions for the next call, where P_i denotes the probability of call i. We define a threshold-based classifier as follows:

$$C(x_{1:N}; \theta) = \begin{cases} anomaly & \text{if} P_i < \theta, \forall i \in \{1, \ldots, k\} \\ normal & \text{otherwise} \end{cases} \tag{6}$$

By changing the threshold value θ, we construct an operating characteristic curve to evaluate the performance of our model and choose the threshold that yields the best accuracy (see Section 5.6.2).

5.4.2 *Trusted Execution Environment for Anomaly Detection*

In this paper we use the commercially available ARM TrustZone as trusted execution environment (our architecture can also use others) to secure the anomaly detection process. We first provide a brief background on the architecture of ARM TrustZone for Cortex-M devices, followed by how SPADE leverages the features of TrustZone to design a secure anomaly detection technique (**C1**).

Background on ARM TrustZone for Cortex-M. TrustZone allows developers to design a secure application using trusted execution environment. The secure region code can access all the peripherals, code, and data residing in both the secure and non-secure regions. However, the non-secure region can access only itself. Switching between the two regions is done in hardware, thereby keeping the latency overhead of context switching to a minimum. Since TrustZone for Cortex-M based devices differs from that for Cortex-A devices, in the following we use TrustZone-M to specifically refer to the TrustZone component of a Cortex-M device.

Secure anomaly detection. Our proposed program anomaly detection scheme (i) captures runtime traces and (ii) compares them against a GRU model to detect anomalies. However, without securing the tracing process and the model, an attacker can tamper with either (i) or (ii) to circumvent the objective of the underlying detection scheme, making it futile. The hardware isolation of TrustZone-M is used in SPADE for achieving the following:

SECURE BOOT: The secure boot feature of a TrustZone-M device is executed each time the device is reset. It verifies the integrity and authenticity of the user application running in the non-secure region. By leveraging secure boot, SPADE ensures the non-secure region code is not modified, which makes it impossible for an attacker to execute a mimicry attack and subvert our detection scheme.

SECURE TRACING: SPADE reads the instruction traces from ETB for runtime behavior analysis. We map this hardware feature for tracing the program execution with minimal overhead to the secure region of the embedded device (Figure 17 shows ETM and ETB mapped to secure region), thereby ensuring that the attacker cannot tamper with the tracing mechanism or the traces. When software tracing is used, SPADE uses software hooks at locations where trace features are collected. A software hook induces a jump to the secure region, where the features are extracted and stored in a software trace buffer also residing in the secure region (see Figure 15).

Table 5: Software and hardware traces for the code in Listing 5.1.

Software traces	Hardware traces
`loop_pass:33->config`	`config:15`
`config:15->serial_avail`	`config:16`
`config:16->serial_read`	`config:18`
`config:18->auth_password`	`config:20`
`config:20->serial_write`	`config:26`
`config:26->read_cmd`	`loop_pass:33`
`loop_pass:34->process_cmd`	`process_cmd:7`
`process_cmd:7->bolus`	`loop_pass:34`

5.4.3 *Tracing*

The feature collection process impacts the accuracy of the neural network model used for anomaly detection. We extract various features from the application for training the model and successfully detecting an attack at runtime. Collection of training data and model training takes place under supervision and before deployment, and hence overhead there is not a concern. However, the overhead to collect the trace data for inference at runtime has to be minimal to ensure minimal effect on the application's performance. SPADE supports two different ways of tracing: first a hardware approach that has negligible overhead; second a software approach that provides flexibility in tracing granularity and features captured, and can also be used in absence of the required hardware. Before detailing the tracing techniques, we provide a brief background on the ARM CoreSight architecture used in hardware tracing.

Background on CoreSight architecture. Figure 17 shows a simplified version of the CoreSight architecture [78] retaining only the components relevant to SPADE. The ETM is an optional debugging component which generates *trace packets*.

To extract traces from ETM, an external hardware debugging device is required and the traces have to be captured at the speed of the processor. CoreSight also provides an optional on-chip buffer, the embedded trace buffer (ETB), which stores the trace packets generated by ETM and can be read at a slower rate. The trace data from ETB can be accessed from the software running

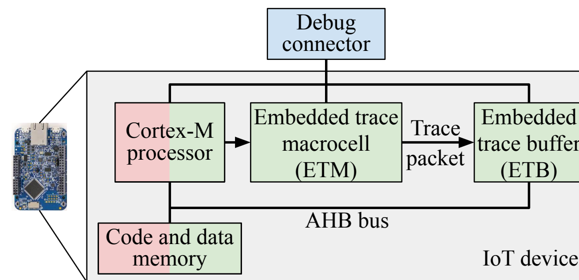


Figure 17: ARM CoreSight debug architecture (simplified from [78]) showing ETM and ETB interaction with the core processor.

on the processor through the advanced high-performance bus (AHB) bus. The ETB's size depends on the manufacturer; it is typically limited to 4-8 KB.

Hardware tracing. We propose a novel mechanism of tracing by leveraging the features of the underlying hardware with which the tracing overhead is negligible. We use ARM's CoreSight on-chip debug architecture which includes several components, of which we access the instruction tracing feature of ETM. For anomaly detection, we consider the branch address packets generated by the ETM. A sample trace packet extracted from the ETB is as follows

```
BranchAddressPacket Bytes=89d600, addr=0x2b08
```

where the `addr` indicates the return location of the branch. Table 5 shows the sample traces collected for the program shown in Listing 5.1.

Software tracing. A pure software-based tracing does not make any assumptions on the hardware. Here we instead instrument the source code to collect function entries and their precise call sites (cf. Section 5.3.1). The functions in an embedded program along with their call sites are considered to model the behavior of a program. By using this combined information we can detect control-oriented and aberrant path attacks. For the program listing shown in Listing 5.1, `config` is invoked from `loop_pass` at Line 33. Assuming – for ease of presentation only – that the line numbers represent actual addresses in the application binary, `loop_pass:33->config` represents a 1-level calling context. Table 5 shows sample traces collected using software tracing.

5.5 IMPLEMENTATION

We implemented the various components of SWARNA (see Figure 15) in the C programming language. The code introduced by SWARNA for anomaly

detection in the secure region is only 1.2 KB, not including the neural network model. Additional care is taken to ensure that there are no vulnerabilities in the secure region, and that the code in the secure region does not interact with the outside world (C2). This section highlights several implementation issues.

5.5.1 *Traces for Training*

In order to collect the training data with hardware tracing enabled, traces are read from ETB while having the embedded device connected to the desktop. A debugger is used to download the ETB data onto the desktop computer. For software tracing, the application is instrumented to capture function entries and their calling location. For instrumentation, an application is compiled with the GNU option `finstrument-function`, which invokes a pre-defined function during the entry and exit of every function in the application. We limit the traced functions to the application level, critical system functions, and actuator code.

A typical embedded control program runs in an infinite loop after a set of initialization functions. Due to this nature, initialization functions appear only once in the collected trace. Hence, we remove such entries and consider only function invocations within the infinite loop, greatly reducing the vocabulary size: for the considered applications, we see a reduction in vocabulary size by 10-82%. This also reduces the model size and runtime inference. We eliminate initialization traces for model training as an optimization technique. An attack that jumps to the initialization function during program runtime will still be detected. However, an attack during initialization can lead to higher false positives, but the attack will not go undetected. The anomaly detector will raise an alarm because it encounters a path not seen during the training process. The additional optimization can always be disabled, if required.

5.5.2 *Neural Network Model*

We used the Keras APIs [69] to implement the GRU network and TensorFlow to generate a trained model. Several experiments were conducted to choose hyperparameters for training the GRU model. We use STM32Cube.AI [103] for compressing the TensorFlow model to a lightweight model suitable for micro-controllers and to implement a neural network inference engine on the embedded device.

We implemented a language model based anomaly detection

(see Section 5.4.1) using Python. The model is trained using benign traces. We implemented a GRU neural network architecture as outlined in Figure 16. Table 6 shows the chosen values for each of the hyperparameters for the given network architecture. The detection scheme is first evaluated on a desktop computer against the test data to derive optimal thresholds for identifying anomalies. The thresholds derived are later used by the anomaly detector module on the embedded device to detect anomalies at runtime (see Equation 6).

Table 6: Hyperparameters for training neural network.

Hyperparameter	**Value**
Hidden layers	1
Neural nodes	32
Epochs	150
Batch size	16
Sequence length	10
Dropout rate	0.2
Learning rate	0.001

5.6 EVALUATION

We extensively evaluate the performance and detection accuracy of SPADE. In particular, we address the following research questions:

RQ1: What is the effectiveness of SPADE in detecting real-world attack variants?

- **(a)** What is the detection accuracy for each attack variant and how sensitive is the detection accuracy to the classification threshold?
- **(b)** How does SPADE compare to the state of the art?

RQ2: What are the impacts of software and hardware tracing on SPADE's performance?

- **(a)** What are the static and runtime overheads due to tracing and modeling?
- **(b)** How does software and hardware tracing affect attack detection?

5.6.1 *Experimental Setup*

Case studies of embedded applications. In order to evaluate the performance of SPADE, we consider five real-world embedded control programs. A control program will sense the environment, perform lightweight processing, and, depending on the results, act on the actuators present on the device.

FALL DETECTION: A simple threshold-based fall detection [17] is implemented using the on-board accelerometer sensor. The application allows an authenticated user to configure threshold value on the press of a button. The fall

detection algorithm monitors the 3-axis accelerometer values, and raises alarm if the computed fall value is greater than a configured threshold.

OPEN SYRINGE: The open syringe application has been widely used in the literature to showcase various attacks [29, 3, 104]. The application takes user commands to control the bolus of a fluid-filled syringe. The application accepts commands to set the quantity of liquid to be dispensed and commands to push/pull the syringe by the set quantity value. We use the serial port for input and output.

CRYPTOGRAPHY: IoT applications that ensure data privacy perform some form of encryption or decryption on the embedded device [79]. We implement an application that accepts user text over the serial port and an input command. Based on the command, it encrypts or decrypts using the AES-CBC algorithm [50] and prints the result.

UDP DATA COLLECTION: Data collection is widely used in several IoT applications like smart grid, city, health, and many more [79]. We implement a periodic UDP-based data collection application that runs a low-power IP stack (LwIP). The embedded device sends the temperature value once every 2 s to the UDP server over the network. The embedded application accepts commands over the network to stop sending the data or change the sending rate.

LIGHT CONTROLLER: The on-board LED is controlled using commands to turn-on/off the LED or to change its color and brightness. The application is similar to the open-source light controller [77].

Choice of hardware. The experiments are evaluated on a tiny embedded device with a Cortex-M processor. Existing works on anomaly detection for IoT consider higher-end devices like Raspberry Pi which has Cortex-A processor (see Figure 14). SPADE can run on Cortex-M and thus on higher-end devices (whilst the opposite is not generally true) making it applicable to a large class of IoT devices.

We evaluate SPADE on the latest hardware by STMicroelectronics (STM) – STM32L56 with ARM Cortex-M33 core, TrustZone-M, 512 KB of flash memory, 256 KB of SRAM, and clock cycle of 110 MHz. Though the underlying Cortex-M33 core supports the ETB used for hardware tracing, it is not enabled on STM32L56. Cortex-M33 devices are available only from 2019 [96], and none of these devices currently has ETB enabled. Therefore to evaluate SPADE with hardware tracing, we extract ETB data from NXP's freedom K64F with Cortex-M4 processor. Cortex-M33 and Cortex-M4 differ mostly in the TrustZone

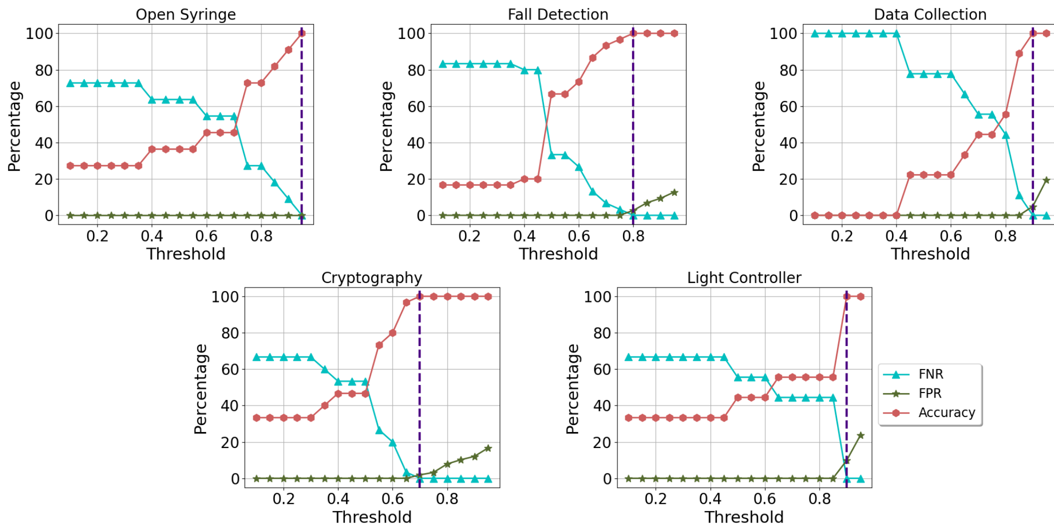


Figure 18: The graphs indicate the achieved false negative rate (FNR), false positive rate (FPR), and anomaly detection accuracy for various applications. The vertical line (dashed blue) in each graph shows the threshold values for the application.

availability [36].[1] The extracted ETB traces are stored on the TrustZone-M enabled STM32L56 as a simple buffer just for evaluation purposes without loss of validity.

5.6.2 *RQ1: Real-world Attack Detection*

Data preparation. For data collection, the embedded applications are run for 1h, during which the user interacts with the applications at random times. We split the collected data into (a) training data (65%) used for training the neural network model and (b) test data (35%) used for deriving the threshold θ in Equation 6 for attack detection. However, importantly, runtime attack detection and runtime overhead are evaluated on the IoT device.

Attack variants. The case study applications have buffer overflow vulnerabilities which we exploit and carry various attacks at runtime. We repeat the below experiments for 20 times and plot the results.

1 A Cortex-M33 based device with ETB enabled is very likely to become available in the near future due to the fact that the processor itself can support ETB and also most of these devices already have CoreSight architecture's ETM implemented (cf. Figure 17).

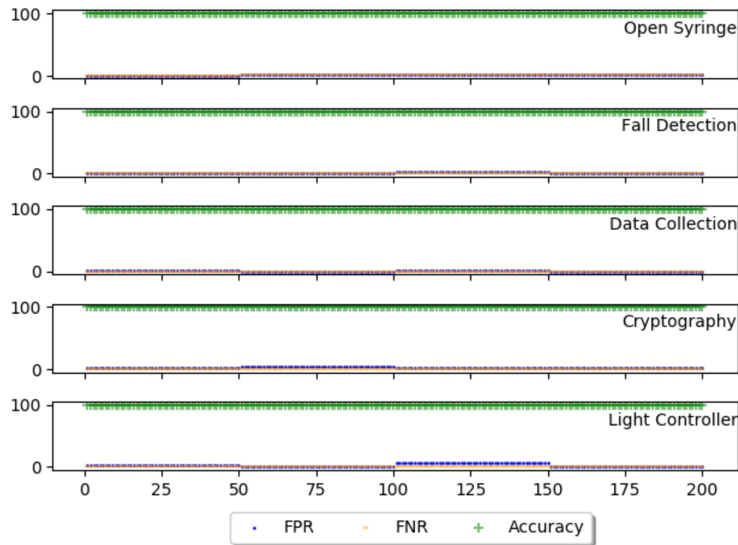


Figure 19: Detection accuracy of aberrant-path attacks for various applications

REDIRECT CONTROL FLOW: We modify the return address and jump to a critical function in the program. At runtime, for each application we carry out such a control-oriented attack 10 times and redirect the control flow to a different location each time. SPADE can detect all the attacks (refer to Section 4.8 for a detailed analysis).

