Trustworthy Wireless Personal Area Networks

Travis W. Peters

Technical Report TR2020-878

Dartmouth Computer Science

TRUSTWORTHY WIRELESS PERSONAL AREA NETWORKS

A Thesis
Submitted to the Faculty
in partial fulfillment of the requirements for the
degree of
Doctor of Philosophy
in
Computer Science
by
Travis W. Peters
Guarini School of Graduate and Advanced Studies
Dartmouth College
Hanover, New Hampshire
August 2019

Examining Committee:

(chair) David Kotz, Ph.D.

Sean Smith, Ph.D.

Xia Zhou, Ph.D.

José Camacho, Ph.D.

F. Jon Kull, Ph.D.

Dean of the Guarini School of Graduate and Advanced Studies

Abstract

In the *Internet of Things (IoT)*, everyday objects are equipped with the ability to compute and communicate. These *smart things* have invaded the lives of everyday people, being constantly carried or worn on our bodies, and entering into our homes, our healthcare, and beyond. This has given rise to wireless networks of smart, connected, always-on, personal *things* that are constantly around us, and have unfettered access to our most personal data as well as all of the other devices that we own and encounter throughout our day. It should, therefore, come as no surprise that our personal devices and data are frequent targets of ever-present threats. Securing these devices and networks, however, is challenging. In this dissertation, we outline three critical problems in the context of Wireless Personal Area Networks (WPANs) and present our solutions to these problems.

First, I present our Trusted I/O solution (**BASTION-SGX**) for protecting sensitive user data transferred between wirelessly connected (Bluetooth) devices. This work shows how in-transit data can be protected from privileged threats, such as a compromised OS, on commodity systems. I present insights into the Bluetooth architecture, Intel's Software Guard Extensions (SGX), and how a Trusted I/O solution can be engineered on commodity devices equipped with SGX.

Second, I present our work on **Amulet** and how we successfully built a wear-

able health hub that can run multiple health applications, provide strong security properties, and operate on a single charge for weeks or even months at a time. I present the design and evaluation of our highly efficient event-driven programming model, the design of our low-power operating system, and developer tools for profiling ultra-low-power applications at compile time.

Third, I present a new approach (**VIA**) that helps devices at the center of WPANs (e.g., smartphones) to verify the authenticity of interactions with other devices. This work builds on past work in anomaly detection techniques and shows how these techniques can be applied to Bluetooth network traffic. Specifically, we show how to create normality models based on fine- and course-grained insights from network traffic, which can be used to verify the authenticity of future interactions.

To Mary – my best friend and partner in all things

Acknowledgements

While