OVERWRITE DECISION-MAKING VARIABLE: At runtime, we carry out 10 different aberrant-path attacks on each of the application to override the decision made (e.g., execute an `else` block instead of `if` block) to illegally execute critical block of code.

Model Prediction Accuracy. Given trace sequences of length N, our GRU model predicts the $(N+1)^{th}$ call. Table 7 shows the prediction accuracy for the various case study embedded applications. From the table and Figure 19, we observe that models with lower prediction accuracy have higher FPR. However, no attacks go undetected.

Table 7: Model prediction accuracy

Application	Prediction Accuracy (%)
Open syringe	94.5
Fall detection	99.3
Data collection	97.4
Cryptography	96.2
Light controller	97.5

Detection accuracy. To detect an anomaly, a GRU is executed against a predefined length of runtime traces. If all the probabilities in the encoded output vector are below a threshold,

then the anomaly detector marks it as an anomaly (see Equation 6). Detection accuracy is the number of these anomalies SPADE detects successfully and halts execution. Figure 19 shows the false negative rate (FNR), false positive rate (FPR), and detection accuracy for aberrant-path attack variants. SPADE achieves 100% accuracy with 0-0.5% false positives and 0% false negatives. For control-oriented attacks, our system achieves 100% accuracy with 0 FPR and FNR. Please refer to Table 5.6.2 for prediction accuracy of the GRU model.

Threshold and its sensitivity. The threshold selection is performed based on a subset of the data collected during an application's *normal* runtime; we identify the threshold value θ using the test data. For a given dataset, let S be the set of possible context-sensitive function calls of length k. An input call sequence $(x_1, x_2, ..., x_N)$ is fed to the GRU model that yields a vector of normalized probabilities $P = (P_1, P_2, ..., P_i, ..., P_k)$ as predictions for the next call, where Pi denotes the probability of call i. Denoting the ground truth for the next call as $\hat{x}_j \in S$ for $j \leqslant k$, for a given P the threshold is evaluated as $\theta_j = P_j$, where P_j is the probability of $\hat{x}_j$ in the output vector P. Finally, we derive the anomaly detection threshold as:

$$\theta = \min(\theta_1, \theta_2, ...\theta_j, ..., \theta_{l-N}), \tag{7}$$

where l is the length of test data S.

Figure 18 shows the sensitivity of FNR, FPR and detection accuracy against the threshold values. The vertical line shows the chosen threshold value for the application.

SPADE vs existing techniques. We compare SPADE against the basic n-gram anomaly detection technique [47] and LAD [101] that is closest to our work. The traces during an aberrant path attack are collected for the fall detection application. The n-gram model stores traces of length n in a database during training. At runtime, if the n-length trace collected does not match any entry in the database an anomaly is flagged. To detect an aberrant path attack using n-grams, the minimum value of n required is 23, for which the observed FPR is 0.6% and 0% FNR. We observed that for a higher n value the FPR only increases. However, SPADE detects this attack with 0.002% FPR and 0% FNR. LAD captures long-term behavior using co-occurrence and frequency analysis. However, such an approach fails to capture the order of function calls. Therefore, LAD fails to detect a subtle form of aberrant path attack where the attack trace resembles a normal trace with a few missing sequences (cf. Table 3). For a variant of aberrant path attack which introduces illegal trace sequences, LAD is shown to achieve over 90% detection accuracy with 0.01% FPR [101].

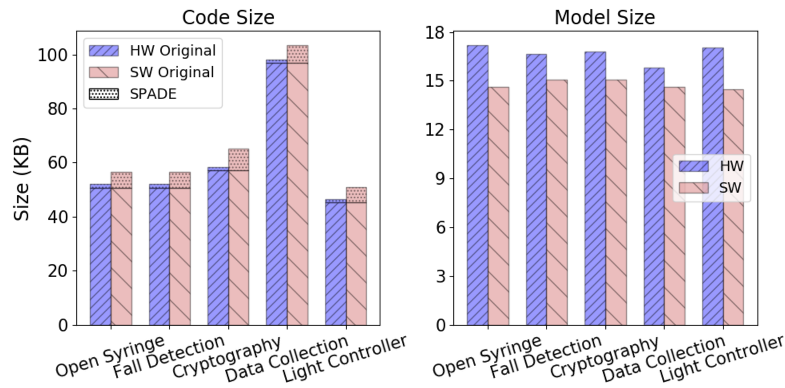


Figure 20: Static overhead that shows the application code size with SPADE overhead (tracing + anomaly detection), and GRU model size for hardware-based (HW) and software-based (SW) traces.

While LAD raises slightly fewer false alarms than SPADE, LAD fails to detect all the anomalies, whilst SPADE never misses any anomaly.

5.6.3 *RQ2: Overhead and Trade-offs*

Static overhead. Figure 20 shows the application code size with SPADE overhead, and GRU model size when hardware and software tracing are enabled respectively (cf. Section 5.4.3). SPADE code size overhead includes tracing functionalities in the non-secure region, and the anomaly detection (1.2 KB of code) in the secure region. SPADE incurs 11.3% average code size overhead with software tracing and only 2% with hardware tracing. We do not see a significant overhead with hardware tracing as only a couple of lines of code are introduced to read the data from ETB. However, we see that the GRU model has a smaller size with software tracing compared to hardware tracing. With software tracing, we can have the flexibility of choosing only the required files/functions to be traced. Figure 21 shows how the model size increases when more modules are considered for tracing. We trace user code and functions related to sensors and actuators (in our case LEDs, UART, LwIP) running in the non-secure region. However, when hardware tracing is enabled, the function traces for all the software modules are captured by the hardware. Hence, the number of unique samples in the traces collected is higher for hardware tracing compared to software tracing, thereby increasing the model size. However, hardware tracing has less overhead, when all the modules are considered for software tracing (see Figure 21).

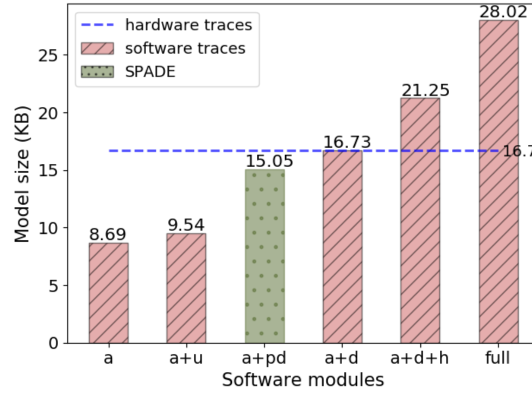


Figure 21: The graph shows the GRU model size when various software modules of the fall detection application are considered for tracing. 'a' - app, 'u' - utilities, 'd': drivers, 'pd': partial code from drivers, 'h' - hardware.

Runtime overhead. We evaluate the runtime overhead of SPADE on the embedded device. Table 8 shows the values of runtime performance for various applications when hardware or software tracing are enabled. We use a benchmark application that is part of STM32Cube.AI [103] to measure a single runtime GRU inference which is averaged over 16 runs. The overall execution time includes time to switch the context between secure and non-secure regions, executing the GRU inference engine, and detecting whether there is an anomaly. The context switching time is hardware-dependent and is reported as 3-4 cycles per switch [64]. A context switch is performed when a checkpoint for anomaly detection is executed, and also for trace collection with software tracing. The latency involved in a single GRU inference with hardware tracing is higher than software tracing due to increased number of unique functions traced with hardware tracing However, the overall execution time with hardware tracing is smaller than with software tracing. In order to secure the software trace buffer and the tracing mechanism, there is a context switch for secure tracing (see Section 5.4.2) which introduces a noticeable runtime overhead.

Trade-offs in Tracing. Tracing with ETB incurs negligible overhead compared to software tracing. Though the model size with hardware tracing is slightly larger (approx. 2 KB) than software tracing (Figure 20), the overall runtime overhead of software tracing is 20-80% higher than hardware (Table 8) tracing for various applications. Hence, when the chosen IoT embedded device supports ETB, hardware tracing can be chosen over software. Hardware tracing is also beneficial when third-party libraries are used for which source code is not available. Yet, with hardware tracing, only return information can be captured on

Table 8: Runtime overhead of SPADE using software-based (SW) and hardware-based (HW) traces.

		# of context switches	**Single GRU inference (ms)**	**Overall exec. (ms)**
Open syringe	HW	2	5.3	5.9
	SW	1178	3	7.6
Fall detection	HW	2	5	5.6
	SW	4131	3.2	18.3
Cryptography	HW	2	5.1	5.7
	SW	9365	3.4	29.24
Data collection	HW	2	4.3	4.9
	SW	3241	3	13.8
LED controller	HW	2	5.3	5.9
	SW	868	3	7.64

the device, and hence when code injection attacks (a variant of control-oriented attacks) that do not use return calls, cannot be detected. However, with static binary information combined with the trace packet information such an attack can be detected offline. Since the focus of SPADE is to detect attacks on the device, we do not demonstrate off-line detection. The off-line detection scheme does not affect the performance of SPADE.

Though software tracing incurs higher static and runtime overhead, it provides the flexibility in determining the trace features and granularity of traces, thereby detecting a larger class of attacks than hardware tracing. Evidently, it can be implemented on devices even when ETB is not enabled.

5.6.4 *GRU vs LSTM*

Table 9 compares GRUs with LSTMs as a potential alternative. LSTMs achieve 100% detection accuracy with 0 FNR *and* 0-0.1% FPR, and GRUs achieve 100% accuracy with 0 FNR and 0-0.5% FPR for various applications. However, GRUs do so with much reduced overhead. Their models are at least 25% smaller and take 41% less time for a single inference compared to LSTMs.

To understand the effect of FPR, we plot the number of times the anomaly detector is triggered against its execution latency. We assume an over-the-air

Table 9: GRU vs LSTM – compares the model size, single runtime inference, and anomaly detection accuracy. In order to achieve 100% accuracy, the false negative and false positive rates are tabulated. Applications with least and highest FPR are tabulated.

		Model size (KB)	Inference (ms)	Accuracy (%)	
				FNR	FPR
Open syringe	GRU	14.64	3	0	0
	LSTM	19.76	5.3	0	0
Data collection	GRU	14.62	3	0	0.5
	LSTM	19.74	5.2	0	0.1

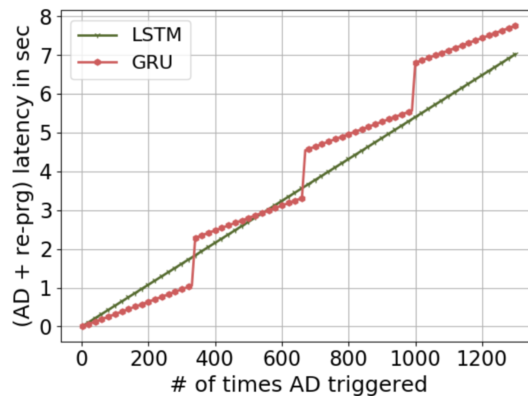


Figure 22: The graph shows the effect of 0.3% FPR on the anomaly detector's performance. A false alarm triggers unnecessary reprogramming (the peaks in GRU). The peaks in GRU indicates reprogramming.

reprogramming is performed when an anomaly is detected. Hence, a false alarm reprograms the device. A reprogramming cost of 1.2 s is considered [38]. Fall detection application has an FPR of 0.3% for GRUs and 0 for LSTMs. With 0.3% FPR, a reprogramming happens after anomaly detector is triggered over 330 times.

For fall detection application, Figure 22 shows that with 0.3% FPR, a reprogramming happens after anomaly detector is triggered over 330 times. A trigger to the anomaly detector depends on the application, where it can be once in days, weeks or months. In the fall detection application anomaly detector is executed when there is a read for threshold configuration or when the actuator is triggered. Assuming the anomaly detector is executed once every hour, a 0.3%

FPR raises a false alarm indicating an anomaly is detected once in two weeks. A lower FPR or an infrequent trigger to the anomaly detector will further reduce the false alarms. In contrast, LSTMs do not falsely detect anomalies. Hence, LSTMs may perform better (depends on the FPR of the application) over time than GRUs considering the reprogramming cost involved.

5.7 SECURITY ANALYSIS

In this section we consider the attacks discussed in the threat model (see Section 5.2.1), and show how SPADE thwarts them.

5.7.1 *Control-oriented Attacks*

Control-oriented attacks modify the return addresses of vulnerable functions to jump into any other functions within the application program (code re-use) or to a location in the code injected by the attacker (code injection). We demonstrate how SPADE successfully detects code re-use attacks using the open syringe application (Listing 5.1). The overflow vulnerability at Line 12 is exploited to execute the `bolus` function and inject the chemical at an unexpected time. In the application, the `bolus` function is always invoked from `process_cmd`. The attacker exploits the buffer overflow vulnerability in the `config` function to jump directly to the `bolus` function. In SPADE, caller-callee relationships are used as trace features for model training. Since the trace entry `config->bolus` is not seen during training and occurs only at runtime, SPADE can successfully detect such an attack (**G1**). With hardware tracing, where only return addresses are considered, the GRU model that remembers long-term dependencies detects such an attack due to incorrect order of return addresses. Code-injection attacks are detected similarly.

5.7.2 *Stealthy Attacks*

Stealthy attacks are intricately crafted to resemble the normal execution of a program.

Aberrant path attacks. While control-oriented attacks induce illegal control paths, aberrant path attacks do not take any illegal control path making them more difficult to identify. In order to induce an aberrant path attack, a decision-making variable is modified to illegally execute critical functions accessible only to authenticated users. The attack can be demonstrated via Listing 5.1, where an attacker overflows the `buffer` at Line 12 to set the `authenticated`

flag. Though function auth_password returns false, the attacker can access critical functions (Line 26-Line 28).

As shown in Table 3, under normal authenticated execution, after config->auth_password, config->serial_write and other following functions are executed. However, if auth_password returns false (wrong password), config->serial_write, read_cmd, and the following functions are never invoked. The attacker exploits a vulnerability to execute read_cmd and the following critical functions even after auth_password returns false. The GRU-based anomaly detection scheme considered by SPADE remembers long-term sequences and hence can detect such an attack (**G2**) with 100% detection accuracy (see Figure 18).

Mimicry attacks. As can be seen in Table 3, in order to evade detection as an aberrant path attack, an attacker can invoke the missing functions between config->auth_password and config->read_cmd to mimic a normal execution of an authentic case. However, to mimic the normal sequence, the attacker has to modify either of the following: 1. the application binary; 2. runtime traces; 3. the anomaly detection process to simply return true always. SPADE leverages the advantages of hardware security provided by TrustZone. When an attacker tries to install a modified binary, the secure boot fails and raises an alarm. Runtime traces are stored in the secure region, and the tracing is performed securely (see Section 5.4.3). Hence, the attacker cannot modify the runtime traces. For the same reason, the attacker cannot tamper with the anomaly detector module, which resides in the secure region (**G3**).

5.7.3 *Modifying Checkpoints*

An attacker can try to load a modified firmware to remove a checkpoint. However, the secure boot process of the trusted hardware prevents the device from booting with an untrusted firmware. Instead of attempting to replace the firmware, an attacker can try to exploit a vulnerability in the application program to skip a checkpoint. An attempt to skip a checkpoint incurs a control-oriented attack. Such an attempt is futile because the watchdog module will periodically trigger anomaly detection to identify changes in the control flow of the program.

5.8 DISCUSSION

Benchmark applications and their complexity. At the time of writing, there are no standard benchmarks for bare-metal embedded devices. Hence it is typical in related work to choose a certain number of applications (1 to 5) for evaluation [17, 29, 3, 104]. Most of the embedded applications are simple, performing a single task which reflects in our chosen low-complexity applications (light controller). However, SPADE can be applied to a more complex application on a higher-end IoT device, when ensuring that the training data is collected with a high code coverage, and tuning the model hyperpameters to achieve a good prediction accuracy.

Criteria for checkpoints. For detecting anomalies with high accuracy, the anomaly detector can be triggered after executing sensor and actuator APIs, where critical tasks that can alter the behavior of an embedded program are executed. A programmer is the best judge in identifying such APIs. However, one can always eliminate the need for introducing the checkpoints by simply triggering our anomaly detector periodically at the potential expense of increased overhead.

Security limitations. In order to achieve the security goals described in Section 5.2.2, the current implementation of SPADE runs an anomaly detector at critical locations considering recent trace data. If the attack takes place very early SPADE may not be able to catch it depending on trace window length and checkpoint location. However, this is a performance choice and one can trigger an anomaly detector at the beginning to identify such an attack.

Also, if the trace during an aberrant path attack is exactly the same as a benign trace, the attack will go undetected. Due to several software modules (Figure 21) involved in a program, running into such a corner case is very unlikely, as also seen in related work [101].

5.9 CONCLUSION

In this paper, we proposed a novel *secure* program anomaly detection technique for embedded IoT devices called SPADE. SPADE can detect control-oriented attacks that introduce illegal control flows. SPADE also detects intricately crafted stealthy attacks such as aberrant path and mimicry attacks that do not introduce illegal control flows and closely resemble a normal execution. SPADE runs on embedded devices to detect attacks at runtime.

We implemented a GRU-based anomaly detection scheme to remember *long-term dependencies*, which helps detect aberrant path attacks. To create a GRU model, we consider fine-grained traces with *precise call sites* as calling context for function invocations. The traces are collected using a hardware-based approach where we leverage an embedded trace buffer (ETB) from an on-board debugging component, as well as an alternate software-based tracing, where the source code is instrumented. SPADE incurs 11.3% average code size overhead with software-based tracing and only 2% with hardware-based. The overall execution time of software-based tracing is on average 52% higher than hardware-based tracing. However, we also showed that software-based tracing provides flexibility in determining the granularity of the traces and the trace features, thereby detecting a larger class of attacks than hardware-based tracing. SPADE achieves 100% detection accuracy with 0-0.5% false positives and 0% false negatives. We evaluate static and runtime overhead of SPADE using real-world embedded applications, demonstrating feasibility of our solution.

The GRU model introduces a higher overhead compared to hardware-based tracing and anomaly detection scheme. There have been some efforts in the literature on model pruning and compression for neural network architectures [118], which we consider exploring in future work along with other techniques to minimize the overhead of the GRU model.

6 OPADE: ONLINE PROGRAM ANOMALY DETECTION ON EMBEDDED IOT DEVICES

CHAPTER CONTENTS

6.1 Introduction 90
6.2 Problem Statement 92
6.2.1 Behavioural Control Anomalies 93
6.2.2 Threat Model 94
6.2.3 Example: A Vulnerable Insulin Pump 95
6.3 OPADE 96
6.3.1 Overview 96
6.3.2 Features for Anomaly Detection 98
6.3.3 Tracing 100
6.3.4 Online Anomaly Detection using HTMs 101
6.4 Implementation 104
6.4.1 Source Instrumentation 104
6.4.2 Hierarchical Temporal Memory Algorithm 105
6.5 Evaluation 105
6.5.1 Experimental Setup 105
6.5.2 RQ1: Systematic Evaluation Program Behaviour Anomalies 107
6.5.3 RQ2: Real-world Attack Detection 108
6.5.4 RQ3: Overhead 111
6.6 Conclusion 113

6.1 INTRODUCTION

Embedded IoT systems permeate our daily lives through various applications like smart homes, healthcare wearables, and more. Due to the advancements in hardware and software domain, in most of these IoT applications, the processing has moved from remote servers to edge and end nodes. As a result,

embedded IoT devices are now required to handle sensitive information or perform critical tasks, which has made embedded devices an attractive target for cyber attacks. An attacker in control of an IoT device can completely alter the behavior of the embedded program and can also instigate a severe disruption by controlling the entire back-end through these Internet-connected embedded devices.

In most recent software attacks, the vulnerabilities present in the code were exploited to gain access to the device. For example, 66% of the vulnerabilities reported by CERT use buffer overflow [13]. In the literature, solutions are proposed to detect such memory corruption attacks. However, the proposed mechanisms are inapplicable to run on embedded devices, or the attack detection is not realtime. The detection is run later on data collected from embedded devices. Whenever an attacker exploits the vulnerability to take control of the target device, the attacker immediately executes the intended malicious action, like changing the quantity of chemicals or remotely controlling the steering wheels, brakes, and engine of a car. Considering the impact of the attacks, it is vital to detect the attack when it takes place on the device and stop any further execution or escalate it to the concerned users for immediate action.

In this work, we propose a lightweight solution that runs on an embedded device and monitors the behavior of a running program to detect changes in the program's behavior at runtime. Several attacks have been discovered in the literature, and solutions have been proposed to address them individually. In this chapter, we propose to detect BCAs, irrespective of the attack causing them. We define a BCA as an anomaly seen during program execution that affects the behavior of a program by modifying one or several control aspects. We further classify a BCA into three types: (1) control flow anomaly that occurs when an attack changes the control flow of a program, (2) control branch anomaly that indicates that an illegal control branch is executed without altering the control flow of the program, and (3) control intensity anomaly that stipulates the frequency of a control loop is altered.

In order to detect the BCAs in realtime, we propose an online program anomaly detection for embedded IoT devices– OPADE. To detect the anomalies, we capture various program features during the execution. In particular, we capture the sequence of functions executed with their precise calling sites as context, the number of times a function call is invoked, and the loop execution cycle count. We use a combination of source code instrumentation and hardware performance counters to capture the various features. We implement an online anomaly detection algorithm based on HTM [57]. We implement a proof-of-concept prototype on Raspberry Pi, a popular choice for building IoT applications. We evaluate OPADE for its accuracy in detecting real-world

attacks and synthetically generated anomalies. OPADE introduces 11.3% of static overhead and 3.4% of runtime overhead on average.

The rest of the chapter is organized as follows. Section 6.2 presents the definitions of various behavioral control anomalies and threat models considered. Section 6.3 details the design of our proposed solution. Section 6.4 presents implementation details. Section 6.5 evaluates OPADE on various case study applications. Finally, Section 6.6 concludes the chapter with possible future research directions.

6.2 PROBLEM STATEMENT

In this section, we first introduce behavioural control anomalies, then present our threat model and the challenges involved in detecting anomalies during program execution.

```
❶ auth = read()
❷ if(auth) privileged = true;
❸ else privileged = false;

❹ if(privileged) {
      ...
❺     control_actuator();
  }
  else {
      ...
❻     retry();
  }

 void control_actuator(){
❼     steps = read();
      for(i = 0; i < steps; i++){
          ...
❽     write();
      }
      ...
 }
```

Listing 6.1: A sample embedded control program where an authenticated privileged user controls the actuator.

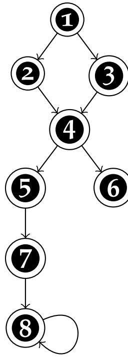


Figure 23: Static control flow path for the sample application program in Listing 6.1.

Table 10: Behavioural control anomalies observed during execution of the program in Listing 6.1.

Anomaly	Control path	Path valid?
CFA	❶ ❸ ❼	invalid
CBA	❶ ❸ ❹ ❺	valid
CIA	... ❽ ❽ ❽ ...	valid

6.2.1 *Behavioural Control Anomalies*

In this section, we present behavioural control anomaly (BCA), an anomaly seen during program execution that affects the behavior of a program by modifying one or several control aspects. Listing 6.1 shows a sample embedded program that provides an authenticated user privileged access to control an actuator. Figure 23 shows a valid control flow execution path for the program in Listing 6.1.

We further categorize BCA into three types:

A1 Control flow anomaly: A CFA is a change in the control flow of a program. Such an anomaly violates the static control flow graph of the program as seen in Tab. 10.

A2 Control branch anomaly: A CBA indicates that the control branch of a program is altered; however, it is still a legal control flow. A CBA will not violate the control flow graph. As seen in Figure 23, there exists a valid path through ❶, ❸, ❹, ❺. But, in Listing 6.1, only a privileged user (❷) should access the critical function `control_actuator` (❺). Here, an external attacker modifies the `privileged` variable by exploiting a buffer overflow vulnerability in `read()`. Hence, during the program execution, a non-accessible branch was traversed illegally.

A3 Control intensity anomaly: A CIA shows that the intensity of a control loop is altered by executing the loop an illegal number of times. In Listing 6.1, the `write` function (❽) can be executed several times. Here, the attacker

increases the `steps` variable to a large number which can have a catastrophic effect.

6.2.2 *Threat Model*

We discuss the attacker abilities assumed and the attacks that can cause a BCA.

Attacker abilities. We assume that an attacker can exploit vulnerabilities present in an embedded application to launch software exploits, thereby gaining access to devices. We also assume that there are no internal threats in the development setting. These basic assumptions are common to existing program anomaly detection works [29, 119, 3].

Attacks. A *Control-oriented attack* exploits vulnerabilities in a program to corrupt control data such as return address or code pointer in the program memory to divert the control flow of the program. The control flows may be redirected to a location in existing (code-reuse attacks) or newly injected code (code injection attack), thereby introducing illegal control flows. In contrast, a *data-oriented attack* manipulates non-control data to alter the behavior of a program. The attack exploits vulnerabilities to corrupt critical data variables or data pointers without violating the integrity of control flow paths. A comparatively recent, *data-oriented programming attack* systematically constructs non-control data exploits. Before the attack, data-oriented gadgets are identified, which are a short sequence of instructions. Then gadget dispatchers are used to chain the disjoint gadgets in a sequence to carry out an attack.

Figure 24 shows the relation between the various runtime attacks and the BCAs. We further explain how these attacks can be carried out that cause various BCAs.

- A control-oriented attack modifies the control flow of a program and hence introduces a CFA.
- Data-oriented attacks - A data-oriented attack corrupts data variables which can have a differing effect on the program behavior depending on how the data variable is used. An attack that modifies a critical decision-making variable may introduce CBA. An example in Listing 6.1 shows how an authentication variable can be altered to illegally access the control branch (❺) of a program accessible only to authenticated users. A data-oriented attack that modifies the data variable to manipulate the frequency of control operations may introduce CIA. E.g., the number of loop iterations can be altered to inject large amounts of a drug (❽).

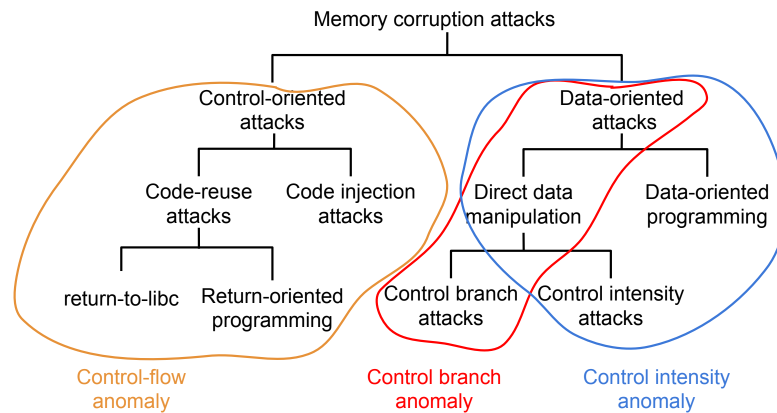


Figure 24: The figure shows various runtime attacks and the BCAs caused by these attacks.

- Data-oriented programming - Data-oriented programming attack has two steps: finding gadgets and chaining them in an arbitrary sequence using a dispatcher. The dispatcher is usually a loop [59] because, within a loop, an attacker can execute a sequence of instructions several times to achieve the desired effect. Hence, a data-oriented programming attack may introduce a CIA.

6.2.3 *Example: A Vulnerable Insulin Pump*

We present a simple example of the control program used to operate an insulin pump. The program shown in Listing 6.1 is based on the working of insulin pump [91] since the actual source code for the insulin pump is proprietary. An insulin pump is a small device that regularly delivers insulin throughout the day called a basal dose. When required, insulin can also be delivered as a surge, called bolus dosage. In order to indicate the basal and bolus dosages, the insulin pump usually communicates with an external remote controller. The users can input and program the insulin pump through the remote controller. Several attacks on insulin pumps have been demonstrated in the past [70, 108]. Listing 6.1 shows a sample attack on an insulin pump. When the insulin pump receives a packet from a remote controller, it checks if it is from a paired controller. The function `recv_pkt` has a buffer overflow vulnerability that the attacker exploits to carry out various attacks depending on the level of expertise. The attacker can modify the return address of `control_loop` to jump `bolus` function and

Table 11: Sample of traces under normal conditions, during a control-oriented attack, aberrant path attack, and mimicry attack.

Benign - auth	Benign - fail	CFA	CBA
`loop->recv_pkt`	`loop->recv_pkt`	`loop->recv_pkt`	`loop->recv_pkt`
`loop->verify_device`	`loop->verify_device`	`loop->verify_device`	`loop->verify_device`
`loop->get_insulin_units`	`loop->config_mesg`	`loop->config_mesg`	`loop->config_mesg`
`loop->bolus`		`loop->bolus`	`loop->get_insulin_units`
`bolus->hw_inject`			`loop->bolus`
loop-ID 127238			`bolus->hw_inject`

inject insulin at the wrong times (control-oriented attack). The attacker can also exploit the vulnerability to simply change `paired_device = 1`, thereby illegally triggering the `bolus` function (data-oriented attack). The attacker can also modify the `bolus_units, BASAL_UNIT` to change the amount of insulin injected when the bolus is invoked (data-oriented attack).

With the next-generation closed-loop implementation of an insulin pump, where a continuous glucose monitor measures realtime glucose values and adjusts the dosage on the insulin pump without manual programming, the attacks demonstrated will be more relevant.

6.3 OPADE

This section details our program anomaly detection scheme and the program features considered for anomaly detection.

6.3.1 *Overview*

The various software modules running on an IoT device are shown in Figure 25. ARM TrustZone isolates processes, peripherals, and memory regions into secure (green; right) and non-secure regions (red; left). Importantly, OPADE runs in the secure region and the underlying operating system (if any) and user applications run in the non-secure region.

OPADE further contains a software buffer that stores the features collected from a user application. The instrumented application invokes a *buffer writer* module, which structures and stores the features in a software buffer. A *buffer reader* reads the partial traces from the buffer and encodes it into the format

```
/* Insulin pump injects basal units of insulin periodically */
unit8_t BASAL_UNIT = 2;

/* Control insulin bolus */
void bolus(int units) {
  int i;
  int bolus_units = units / BASAL_UNIT;

  for(i = 0; i < bolus_units; i++){
    // invoke hardware to inject insulin
    hw_inject();
  }
}

void control_loop() {
  int paired_device = 0, units;
  pkt_t pkt;

  while(1) {
    // pkt received from a remote controller
    if(recv_pkt(&pkt){ // vulnerable
       if(verify_device(pkt)){
        // pkt is from a paired device
        paired_device = 1;
       } else {
        // pkt from an unknown device
        config_mesg();
       }

       if(paired_device){
        // get units from pkt received
        units = get_insulin_units(&pkt);
        bolus(units);
       }

    }
}
```

Listing 6.2: An example of a control loop in insulin pump.

required by the *anomaly detector*. Then, the anomaly detector executes the HTM algorithm within a time window periodically that checks the runtime traces against its learned program behavior. The main objective of the buffer reader and buffer writer modules are to continuously provide the traces in a streaming fashion to the HTM algorithm for online and continuous anomaly detection.

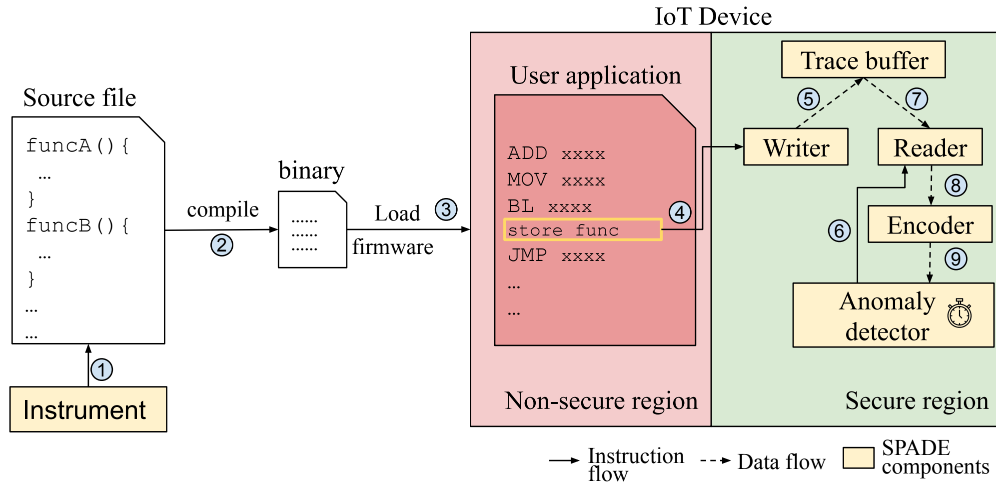


Figure 25: Overview of OPADE.

6.3.2 *Features for Anomaly Detection*

This section discusses various features collected during program execution for detecting anomalies presented in Section 6.2.1.

Sequence modeling with calling context. We capture the behavior of a program and design an anomaly detection scheme. Our scheme can jointly learn the semantics of individual function calls and their interactions appearing in the trace sequences. Existing solutions are limited to coarse-grained features and pairwise function call patterns. They are, therefore, suitable for only attacks that exhibit deviations in short traces. Our anomaly detection approach directly addresses these limitations by capturing short- and long-term patterns with precise function invocation locations.

As shown in Table 11, though each entry of CBA reflects a legal control path (e.g., `control_loop->recv_pkt`), the sequence as a whole is illegal. The program in Listing 6.2 shows that `config_mesg` is executed when the pump receives a packet from an unknown controller and `get_insulin_units, bolus` is executed for a paired device. Hence, the functions invoked in the order

verify_device, config_mesg, get_insulin_units are illegal. In order to detect such an attack, it is important to preserve the ordering of function call occurrences. We use HTMs to capture short- and long-term patterns and achieve order sensitivity.

Algorithm 7: Handling loops while collecting execution sequence trace.

```
  Variables:
1 upon FUNC_ENTERED(func, call_site)
2   if func = temp_func ∧ call_site = temp_call_site then
3     func_called ← func_called + 1
4   STORE(func, call_site, func_called)
5   temp_func ← func
6   temp_call_site ← call_site
7   func_called ← 0
```

HANDLING LOOPS: The functions invoked within a loop will appear several times in the trace depending on the loop iterations. A large loop iteration count may fill the trace buffer quickly, leading to the loss of important trace information, leaving us only with a sequence of the same function. Also, OPADE detects anomalies by learning the sequence of function calls. The same function appearing several times does not provide new information for the algorithm to learn.

Algorithm 7 shows how OPADE handles loops. Before storing a function `f` entry to the trace, we check if the previous function stored is `f`. If the same function is invoked several times, we increment the counter of this function. The final output stored in the trace looks like `f1->f2:n`, where `n` indicates the number of times `f2` is invoked from `f1`. For the example program in Listing 6.2, the function `hw_inject` which is executed within a `for` loop will be stored as `bolus->hw_inject:bolus_units`

Program behavior modeling using precise call sites. In OPADE, we capture short- and long-term patterns and their precise calling location. Capturing the precise location enables us to detect a broad range of attacks, thereby improving the accuracy of detecting an attack.

For the program in Listing 6.2, during program execution, a control-oriented attack generates `control_loop:38->bolus` as execution sequence, which is a CFA. However, without a precise call site, the 1-level calling context trace during the same attack shows `control_loop->bolus`, which is not an anomaly because there exists a legal control path.

Loop cycle count. Modeling the control sequence of a program can detect attacks that modify the control branch during execution. However, there are attacks that corrupt a data variable to manipulate the number of control operations [53]. For example, an attacker may corrupt the state variable that controls the number of times a loop is executed. In Listing 6.2, the local variable `units` can be modified to alter the amount of insulin injected in `bolus`, which can have a catastrophic effect.

We capture the number of cycles consumed to execute a loop which aids in detecting attacks on loop iterations. When capturing loop execution cycle count, it is crucial to have high precision and low overhead. A low-precision counter may not capture loop anomalies that increase the loop iterations by a small number. In order to differentiate the loops in the program, we assign a unique ID for each loop. Section 6.3.3 provides a detailed description of how the loop execution cycle count is collected. For nested loops, we capture the execution cycle count for the outer loop.

6.3.3 *Tracing*

We extract various features from the application to detect an attack successfully at runtime. The feature collection process impacts the accuracy of the model in detecting anomalies. OPADE uses an online detection algorithm that learns on the fly and runs on the same embedded device. Hence, the overhead to collect the trace data has to be kept minimal to ensure minimal effect on the application's performance.

We perform fine-grained source code instrumentation to collect function entries, their precise call sites, and the number of cycles required to execute critical loops. The collected features are used to model the normal behavior of a program. Any deviation from the normal behavior of a program is flagged as an anomaly, and by using the combined features, we can detect BCA presented in Section 6.2.1. Software-based tracing provides flexibility in tracing granularity and features captured.

For the program listing shown in Listing 6.2, `bolus` is invoked from `control_loop` at Line 34. Assuming – for ease of presentation only – that the line numbers represent actual addresses in the application binary, `control_loop:34->bolus` represents a 1-level calling context. Table 11 shows sample traces collected using software tracing.

Function tracing. In order to capture the function traces and the precise call sites, OPADE instruments all the function entries. When capturing the information about a function entry, we also store the information of the caller site

by reading the EAX register that stores the return address of the invoked function [110]. By reading the return address from the stack, we avoid additional instrumentation for capturing the caller site information.

Loop execution cycle count. In order to capture the execution time of a loop, we use hardware performance counters commonly available on IoT devices. In addition, we leverage the cycle count register of performance monitoring unit (PMU) to calculate the time required for executing a loop. A PMU is essentially a set of registers and counters of a processor that one can program to capture various events during the execution of an application. At the end of a measurement, the PMU software provides the aggregated results of the counter values. The advantage of using a hardware counter is its high precision and low overhead. In addition, a high-precision counter is advantageous for anomaly detection because there may be loops with very few iterations. If this loop is vulnerable to attacks, capturing the time taken to execute the loop with high precision detects minor variations that can occur during an attack.

6.3.4 *Online Anomaly Detection using HTMs*

Background on HTMs. Hierarchical temporal memory (HTM) [56] is a biologically inspired technology that mimics the behavior and architecture of the neocortex, the largest area of the human brain. HTM by design supports sequence and continual learning. Hence, HTM does not require separate training or manual intervention. Also, HTM allows continuous online learning where the applications can learn new patterns without any retraining.

HTMs represent data using sparse distributed representation (SDR). SDR is a binary vector with only a small percentage of active bits, typically less than 2%. The bits in the SDR correspond to the neurons in the neocortex and are based on the observation that only a small percentage of neurons are active in the brain at any point in time. The most important property of SDR is the semantic property. SDRs that have active bits in the same location are semantically related. The more common active bits, the more semantically they are similar.

HTM learning algorithms work on SDRs. There are encoders designed to convert different data types - it could be a GPS location, temperature, image, time, or just a number - into a SDR that can be recognized by the HTM algorithm. The key component of HTM networks is the spatial pooler that continuously encodes streams of inputs into SDRs. The objective of the spatial pooler is to learn feedforward connections and form efficient input

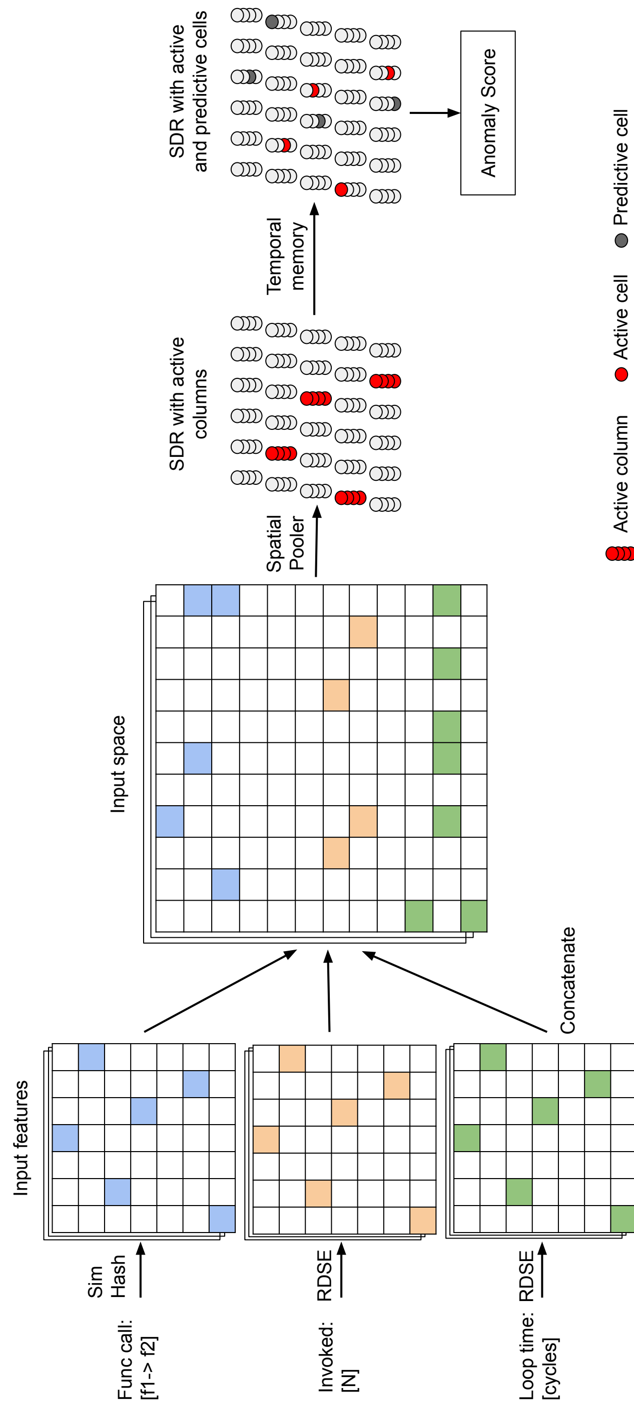


Figure 26: Architecture of HTM used in OPADE for program anomaly detection.

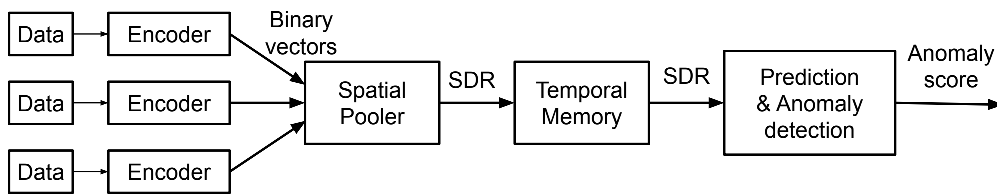


Figure 27: Workflow of HTM program anomaly detection.

representations. The learning algorithm in HTM is called temporal memory, which is responsible for learning sequences of SDRs, which are formed by spatial pooling algorithm, and making predictions. Temporal memory learns by storing the transitions in a data stream. The memory of HTM is updated every time there is a change in input and predicts what to expect next. This prediction is then compared to what happens next and minimizes the prediction error, thereby building a predictive model. The main advantage of continuous learning is that it can adapt to changing patterns on the fly without retraining outside the device.

Encoder for traced features. In OPADE we capture the following features for detecting anomalies during program execution:

- Sequence of function calls
- Precise calling location as context
- Number of times the function call is invoked
- Loop execution cycle count

Table 11 shows a partial sample of the collected program execution trace. A function call with its precise calling context is represented as `caller_loc->callee`. The number of times a function is invoked (because it is executed within a loop) is represented as `caller_loc->callee:n`, where `n` indicates the number of times `caller_loc->callee` is executed. In Listing 6.2, Line 10 shows the example of a loop execution that is represented as `loop-ID:cycles` in traces, where `cycles` is the number of cycles it took to execute the loop with identifier `ID`.

In OPADE, we use random distributed scalar encoder (RDSE) to encode the number of times a function is invoked and the loop execution cycle count. Given a numeric scalar value, RDSE encodes the number into an SDR. The RDSE encodes input patterns where the on-bits are distributed along the output of the encoder. The benefit of RDSE is that it does not require a minimum or maximum of an input range, and the encoding is determined at runtime [22]. A function call represented as `caller_loc->callee` is encoded using

SimHash document encoder (SHDE). A SHDE encodes text and documents into SDRs. SHDE is a locally sensitive hash algorithm that was first proposed by Charinikar et al. [27]. The encodings of similar words will share similar representations of SDRs. The SHDE maintains bitwise similarity by calculating hamming distance, where the words with small hamming distance have high overlap.

Program anomaly detection. HTM offers many features which prove to be advantageous for program anomaly detection: higher-order temporal prediction abilities, online learning, which can detect anomalies in real-time, and adaptability in a streaming setting which makes the algorithm robust to changes in the data, as well as noise.

In order to detect anomalies during program execution OPADE runs HTM's spatial pooler and temporal memory algorithms. OPADE executes the algorithms periodically with the collected runtime traces. At time `t`, the encoded runtime data is fed into the spatial pooling algorithm. Each layer in HTM network is structured as mini-columns, and the spatial pooling algorithm activates/deactivates the columns based on the input. The output of spatial pooling is an SDR. The temporal sequences from this glspl sdr are learned by the temporal memory algorithm by activating the individual cells in the mini-columns. The activated cells, in turn, will cause a few other cells to enter into a predictive state based on the connections between the cells at time `t`. At time `t+1`, when the subsequent encoded data arrives, again the spatial pooling and temporal memory algorithms are executed. The temporal memory calculates the anomaly score based on the number of cells in the predictive state activated in time `t+1`.

6.4 implementation

The various components of OPADE (see Figure 25) is implemented in the C programming language. Furthermore, the code in the secure region does not interact with the outside world, minimizing the attack vector space. This section highlights several implementation issues.

6.4.1 *Source Instrumentation*

The C source files of an application are instrumented to collect function entries with their calling location. For instrumentation, an application is compiled with the GNU option `finstrument-function`, which invokes a pre-defined function during the entry and exit of every function in the application. However, the

underlying OS and additional library functions cannot be captured with only source code instrumentation. Hence, when an OS is used, we also capture *system and library calls* with its calling locations. We use `strace -i` for system calls and `ltrace -i` for library calls, respectively. Note that no additional instrumentation is performed for capturing the precise call location.

In order to capture loop execution count, we instrument loop constructs of C programming language (e.g., `for, while`). In addition, a Python module is implemented in order to instrument the loop constructs.

6.4.2 *Hierarchical Temporal Memory Algorithm*

The HTM architecture (see Figure 26) for anomaly detection is implemented using the C++ library `htm.core`. In order to find optimal parameters of HTM for a given application, we perform an offline parameter optimization. For parameter optimization, we run an application for 1 hour to collect the traces. Then, we perform particle swarm optimization to fine-tune the model performance.

6.5 EVALUATION

We extensively evaluate the performance and detection accuracy of OPADE. In particular, we address the following research questions:

RQ1: What is the accuracy of OPADE in detecting program behaviour anomalies?

RQ2: How effectively does OPADE detect real-world attacks that cause BCAs?

RQ3: What is the overhead incurred by OPADE?

6.5.1 *Experimental Setup*

Case studies of embedded applications. In order to evaluate the performance of OPADE, we consider five real-world embedded control programs. A control program will sense the environment, perform lightweight processing, and, depending on the results, act on the actuators present on the device.

FALL DETECTION: A simple threshold-based fall detection [17] is implemented using a 3-axis accelerometer sensor. The application verifies the input credentials and allows an authenticated user to configure the threshold value. The fall detection algorithm monitors the 3-axis accelerometer values and raises the alarm if the computed fall value is greater than a configured threshold.

OPEN SYRINGE: The open syringe application has been widely used in the literature to showcase various attacks [29, 3, 104]. The application takes user commands to control the bolus of a fluid-filled syringe. The application accepts commands to set the quantity of liquid to be dispensed, and commands to push/pull the syringe by the set quantity value. We use the serial port for input and output.

MQTT SUBSCRIBER AND PUBLISHER: MQTT is a standard messaging protocol used in IoT. It is lightweight and designed for low-bandwidth networks and high-latency networks. MQTT works on the principle of publish/subscribe model with a central broker. We use the open-source MQTT-C implementation, which is written for embedded systems and systems alike [18]. For all our experiments, we run publisher and subscriber on the target hardware and simply combine the final results.

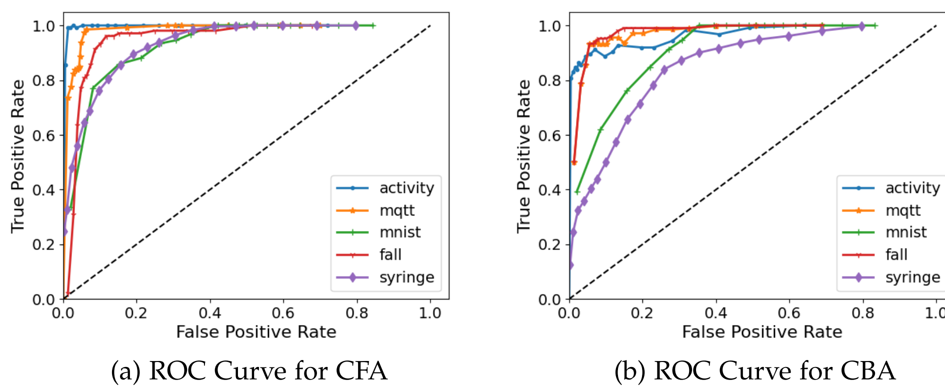
MNIST: A feed-forward neural network is trained to classify the numbers from a popular MNIST dataset [73]. The program is written in C. The trained model is saved on the embedded device, and we run only the prediction on the device. The program [73] reads 200 images stored in a `csv` file and returns a score based on the number of images classified correctly.

ACTIVITY DETECTION: The application reads the accelerometer and gyroscope data from the embedded device and detects if the activity is a walk or a run. The application uses a 3-axis accelerometer data similar to fall detection; however, instead of threshold-based detection, a machine learning model is trained and used for activity detection. We use the open-source code [113] for training the model and convert it to a TensorFlow-lite model using the C++ library [106]. Finally, prediction using TensorFlow-lite in C++ is performed on the embedded device.

Choice of hardware. We demonstrate and evaluate OPADE on ARM-based devices (mainly Raspberry Pi), which are widely popular for building IoT applications. In addition, we use OP-TEE as the OS for a trusted execution environment and Linux as the normal world OS. OPADE does not use any OS-specific features and is designed for bare-metal embedded devices. `strace` and `ltrace` are used in OPADE along with application traces only if they are available. Otherwise, OPADE instruments all the available source code making it easily applicable to bare-metal systems.

Table 12: Code coverage.

Applications	Lines Executed	Branches Executed	Calls Executed
Fall detection	100%	100%	100%
Open Syringe	82.76%	100%	95.45%
mnist	72.83	70.16%	68.7%
tflite	100%	100%	100%
MQTT	66.09%	83%	68.4%



(a) ROC Curve for CFA (b) ROC Curve for CBA

Figure 28: The graph shows the ROC curve indicating the rate of false positives and true positives when the threshold for anomaly detection is varied. (a) shows the ROC curve for CFA and (b) shows the ROC curve for CBA for various applications.

6.5.2 *RQ1: Systematic Evaluation Program Behaviour Anomalies*

We systematically demonstrate the accuracy of OPADE through ROC and PR curves.

Data Collection. For systematic analysis, we first collect the data from the application running on the device to an external system. We then introduce synthetic anomalies to evaluate the detection accuracy. Next, we execute each case study application for 1 hour, during which the user interacts with the applications at random times, and collect the data. The code coverage for each application is shown in Table 12. We use the open-source Linux tool GCOV to calculate the code coverage. Finally, the average of all the application files is presented for each application.

Synthetic Anomalies. We generate three types of synthetic anomalous traces for each embedded application considered. Given an application trace T of length N, we generate T_a with the following anomalies:

- CFA: Given T, 10% of N entries are modified to mimic a CFA. We first generate a set of 20 addresses that is not present in T. To create a CFA, we randomly choose a function trace entry `fx->fy` in T and replace the calling location `fx` with one of the 20 generated addresses.
- CBA: Given T, 10% of N entries are added to mimic a CBA. First, an anomaly pool is created by randomly picking 20 function trace entries from T. To generate the anomalous application trace T_a, we introduce one of the function trace entries `fx->fy` from the anomaly pool at a random location in T.
- CIA: Given T, 5% of loop iteration entries are modified to mimic a CIA. We randomly choose a loop iteration entry in T and increase the loop cycle count by m times, where m is a random number between 2 and 10.

Detection accuracy. Figure 28 shows the receiver operating characteristic (ROC) curve for various applications. The HTM algorithm provides an anomaly score based on the prediction made at time $t-1$ and actual data seen at time t. We evaluate the rate of false positives and true positives by varying the threshold to mark a prediction as anomalous based on its anomaly score. As seen in Figure 28a, the ROC curve for CFA converges sooner because a CFA introduces a new `caller->callee` relation and the algorithm can quickly identify with high confidence. Figure 28b shows that the ROC curve for CBA also converges, but, at a higher threshold because CBA does not introduce any unknown `caller->callee` entry. Instead, the algorithm learns the long-term pattern to identify CBAs.

6.5.3 *RQ2: Real-world Attack Detection*

In this section, we evaluate the performance of OPADE in detecting real-world attacks. We first deploy an application and OPADE on the target device. Then, at application runtime, we carry out one of the attacks described in the Section 6.2.2 and present the results of how OPADE detects these attacks at runtime.

Attack variants.

A1 - REDIRECT CONTROL FLOW TO CAUSE CFA: We consider the fall detection application, which has a buffer overflow vulnerability when it reads a

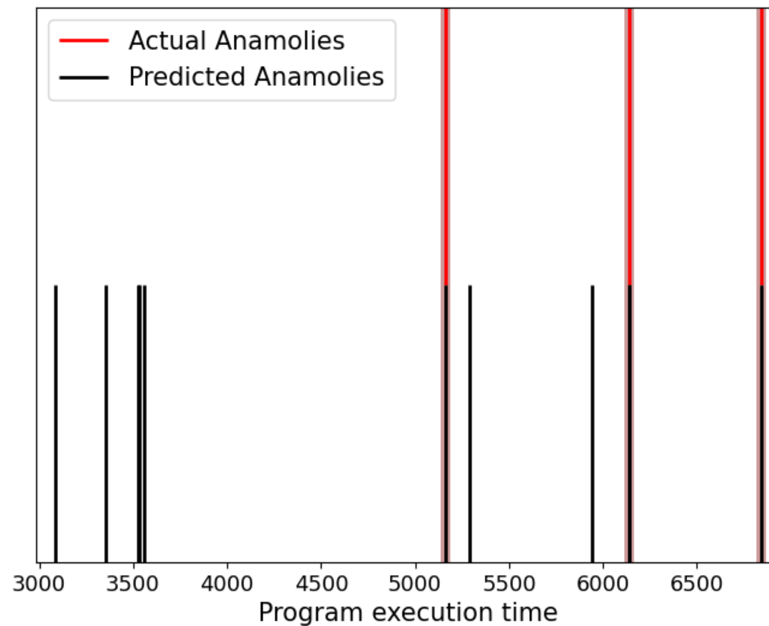


Figure 29: For fall detection application, control-oriented, and data-oriented attacks were carried out during program execution. As a result, we see CFA and CBA during the attack. The figure shows actual anomalies which are generated during the attacks and anomalies predicted by OPADE's HTM algorithm at runtime. The figure also shows the true positives where HTM predicts the anomalies correctly when there is an actual anomaly and false positives where HTM marks a benign control flow as an anomaly.

user input for configuring the threshold value. As an attacker, we falsely trigger a fall alarm even when there is no fall. At application runtime, we exploit the buffer overflow vulnerability to modify the return address and jump to the critical function in the program that triggers a fall alarm. Redirecting the control flow will cause a CFA.

A2 - OVERWRITE DECISION-MAKING VARIABLE TO CAUSE CBA: In the fall detection application, as an attacker, we change the threshold value for fall detection without the required correct credentials. We exploit the buffer overflow vulnerability in the application to override the decision-making variable and execute an `else` block (for authenticated user) instead of `if` block (for non-authenticated user).

A3 - INCREASE LOOP ITERATION TO CAUSE CIA: The open syringe application exhibits a buffer vulnerability when accepting commands from the user.

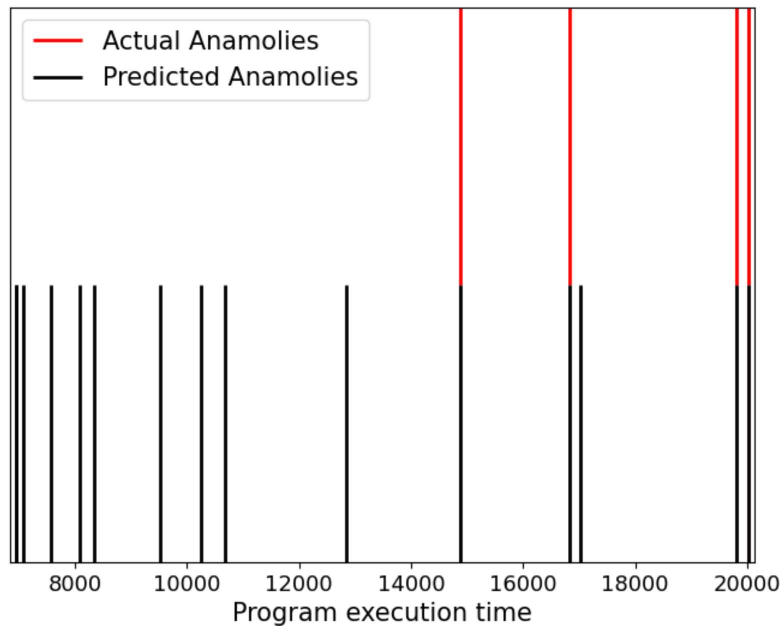


Figure 30: For open syringe application, we carry out a data-oriented attack to dispense a large quantity of chemicals during program execution. As a result, we see a CIA during the attack. The figure shows actual anomalies which are expected during the attacks and predicted anomalies by OPADE's HTM algorithm at runtime. The figure also shows the true positives where HTM predicts the anomalies correctly when there is an actual anomaly and false positives where HTM marks a benign loop intensity as an anomaly.

When the user enters a required quantity of liquid to be dispensed, the application checks if the quantity is within the required range. However, as an attacker, we exploit the vulnerability to change the quantity of liquid to be dispensed to a very large value outside the verified range.

Detection accuracy. For fall detection application, Figure 29 shows the actual anomalies CFA and CBA introduced when we redirect the control flow (A1) and overwrite a decision-making variable respectively. The figure also shows the anomalies predicted by the online HTM algorithm during runtime. Note that we consider the first 20 min of the data as a probationary period required for the algorithm to learn the program behavior. The probationary period can be increased or decreased depending on the complexity of the application. As seen in the figure, all the actual anomalies are detected by our HTM algorithm. However, we observe that the false positive rate is 0.17%.

Figure 30 shows the actual CIA anomalies seen when we, as an attacker, carry out an attack to illegally dispense liquid which is 2-10 times higher than the normal range (A3). The figure also shows the anomalies predicted as CIA by our system. Again, we consider the first 20 min as a probationary period. The figure shows that all the actual CIA anomalies are detected by OPADE, and there is 0.07% of false positives.

6.5.4 *RQ3: Overhead*

This section presents the static and runtime overhead introduced when executing OPADE.

Static overhead. We measure the static overhead in terms of increase in binary size of an application. In this section, we present the static overhead introduced by OPADE.

Figure 31 shows the original code size of various applications and the application code size when compiled with OPADE. OPADE code size overhead includes tracing functionalities in the non-secure region and anomaly detection in the secure region. OPADE incurs 11.3% average code size overhead. The absolute increase in code size ranges from 0.2 to 4.5 KB. The introduced overhead is very minimal compared to the available program memory on IoT devices. The increase in code size is not proportional to the original application code size and depends on the number of functions in the application that are instrumented. The advantage of using HTM as a learning model is that we eliminate the need to store a model on the device.

Runtime overhead. Figure 32 shows the overhead caused by running OPADE on embedded device. The overhead is caused by three major sources: executing HTM algorithm (encoding and anomaly detection), switching to secure region, and overhead caused by tracing. For evaluating runtime overhead, we use Python scripts to feed the inputs to the application program at runtime. The same script is used with and without OPADE, and we calculate the overall increase in the runtime execution of a program which is shown in Figure 32. The figure shows that the overhead caused by OPADE for each application falls between 2%-5%.

We also evaluated the time taken to execute an individual instance of HTM. In OPADE the anomaly detector is executed every T sec (ref Figure 25), and for the evaluated use cases, we choose $T = 5$. We further explain that the values of T have minimal to no impact on the runtime overhead but may impact the detection accuracy. In our evaluation, the time taken to execute a HTM, h_t

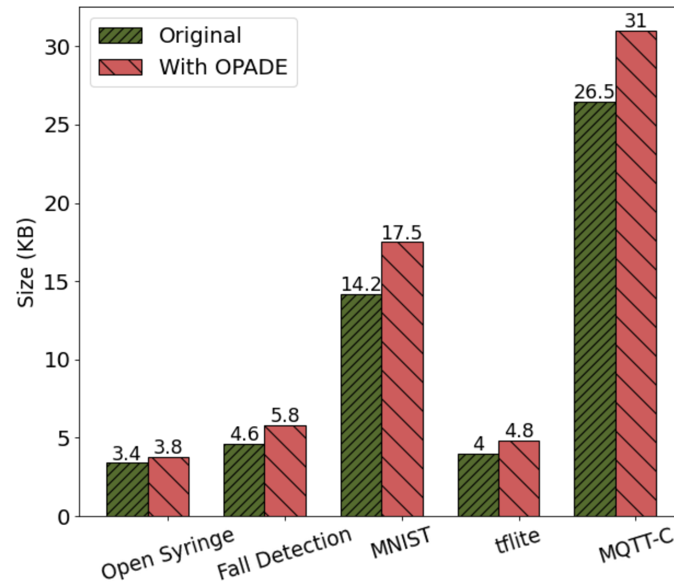


Figure 31: The graph shows the code size of various applications with and without OPADE.

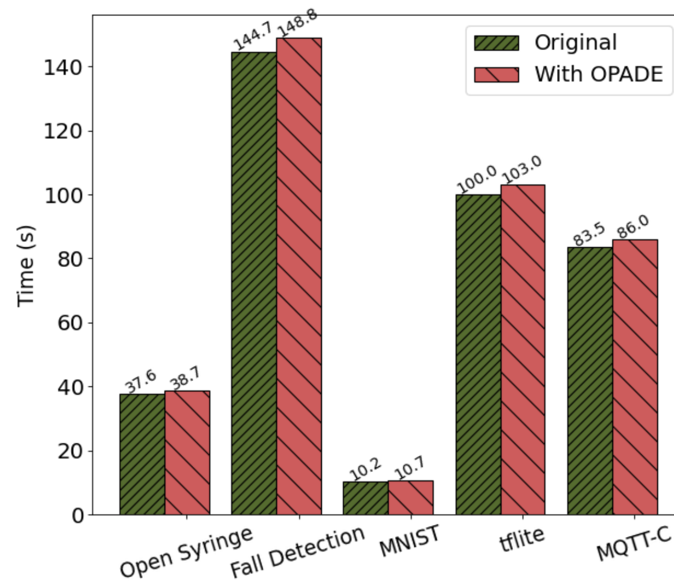


Figure 32: The graph shows the execution time of various applications with and without OPADE. The runtime overhead includes the overhead introduced by the HTM algorithm and other components of OPADE, which includes tracing and context switching overhead.

over one trace point is 7 ms. If n trace points are generated by an application within the time window T, then the overall time taken by HTM for the time period T is $h_t * n$. Since the features collected by OPADE are the same for any given application, the average h_t remains the same. Therefore, T is directly proportional to n, and varying T will have minimal effect on the overall runtime overhead.

6.6 CONCLUSION

In this chapter, we proposed OPADE, an online program anomaly detection technique for embedded IoT devices. OPADE is implemented on trusted hardware, thereby ensuring that OPADE components are secured by design. We introduced various behavioural control anomalies (BCAs), namely control flow anomaly (CFA), control branch anomaly (CBA), and control intensity anomaly (CIA) that are seen when the behavior of a program is altered during the execution of a program. We also discussed how and which control-oriented and data-oriented attacks presented in literature cause each of the BCAs.

We implemented an anomaly detection technique that uses HTM to detect BCAs. The anomaly detection algorithm considers fine-grained function calls with precise-call sites as a context for function invocations and loop execution cycle count derived from hardware performance counters. We run OPADE on synthetically generated anomalous traces and also real-world attacks on the device to evaluate the accuracy of OPADE in detecting BCAs. In both cases, OPADE successfully detected all the anomalies. A false positive rate of 0.12% on average was observed in detecting real attacks. OPADE introduces 11.3% of static overhead and 3.4% of runtime overhead on average.

7 CONCLUSION

CHAPTER CONTENTS

7.1 Dissertation Summary 115
7.2 Future Research Directions 116

In this dissertation, we presented solutions to detect software attacks targeted at embedded IoT devices. In this chapter, we summarize the contributions of the research work carried out as part of this dissertation and outline possible future research directions.

7.1 DISSERTATION SUMMARY

The increase in Internet-connected embedded devices in IoT applications has made these devices an attractive target for various software attacks. We showed that the existing mechanisms in detecting software attacks have limited applicability for embedded devices due to their high overheads, chosen platforms, and attacks detected. The main objective of this dissertation is to build trustworthy and reliable IoT applications by detecting software attacks and securing embedded IoT devices.

The proposed solutions addressed two major security goals:

Part I The first part discussed the solution to detect software tampering attacks. In IoT applications where the embedded devices interact with each other to collectively perform specific tasks are deployed as small or large-scale networks. In Chapter 4, we presented SWARNA to collectively verify the integrity of the software running on remote IoT devices employing remote attestation. Existing works on swarm attestation require trusted hardware, whereas SWARNA is a pure software-based solution for swarm attestation. Software-based attestation solution works on the principle of time guarantees. Hence, we also designed deterministic communication paths for attestation that enforces strict time bounds across multi-hop networks using IEEE 802.15.4 time-slotted MAC protocol. We also showcased that SWARNA

is secure against various network attacks and can successfully detect the presence of passive and active malware on the devices.

PART II The second part focused on detecting runtime software attacks. We proposed two secure program anomaly detection techniques that run on embedded devices to detect memory corruption attacks that occur during the runtime execution of a program. In Chapter 5, we presented SPADE to detect control-oriented attacks and stealthy attacks such as aberrant path and mimicry attacks. Control-oriented attacks introduce illegal control flows, and stealthy attacks resemble normal execution without introducing any illicit control flows. To detect the attacks, we implemented a GRU-based anomaly detection algorithm with precise call sites as a context for function invocations. In Chapter 6, we first introduced the definition of BCA, an anomaly seen during program execution that affects the behaviour of the program by modifying one or several control aspects of the program. Then, we presented an HTM-based online program anomaly detection algorithm to detect various types of BCAs. The runtime overhead to execute a single instance of HTM is constant, irrespective of the application. However, a single GRU inference time depends on application complexity and is higher than HTM. Due to the complexity of the underlying machine learning technique, OPADE can run on devices with Cortex-A processors or processors with higher processing abilities. In contrast, SPADE can run on tiny embedded devices with Cortex-M processor (processing capabilities and memory size: Cortex-M < Cortex-A).

7.2 FUTURE RESEARCH DIRECTIONS

We outline some possible future research for which this dissertation acts as a groundwork.

Hybrid Attack Detection for IoT:. In this dissertation, we presented techniques for on-device attack detection and remote attack detection using a trusted verifier. Some works use remote verification in the literature to detect memory corruption attacks and not just software tampering attacks [104, 35, 120]. However, the limitation remains in identifying the time to trigger the verification before the attack goes unnoticed (TOCTOU [88]) and late detection of an attack where the catastrophic effect has already occurred.

Anomaly detection and remote attestation for memory corruption attacks require traces of various features during program execution. Partial traces of these features can be used locally on the device for low overhead anomaly detection, and the result can be used as a trigger for the device to send the

full traces to a remote verifier. The remote verifier, which does not have any resource constraints, can extensively investigate if there was an attack with high confidence and accuracy. However, it is challenging to maintain and verify the possible runtime states of the IoT device.

Reinforcement learning for parameter tuning:. In machine learning, hyper-parameter tuning is an omnipresent problem. We can implement a fully autonomous system with on-device anomaly detection and online learning algorithms like HTM. However, the issue of hyper-parameter tuning needs to be addressed for deploying an autonomous system. Reinforcement learning can be used to include user feedback and optimize the hyper-parameters, thereby adapting the algorithm to firmware updates. Some initial works in this direction optimize the parameters based on the model loss [67], which needs to be adapted to derive and incorporate user knowledge.

However, it is crucial to keep the overhead minimal for IoT devices; hence, we can combine reinforcement learning with hybrid attack detection. When we have a hybrid approach, the opportunities are endless. For example, we can use reinforcement learning on a remote server to include user feedback regarding anomalies. We can also use federated learning to optimize the parameters and send the learnt parameters to the end-device [63].

Research in attacks side effects. A BCA is introduced when there is an illegal control path, a data variable is altered that executes a legal-but-incorrect branch, or the altered variable affects the loop execution. Although by identifying BCAs, we can detect a large class of attacks, the attacks that do not introduce any BCA will go unnoticed. An attacker may inject a small amount of chemical several times instead of a large amount one single time or even exploit an integer overflow vulnerability to allocate more than the required memory to carry out an attack [122]. Research exploring the side effects caused by such attacks is required to detect the attacks efficiently. E.g., monitoring stack or heap usage, especially monitoring the pattern of heap allocation and deallocation, may indicate an illegal use of heap allocation.

BIBLIOGRAPHY

[1] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. "Control-flow Integrity." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2005, pp. 340–353.

[2] Muhamed Fauzi Bin Abbas, Sai Praveen Kadiyala, Alok Prakash, Thambipillai Srikanthan, and Yan Lin Aung. "Hardware Performance Counters based Runtime Anomaly Detection using SVM." In: *TRON Symposium (TRONSHOW)*. IEEE, 2017, pp. 1–9.

[3] Tigist Abera, N. Asokan, Lucas Davi, Jan-Erik Ekberg, Thomas Nyman, Andrew Paverd, Ahmad-Reza Sadeghi, and Gene Tsudik. "C-FLAT: Control-Flow Attestation for Embedded Systems Software." In: *Conference on Computer and Communications Security (CCS)*. 2016, pp. 743–754.

[4] Tamer AbuHmed, Nandinbold Nyamaa, and DaeHun Nyang. "Software-Based Remote Code Attestation in Wireless Sensor Network." In: *Proceedings of the Global Communications Conference, (GLOBECOM)*. IEEE, 2009, pp. 1–8.

[5] C. Adjih, E. Baccelli, E. Fleury, G. Harter, N. Mitton, T. Noel, R. Pissard-Gibollet, F. Saint-Marcel, G. Schreiner, J. Vandaele, and T. Watteyne. "FIT IoT-LAB: A large scale open experimental IoT testbed." In: *World Forum on Internet of Things, (WF-IoT)*. IEEE Computer Society, 2015, pp. 459–464.

[6] Roger Alexander, Anders Brandt, JP Vasseur, Jonathan Hui, Kris Pister, Pascal Thubert, P Levis, Rene Struik, Richard Kelsey, and Tim Winter. *RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks*. RFC 6550. 2012. DOI: 10.17487/RFC6550. URL: https://rfc-editor.org/rfc/rfc6550.txt.

[7] Muhammad Naveed Aman, Mohamed Haroon Basheer, Siddhant Dash, Jun Wen Wong, Jia Xu, Hoon Wei Lim, and Biplab Sikdar. "HAtt: Hybrid Remote Attestation for the Internet of Things With High Availability." In: *IEEE Internet Things Journal* 7.8 (2020), pp. 7220–7233.

[8] Moreno Ambrosin, Mauro Conti, Ahmad Ibrahim, Gregory Neven, Ahmad-Reza Sadeghi, and Matthias Schunter. "SANA: Secure and Scalable Aggregate Network Attestation." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2016, pp. 731–742.

[9] Mahmoud Ammar, Bruno Crispo, Bart Jacobs, Danny Hughes, and Wilfried Daniels. "Sµv — The Security MicroVisor: A Formally-Verified Software-Based Security Architecture for the Internet of Things." In: *IEEE Transactions on Dependable and Secure Computing* (2019).

[10] *ARM Cortex-M3 Reference Manual*. https://developer.arm.com/documentation/ddi0337/e.

[11] ARM TrustZone. https://developer.arm.com/technologies/trustzone. 2018.

[12] Frederik Armknecht, Ahmad-Reza Sadeghi, Steffen Schulz, and Christian Wachsmann. "A Security Framework for the Analysis and Design of Software Attestation." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2013, pp. 1–12.

[13] Divya Arora, Srivaths Ravi, Anand Raghunathan, and Niraj K. Jha. "Secure Embedded Processing through Hardware-Assisted Run-Time Monitoring." In: *Design Automation Conference (DAC)*. IEEE Computer Society, 2005, pp. 178–183.

[14] AUTOSAR. *Specification of Watchdog Manager*. https://www.autosar.org/fileadmin/user_upload/standards/classic/4-3/AUTOSAR_SWS_WatchdogManager.pdf.

[15] Paramvir Bahl, Ranveer Chandra, and John Dunagan. "SSCH: Slotted Seeded Channel Hopping for Capacity Improvement in IEEE 802.11 Ad-hoc Wireless Networks." In: *Annual International Conference on Mobile Computing and Networking (MOBICOM)*. ACM, 2004, pp. 216–230.

[16] Sebastian P Bayerl, Tommaso Frassetto, Patrick Jauernig, Korbinian Riedhammer, Ahmad-Reza Sadeghi, Thomas Schneider, Emmanuel Stapf, and Christian Weinert. "Offline Model Guard: Secure and Private ML on Mobile Devices." In: *Design, Automation, and Test in Europe (DATE)* (2020), pp. 460–465.

[17] Marco Benocci, Carlo Tacconi, Elisabetta Farella, Luca Benini, Lorenzo Chiari, and Laura Vanzago. "Accelerometer-based fall detection using optimized ZigBee data streaming." In: *Microelectron. Journal* 41.11 (2010), pp. 703–710.

[18] Liam Bindle and Demilade Adeoye. *MQTT-C*. https://github.com/LiamBindle/MQTT-C. [accessed Sep-2022]. 2022.

[19] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. DOI: 10.17487/RFC7228. URL: https://rfc-editor.org/rfc/rfc7228.txt.

[20] Ferdinand Brasser, Brahim El Mahjoub, Ahmad-Reza Sadeghi, Christian Wachsmann, and Patrick Koeberl. "TyTAN: Tiny Trust Anchor for Tiny Devices." In: *ACM Proceedings of the Annual Design Automation Conference (DAC)*. 2015, p. 34.

[21] Dominik Breitenbacher, Ivan Homoliak, Yan Lin Aung, Nils Ole Tippenhauer, and Yuval Elovici. "HADES-IoT: A Practical Host-Based Anomaly Detection System for IoT Devices." In: *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2019, pp. 479–484.

[22] Fergal Byrne. *Random Distributed Scalar Encoder*. http://fergalbyrne.github.io/rdse.html/. [accessed Sep-2022]. 2014.

[23] Xavier Carpent, Karim El Defrawy, Norrathep Rattanavipanon, and Gene Tsudik. "Lightweight Swarm Attestation: A Tale of Two LISA-s." In: *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017, pp. 86–100.

[24] Xavier Carpent, Karim El Defrawy, Norrathep Rattanavipanon, and Gene Tsudik. "Lightweight Swarm Attestation: A Tale of Two LISA-s." In: *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2017, pp. 86–100.

[25] Claude Castelluccia, Aurélien Francillon, Daniele Perito, and Claudio Soriente. "On the Difficulty of Software-based Attestation of Embedded Devices." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2009, pp. 400–409.

[26] Miguel Castro, Manuel Costa, and Tim Harris. "Securing Software by Enforcing Data-flow Integrity." In: *Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, 2006, pp. 147–160.

[27] Moses Charikar. "Similarity estimation techniques from rounding algorithms." In: *ACM Symposium on Theory of Computing*. Ed. by John H. Reif. ACM, 2002, pp. 380–388.

[28] Long Cheng, Salman Ahmed, Hans Liljestrand, Thomas Nyman, Haipeng Cai, Trent Jaeger, N. Asokan, and Danfeng (Daphne) Yao. "Exploitation Techniques for Data-oriented Attacks with Existing and Potential Defense Approaches." In: *ACM Transactions on Privacy and Security* (2021).

[29] Long Cheng, Ke Tian, and Danfeng Yao. "Orpheus: Enforcing Cyber-Physical Execution Semantics to Defend Against Data-Oriented Attacks." In: *Proceedings of the 33rd Annual Computer Security Applications Conference*. 2017, pp. 315–326.

[30] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation." In: *Conference on Empirical Methods in Natural Language Processing (EMNL)*. ACL, 2014, pp. 1724–1734.

[31] *Corte-M vs. Cortex-A*. https://www.programmersought.com/article/37344867142/. [accessed Jan-2021]. 2015.

[32] Lucas Davi, Alexandra Dmitrienko, Manuel Egele, Thomas Fischer, Thorsten Holz, Ralf Hund, Stefan Nürnberger, and Ahmad-Reza Sadeghi. "MoCFI: A Framework to Mitigate Control-Flow Attacks on Smartphones." In: *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[33] Lucas Davi, Patrick Koeberl, and Ahmad-Reza Sadeghi. "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation." In: *Design Automation Conference (DAC)*. ACM, 2014, 133:1–133:6.

[34] Lucas Vincenzo Davi. "Code-reuse attacks and defenses." PhD thesis. Darmstadt University of Technology, Germany, 2015. URL: http://tuprints.ulb.tu-darmstadt.de/4622/.

[35] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, and Gene Tsudik. "DIALED: Data Integrity Attestation for Low-end Embedded Devices." In: *Design Automation Conference (DAC)*. 2021, pp. 313–318.

[36] ARM Inc. Diya Soubra. *Top level difference in features between Cortex-M33 and Cortex-M4*. https://community.arm.com/developer/ip-products/processors/trustzone-for-armv8-m/f/trustzone-armv8-m-forum/8338/what-is-the-top-level-difference-in-features-between-cortex-m33-and-cortex-m4. [accessed Oct-2021]. 2019.

[37] Min Du, Feifei Li, Guineng Zheng, and Vivek Srikumar. "DeepLog: Anomaly Detection and Diagnosis from System Logs through Deep Learning." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 1285–1298.

[38] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. "Run-time dynamic linking for reprogramming wireless sensor networks." In: *Conference on Embedded Network Sensor Systems (SenSys)*. ACM, 2006, pp. 15–28.

[39] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. "Contiki-A Lightweight and Flexible Operating System for Tiny Networked Sensors." In: *Conference on Local Computer Networks (LCN)*. IEEE Computer Society, 2004, pp. 455–462.

[40] Simon Duquennoy, Atis Elsts, Al Nahas, and George Oikonomou. "TSCH and 6TISCH for Contiki: Challenges, Design and Evaluation." In: *International Conference on Distributed Computing in Sensor Systems, (DCOSS)*. IEEE, 2017, pp. 11–18.

[41] Simon Duquennoy, Beshr Al Nahas, Olaf Landsiedel, and Thomas Watteyne. "Orchestra: Robust Mesh Networks Through Autonomously Scheduled TSCH." In: *Conference on Embedded Network Sensor Systems (SenSys)*. ACM, 2015, pp. 337–350.

[42] M. R. Ebling. "IoT: From Sports to Fashion and Everything In-Between." In: *IEEE Pervasive Computing* (2016). ISSN: 1536-1268.

[43] EE Times and Embedded.com. `https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf`. [accessed Oct-2021]. 2019.

[44] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." In: *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[45] Karim Eldefrawy, Gene Tsudik, Aurélien Francillon, and Daniele Perito. "SMART: Secure and Minimal Architecture for (Establishing Dynamic) Root of Trust." In: *Network and Distributed System Security Symposium (NDSS)*. The Internet Society, 2012.

[46] Joakim Eriksson, Fredrik Ósterlind, Niclas Finne, Nicolas Tsiftes, Adam Dunkels, Thiemo Voigt, Robert Sauter, and Pedro José Marrón. "COOJA/MSPSim: Interoperability Testing for Wireless Sensor Networks." In: *Simutools*. 2009.

[47] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas A. Longstaff. "A Sense of Self for Unix Processes." In: *Symposium on Security and Privacy (S&P)*. IEEE, 1996, pp. 120–128.

[48] Aurélien Francillon and Claude Castelluccia. "Code Injection Attacks on Harvard-architecture Devices." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2008, pp. 15–26.

[49] Aurélien Francillon, Quan Nguyen, Kasper B Rasmussen, and Gene Tsudik. "A Minimalist Approach to Remote Attestation." In: *Design, Automation, and Test in Europe (DATE)*. European Design and Automation Association, 2014, pp. 1–6.

[50] Sheila Frankel, Rob Glenn, and Scott Kelly. "The AES-CBC cipher algorithm and its use with IPsec." In: (2003).

[51] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin E. Lauter, Michael Naehrig, and John Wernsing. "CryptoNets: Applying Neural Networks to Encrypted Data with High Throughput and Accuracy." In: *International Conference on Machine Learning (ICML)*. Vol. 48. JMLR.org, 2016, pp. 201–210.

[52] Pedro Henrique Gomes, Thomas Watteyne, Pradipta Ghosh, and Bhaskar Krishnamachari. "Competition: Reliability through Timeslotted Channel Hopping and Flooding-based Routing." In: *International Conference on Embedded Wireless Systems and Networks (EWSN)*. ACM, 2016, pp. 297–298.

[53] Andy Greenberg. *Hacker tries to poison water supply of Florida city*. https://www.wired.com/story/oldsmar-florida-water-utility-hack/. [accessed Oct-2021]. 2021.

[54] Oliver Hahm, Emmanuel Baccelli, Hauke Petersen, and Nicolas Tsiftes. "Operating Systems for Low-End Devices in the Internet of Things: A Survey." In: *Internet of Things Journal* 3.5 (2016), pp. 720–734.

[55] Monowar Hasan and Sibin Mohan. "Protecting Actuators in Safety-Critical IoT Systems from Control Spoofing Attacks." In: *Workshop on Security and Privacy for the Internet-of-Things, IoT S&P*. ACM, 2019, pp. 8–14.

[56] J. Hawkins, S. Ahmad, S. Purdy, and A. Lavin. "Biological and Machine Intelligence (BAMI)." Initial online release 0.4. 2016. URL: https://numenta.com/resources/biological-and-machine-intelligence/.

[57] Jeff Hawkins, Subutai Ahmad, and Donna Dubinsky. "Hierarchical temporal memory including HTM cortical learning algorithms." In: *Techical report, Numenta, Inc, Palto Alto* (2010).

[58] Hotel's in-room assistants. https://www.zdnet.com/article/hotels-in-room-assistants-could-have-been-used-to-spy-on-guests/. [accessed Oct-2021]. 2019.

[59] Hong Hu, Shweta Shinde, Sendroiu Adrian, Zheng Leong Chua, Prateek Saxena, and Zhenkai Liang. "Data-Oriented Programming: On the Expressiveness of Non-control Data Attacks." In: *Symposium on Security and Privacy (S&P)*. IEEE Computer Society, 2016, pp. 969–986.

[60] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. "US-AID: Unattended Scalable Attestation of IoT Devices." In: *Symposium on Reliable Distributed Systems (SRDS)*. IEEE Computer Society, 2018, pp. 21–30.

[61] Ahmad Ibrahim, Ahmad-Reza Sadeghi, Gene Tsudik, and Shaza Zeitouni. "DARPA: Device Attestation Resilient to Physical Attacks." In: *ACM Conference on Security & Privacy in Wireless and Mobile Networks, WISEC*. 2016, pp. 171–182.

[62] "IEEE Standard for Low-Rate Wireless Networks." In: *IEEE Std 802.15.4-2015 (Revision of IEEE Std 802.15.4-2011)* (2016), pp. 1–709.

[63] Ahmed Imteaj, Urmish Thakker, Shiqiang Wang, Jian Li, and M. Hadi Amini. "A Survey on Federated Learning for Resource-Constrained IoT Devices." In: *IEEE Internet Things Journal* 9.1 (2022), pp. 1–24.

[64] ARM Inc. *ARM Cortex-M23 Processor Technical Reference Manual*. https://developer.arm.com/documentation/ddi0550/c. [accessed Oct-2021]. 2016.

[65] IoT attacks. https://www.gartner.com/newsroom/id/3869181. [accessed March-2020]. 2018.

[66] Don Johnson, Alfred Menezes, and Scott Vanstone. "The Elliptic Curve Digital Signature Algorithm (ECDSA)." In: *International Journal of Information Security, Springer* (2001).

[67] Hadi S. Jomaa, Josif Grabocka, and Lars Schmidt-Thieme. "Hyp-RL : Hyperparameter Optimization by Reinforcement Learning." In: *CoRR* abs/1906.11527 (2019). URL: http://arxiv.org/abs/1906.11527.

[68] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. "GAZELLE: A Low Latency Framework for Secure Neural Network Inference." In: *USENIX Conference on Security Symposium (USENIX Security)*. USENIX Association, 2018, pp. 1651–1668.

[69] Keras. https://keras.io/. [accessed Oct-2021]. 2019.

[70] Sean Tinneny Kevin Tofel Wanlu Ding and Sarah Wagner. *Hacking Diabetes*. https://sarahrwagner.github.io/. [accessed Sep-2022]. 2019.

[71] Florian Kohnhäuser, Niklas Büscher, Sebastian Gabmeyer, and Stefan Katzenbeisser. "SCAPI: A Scalable Attestation Protocol to Detect Software and Physical Attacks." In: *ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 2017, pp. 75–86.

[72] Constantinos Kolias, Georgios Kambourakis, Angelos Stavrou, and Jeffrey Voas. "DDoS in the IoT: Mirai and other Botnets." In: *IEEE Computer,* (2017), pp. 80–84.

[73] Mark Kraay. *Basic Neural Network for MNIST*. https://github.com/markkraay/mnist-from-scratch. [accessed Sep-2022]. 2022.

[74] Boyu Kuang, Anmin Fu, Shui Yu, Guomin Yang, Mang Su, and Yuqing Zhang. "ESDRA: An Efficient and Secure Distributed Remote Attestation Scheme for IoT Swarms." In: *IEEE Internet of Things Journal* (2019).

[75] John Leyden. "Water treatment plant hacked, chemical mix changed for tap supplies." In: *The Register* (2016).

[76] Yanlin Li, Jonathan M McCune, and Adrian Perrig. "SBAP: Software-Based Attestation for Peripherals." In: *Trust and Trustworthy Computing, Third International Conference (TRUST)*. Vol. 6101. Lecture Notes in Computer Science. Springer, 2010, pp. 16–29.

[77] Light Controller. https://github.com/Barro/light-controller. 2016.

[78] ARM Limited. "Embedded Trace Macrocell, ETMv1 to ETMv3.5 Architecture Specification." In: (2011).

[79] Jie Lin, Wei Yu, Nan Zhang, Xinyu Yang, Hanlin Zhang, and Wei Zhao. "A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications." In: *Internet of Things Journal* 4.5 (2017), pp. 1125–1142.

[80] Fang Liu, Xiuzhen Cheng, and Dechang Chen. "Insider Attacker Detection in Wireless Sensor Networks." In: *International Conference on Computer Communications (INFOCOM)*. IEEE, 2007, pp. 1937–1945.

[81] Jian Liu, Mika Juuti, Yao Lu, and N. Asokan. "Oblivious Neural Network Predictions via MiniONN Transformations." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2017, pp. 619–631.

[82] Kais Mekki, Eddy Bajic, Frederic Chaxel, and Fernand Meyer. "A Comparative Study of LPWAN Technologies for Large-scale IoT Deployment." In: *ICT Express* 5.1 (2019), pp. 1–7.

[83] Charlie Miller and Chris Valasesk. *Remote Exploitation of an Unaltered Passenger Vehicle*. http://illmatics.com/Remote%20Car%20Hacking.pdf. [accessed March-2020]. 2015.

[84] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base." In: *USENIX Conference on Security Symposium (USENIX Security)*. USENIX Association, 2013, pp. 479–494.

[85] Job Noorman, Pieter Agten, Wilfried Daniels, Raoul Strackx, Anthony Van Herrewege, Christophe Huygens, Bart Preneel, Ingrid Verbauwhede, and Frank Piessens. "Sancus: Low-cost Trustworthy Extensible Networked Devices with a Zero-software Trusted Computing Base." In: *USENIX Conference on Security Symposium (USENIX Security)*. USENIX Association, 2013, pp. 479–494.

[86] Number of IoT Devices. https://www.vxchnge.com/blog/iot-statistics. [acc. Jan-2021]. 2020.

[87] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. "{VRASED}: A Verified Hardware/-Software Co-Design for Remote Attestation." In: *USENIX Security*. 2019.

[88] Ivan De Oliveira Nunes, Sashidhar Jakkamsetti, Norrathep Rattanavipanon, and Gene Tsudik. "On the TOCTOU Problem in Remote Attestation." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2021, pp. 2921–2936.

[89] Nuvoton. *Nuvoton Launches NuMicro M2351 Series TrustZone Empowered Microcontroller*. https://www.design-reuse.com/news/44745/nuvoton-trustzone-microcontroller-arm-cortex-m23.html. [accessed Oct-2021]. 2018.

[90] M. T. Nyamukuru and K. M. Odame. "Tiny Eats: Eating Detection on a Microcontroller." In: *Workshop on Machine Learning on Edge in Sensor Systems (SenSys-ML)*. IEEE, 2020, pp. 19–23.

[91] *OpenAPS*. https://openaps.org/. [accessed Sep-2022]. 2017.

[92] Fredrik Osterlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. "Cross-Level Sensor Network Simulation with Cooja." In: *Conference on Local Computer Networks (LCN)*. IEEE Computer Society, 2006.

[93] M. Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M. Songhori, Thomas Schneider, and Farinaz Koushanfar. "Chameleon: A Hybrid Secure Computation Framework for Machine Learning Applications." In: *Asia Conference on Computer and Communications Security (AsiaCCS)*. ACM, 2018, pp. 707–721.

[94] Eyal Ronen, Adi Shamir, Achi-Or Weingarten, and Colin O'Flynn. "IoT goes Nuclear: Creating a ZigBee Chain Reaction." In: *IEEE Symposium on Security and Privacy (S&P)*. 2017, pp. 195–212.

[95] R. Sekar, M. Bendre, Dinakar Dhurjati, and Pradeep Bollineni. "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors." In: *Symposium on Security and Privacy (S&P)*. IEEE, 2001, pp. 144–155.

[96] Nordic Semiconductor. *Nordic Semiconductor unveils world's first dual Arm Cortex-M33 processor*. https://www.nordicsemi.com/News/2019/11/Nordic-unveils-worlds-first-dual-Arm-Cortex-M33-processor-wireless-SoC-nRF5340. [accessed Oct-2021]. 2019.

[97] Arvind Seshadri, Mark Luk, Adrian Perrig, Leendert van Doorn, and Pradeep Khosla. "SCUBA: Secure Code Update by Attestation in Sensor Networks." In: *ACM WiSe*. 2006.

[98] Arvind Seshadri, Adrian Perrig, Leendert Van Doorn, and Pradeep Khosla. "SWATT: Software-based Attestation for Embedded Devices." In: *Symposium on Security and Privacy (S&P)*. 2004.

[99] Nasif Bin Shafi, Kashif Ali, and Hossam S Hassanein. "No-reboot and Zero-flash Over-the-air Programming for Wireless Sensor Networks." In: *Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, (SECON)*. IEEE, 2012, pp. 371–379.

[100] Chi-Sheng Shih, Yao-Ting Wang, and Jyun-Jhe Chou. "Multiple-Image Super-Resolution for Networked Extremely Low-Resolution Thermal Sensor Array." In: *Workshop on Machine Learning on Edge in Sensor Systems (SenSys-ML)*. IEEE, 2020, pp. 1–6.

[101] Xiaokui Shu, Danfeng (Daphne) Yao, Naren Ramakrishnan, and Trent Jaeger. "Long-Span Program Behavior Modeling and Attack Detection." In: *Transactions on Privacy and Security* 20.4 (2017), 12:1–12:28.

[102] Signify Annual Sales Report. https://www.signify.com/static/2018/signify-annual-report-2018.pdf. [accessed March-2020].

[103] STMicroelectronics. *X-Cube-AI: AI Expansion pack for STM32CubeMX*. https://www.st.com/en/embedded-software/x-cube-ai.html. [accessed Oct-2021]. 2018.

[104] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. "OAT: Attesting Operation Integrity of Embedded Devices." In: *Symposium on Security and Privacy (S&P)*. IEEE, 2020, pp. 1433–1449.

[105] Hailun Tan, Wen Hu, and Sanjay Jha. "A TPM-enabled Remote Attestation Protocol (TRAP) in Wireless Sensor Networks." In: *Proceedings of the ACM Workshop on Performance Monitoring and Measurement of Heterogeneous Wireless and Wired Networks*. ACM, 2011, pp. 9–16.

[106] Tensorflow. *Tensorflow-lite minimal example*. https://github.com/tensorflow/tensorflow/tree/master/tensorflow/lite/examples/minimal. [accessed Sep-2022]. 2022.

[107] Pascal Thubert, Maria Rita Palattella, and Thomas Engel. "6TiSCH centralized scheduling: When SDN meet IoT." In: *Conference on Standards for Communications and Networking, CSCN*. IEEE, 2015, pp. 42–47.

[108] Bill Toulas. *Medtronic urgently recalls insulin pump controllers over hacking concerns*. https://www.bleepingcomputer.com/news/security/medtronic-urgently-recalls-insulin-pump-controllers-over-hacking-concerns/. [accessed Sep-2022]. 2021.

[109] Florian Tramèr, Fan Zhang, Ari Juels, Michael K. Reiter, and Thomas Ristenpart. "Stealing Machine Learning Models via Prediction APIs." In: *USENIX Conference on Security Symposium (USENIX Security)*. USENIX Association, 2016, pp. 601–618.

[110] S. Unix and U.S. Laboratories. *System V Application Binary Interface Intel 386 Architecture Processor Supplement*. Prentice Hall PTR, 1994. ISBN: 9780131046702. URL: https://books.google.de/books?id=MdSlPQAACAAJ.

[111] David A. Wagner and Paolo Soto. "Mimicry attacks on host-based intrusion detection systems." In: *Conference on Computer and Communications Security (CCS)*. ACM, 2002, pp. 255–264.

[112] Y. Wang, G. Attebury, and B. Ramamurthy. "A Survey of Security issues in Wireless Sensor Networks." In: *IEEE Communications Surveys Tutorials* 8.1-4 (2016), pp. 2–23.

[113] Benjamin Warren. *Activity detection - Run or walk*. https://www.kaggle.com/code/benwarren5020/neural-network-99-cv-accuracy. [accessed Sep-2022]. 2021.

[114] Christina Warrender, Stephanie Forrest, and Barak A. Pearlmutter. "Detecting Intrusions using System Calls: Alternative Data Models." In: *Symposium on Security and Privacy (S&P)*. IEEE, 1999, pp. 133–145.

[115] Haizhi Xu, Wenliang Du, and Steve J. Chapin. "Context Sensitive Anomaly Monitoring of Process Control Flow to Detect Mimicry Attacks and Impossible Paths." In: *Recent Advances in Intrusion Detection (RAID)*. Vol. 3224. Springer, 2004, pp. 21–38.

[116] Kui Xu, Ke Tian, Danfeng Yao, and Barbara G. Ryder. "A Sharper Sense of Self: Probabilistic Reasoning of Program Behaviors for Anomaly Detection with Context Sensitivity." In: *International Conference on Dependable Systems and Networks (DSN)*. IEEE, 2016, pp. 467–478.

[117] Kui Xu, Danfeng (Daphne) Yao, Barbara G. Ryder, and Ke Tian. "Probabilistic Program Modeling for High-Precision Anomaly Classification." In: *Computer Security Foundations Symposium (CSF)*. IEEE, 2015, pp. 497–511.

[118] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek Abdelzaher. "DeepIoT: Compressing Deep Neural Network Structures for Sensing Systems with a Compressor-Critic Framework." In: *Conference on Embedded Network Sensor Systems (SenSys)*. ACM, 2017, 4:1–4:14.

[119] Man-Ki Yoon, Sibin Mohan, Jaesik Choi, Mihai Christodorescu, and Lui Sha. "Learning Execution Contexts from System Call Distribution for Anomaly Detection in Smart Embedded System." In: *International Conference on Internet-of-Things Design and Implementation (IoTDI)*. ACM, 2017, pp. 191–196.

[120] Shaza Zeitouni, Ghada Dessouky, Orlando Arias, Dean Sullivan, Ahmad Ibrahim, Yier Jin, and Ahmad-Reza Sadeghi. "ATRIUM: Runtime Attestation Resilient under Memory Attacks." In: *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2017, pp. 384–391.

[121] Yundong Zhang, Naveen Suda, Liangzhen Lai, and Vikas Chandra. "Hello Edge: Keyword Spotting on Microcontrollers." In: *CoRR* abs/1711.07128 (2017). URL: http://arxiv.org/abs/1711.07128.

[122] Qian Zhou, Hua Dai, Liang Liu, Kai Shi, Jie Chen, and Hong Jiang. "The Final Security Problem in IoT: Don't Count on the Canary!" In: *International Conference on Data Science in Cyberspace, DSC*. IEEE, 2022, pp. 599–604.

GLOSSARIES

AT overall attestation time. 29, 47, 137
CO overall communication overhead. 28, 47, 49, 137

AHB advanced high-performance bus. 74
AUTOSAR automotive open system architecture. 67

BCA behavioural control anomaly. 4, 91, 93, 94, 95, 100, 105, 113, 116, 117, 138

CBA control branch anomaly. 4, 93, 94, 98, 107, 108, 109, 110, 113, 138, 139
CFA control flow anomaly. 4, 93, 94, 99, 107, 108, 109, 110, 113, 138, 139
CFG control flow graph. 58
CFI control flow integrity. 2, 19, 20, 58
CIA control intensity anomaly. 4, 93, 94, 95, 108, 110, 111, 113, 139

DDoS distributed denial-of-service. 17, 28, 40, 53

ECDSA elliptic curve digital signature. 45
ETB embedded trace buffer. 66, 68, 72, 73, 74, 75, 77, 78, 81, 82, 83, 88, 137
ETM embedded trace macrocell. 4, 59, 60, 72, 73, 74, 78, 137

FNR false negative rate. 78, 80, 83, 138
FPR false positive rate. 78, 79, 80, 83, 84, 85, 138, 141

GRU gated recurrent unit. 4, 5, 20, 59, 60, 68, 69, 70, 71, 72, 75, 76, 79, 80, 81, 82, 83, 84, 85, 86, 88, 116, 137, 138

HMM hidden markov model. 18
HTM hierarchical temporal memory. 4, 5, 91, 98, 99, 101, 102, 103, 104, 105, 108, 109, 110, 111, 112, 113, 116, 117, 138, 139
HW hardware-based. 81, 83, 138, 141

IoT Internet of Things. v, 1, 2, 3, 4, 5, 8, 13, 17, 18, 19, 24, 25, 26, 27, 28, 29, 30, 33, 44, 48, 49, 52, 53, 54, 58, 60, 64, 66, 77, 78, 82, 87, 90, 91, 96, 101, 106, 111, 113, 115, 116, 117, 136

LSTM long short term memory. 20, 70, 83, 84, 85

OAS optimistic aggregate signature. 17

PDR packet delivery ratio. 26, 50, 137

PMU performance monitoring unit. 101
PUF physical unclonable functions. 14

RDSE random distributed scalar encoder. 103
RNN recurrent neural network. 70
ROC receiver operating characteristic. 108

SDR sparse distributed representation. 101, 103, 104
SHDE SimHash document encoder. 104
SVM support vector machine. 19
SW software-based. 81, 83, 138, 141

TPM trusted platform module. 13, 17, 25
TSCH Time Slotted Channel Hopping. 31, 32, 44, 45, 49

LIST OF FIGURES

Figure 1	IoT attacks classification	9
Figure 2	Memory corruption attacks classification	10
Figure 3	Memory corruption attacks description. The figure shows how the control-flow and data-flow is altered during various memory corruption attacks.	12
Figure 4	Overview of Remote Attestation	14
Figure 5	Example topology with *leaf nodes*, *source nodes*, *relay nodes* and a trusted *sink node*. The verifier is a trusted external entity communicating with nodes through the sink.	27
Figure 6	(a): A simple 4 node network (1 sink, 1 leaf, and 2 relay nodes), the verifier, and monitoring & control unit. (b): Example of TSCH schedules created by monitoring & control unit for the network shown in a.	31
Figure 7	Overview of of SWARNA-agg. An external process or an administrator triggers attestation. Time proceeds from top to bottom. On receiving the trigger, the verifier initiates Phase I (bottom-up approach) of the protocol by sending a signed attReq, which is broadcast in the network. The attResps are aggregated along the path from leaves to the base station. On receiving aggregated attResps, the verifier executes Phase I verification to identify correct, malicious, and suspected nodes. If there are any suspected nodes, Phase II (recursive top-down approach) is initiated to mark all suspected nodes as either malicious or correct.	38
Figure 8	Protocol/software stack on IoT devices.	44
Figure 9	Analytical vs simulation results for a perfect binary tree with maximum depth 3 for SWARNA-agg.	44

Figure 10 Attestation time for a 30 nodes network on IoT-LAB testbed, with varying numbers of malicious nodes and transmission powers. Networks of depth 2, 3 and 4 were generated with 3, -9 and -17dBm transmission power. A trade-off between x-axis and y-axis for same transmission ranges can be seen at the intersection of the lines. Up to 50%-80% of malicious nodes in the network, SWARNA-agg has lower attestation time 46
Figure 11 Communication overhead for a 30 nodes network in IoT-LAB testbed, with varying numbers of malicious nodes and transmission powers. Networks of depth 2, 3 and 4 were generated with 3, -9 and -17dBm transmission power. A trade-off between x-axis and y-axis can be seen at the intersection of the lines for same transmission ranges. For up to 50%-75% of malicious nodes in the network, SWARNA-agg has lower CO. 47
Figure 12 Attestation time for different network sizes in the Cooja simulator. For each bar, the dotted region represents the time taken by Phase I of SWARNA-agg, and the remaining time is spent on Phase II. In SWARNA-ind varying the number of malicious nodes has an effect on the AT, unlike for SWARNA-agg. 47
Figure 13 PDR of a data collection application for a 22 nodes 3 hop network in the IoT-LAB testbed with 12 source and 9 relay nodes. In a noisy environment, application PDR drops up to 1.4% and drops by 0.4% when noise is minimal during attestation. PDR recovers faster when noise is minimal. 50
Figure 14 IoT device classes from ARM Cortex family in decreasing order (left to right) of their sizes and processing capabilities, and respective application examples. 59
Figure 15 System architecture overview. 67
Figure 16 GRU network architecture used for program anomaly detection. The intensity of the color in the output vector indicates the probability of the occurrence. 69
Figure 17 ARM CoreSight debug architecture (simplified from [78]) showing ETM and ETB interaction with the core processor. 74

Figure 18 The graphs indicate the achieved FNR, FPR, and anomaly detection accuracy for various applications. The vertical line (dashed blue) in each graph shows the threshold values for the application. 78
Figure 19 Detection accuracy of aberrant-path attacks for various applications 79
Figure 20 Static overhead that shows the application code size with SPADE overhead (tracing + anomaly detection), and GRU model size for hardware-based (HW) and software-based (SW) traces. 81
Figure 21 The graph shows the GRU model size when various software modules of the fall detection application are considered for tracing. 'a' - app, 'u' - utilities, 'd': drivers, 'pd': partial code from drivers, 'h' - hardware. 82
Figure 22 The graph shows the effect of 0.3% FPR on the anomaly detector's performance. A false alarm triggers unnecessary reprogramming (the peaks in GRU). The peaks in GRU indicates reprogramming. 84
Figure 23 Static control flow path for the sample application program in Listing 6.1. 93
Figure 24 The figure shows various runtime attacks and the BCAs caused by these attacks. 95
Figure 25 Overview of OPADE. 98
Figure 26 Architecture of HTM used in OPADE for program anomaly detection. 102
Figure 27 Workflow of HTM program anomaly detection. 103
Figure 28 The graph shows the ROC curve indicating the rate of false positives and true positives when the threshold for anomaly detection is varied. (a) shows the ROC curve for CFA and (b) shows the ROC curve for CBA for various applications. 107

Figure 29 For fall detection application, control-oriented, and data-oriented attacks were carried out during program execution. As a result, we see CFA and CBA during the attack. The figure shows actual anomalies which are generated during the attacks and anomalies predicted by OPADE's HTM algorithm at runtime. The figure also shows the true positives where HTM predicts the anomalies correctly when there is an actual anomaly and false positives where HTM marks a benign control flow as an anomaly. 109
Figure 30 For open syringe application, we carry out a data-oriented attack to dispense a large quantity of chemicals during program execution. As a result, we see a CIA during the attack. The figure shows actual anomalies which are expected during the attacks and predicted anomalies by OPADE's HTM algorithm at runtime. The figure also shows the true positives where HTM predicts the anomalies correctly when there is an actual anomaly and false positives where HTM marks a benign loop intensity as an anomaly. 110
Figure 31 The graph shows the code size of various applications with and without OPADE. 112
Figure 32 The graph shows the execution time of various applications with and without OPADE. The runtime overhead includes the overhead introduced by the HTM algorithm and other components of OPADE, which includes tracing and context switching overhead. 112

LIST OF TABLES

Table 1 Generic notation, node states, and standard functions. . . 34
Table 2 Overall communication overhead (CO) and message size (bytes) of swarm remote attestation techniques for a network of n nodes. 49
Table 3 Sample of traces under normal conditions, during a control-oriented attack, aberrant path attack, and mimicry attack. 63
Table 4 Different encoding techniques. 69
Table 5 Software and hardware traces for the code in Listing 5.1. 73
Table 6 Hyperparameters for training neural network. 76
Table 7 Model prediction accuracy 79
Table 8 Runtime overhead of SPADE using software-based (SW) and hardware-based (HW) traces. 83
Table 9 GRU vs LSTM – compares the model size, single runtime inference, and anomaly detection accuracy. In order to achieve 100% accuracy, the false negative and false positive rates are tabulated. Applications with least and highest FPR are tabulated. 84
Table 10 Behavioural control anomalies observed during execution of the program in Listing 6.1. 93
Table 11 Sample of traces under normal conditions, during a control-oriented attack, aberrant path attack, and mimicry attack. 96
Table 12 Code coverage. 107

Part III

APPENDIX

CURRICULUM VITAE

EDUCATIONAL BACKGROUND

2006 - 2009	Bachelor of Engineering in Computer Science from Visveswaraya Technological University, India
2013 - 2016	Master of Science in Distributed Software Systems from TU Darmstadt, Germany
2017 - 2022	Research Assistant and Ph.D candidate in Computer Science department at TU Darmstadt

WORK EXPERIENCE

Seema Kumar worked as a research assistant in Indian Institue of Science, and Singapore University of Technology and Design focusing mainly on wireless sensors networks and communication protocols. Her research in TU Darmstadt was predominantly on program anomaly detection for embedded devices, and securing wireless sensor networks.

B

ERKLÄRUNG LAUT PROMOTIONSORDNUNG

GEMÄSS DER PROMOTIONSORDNUNG DER TU DARMSTADT

§8 Abs. 1 lit. c Promotionsordnung

Ich versichere hiermit, dass die elektronische Version mit der schriftlichen Version übereinstimmt.

§8 Abs. 1 lit. d Promotionsordnung

Ich versichere hiermit, dass zu einem vorherigen Zeitpunkt noch keine Promotion versucht wurde. In diesem Fall sind nähere Angaben über Zeitpunkt, Hochschule, Dissertationsthema und Ergebnis dieses Versuches mitzuteilen.

§9 Abs. 1 Promotionsordnung

Hiermit erkläre ich, die vorgelegte Arbeit zur Erlangung des akademischen Grades "Doktor-Ingenieur (Dr.-Ing.)" mit dem Titel "Detecting Remote Software Attacks on Embedded IoT Devices" selbständig und ausschließlich unter Verwendung der angegebenen Hilfsmittel erstellt zu haben.

§9 Abs. 2 Promotionsordnung

Die vorgelegte Arbeit hat bisher noch nicht zu Prüfungszwecken gedient.

Darmstadt, 06.12.2022

Seema Kumar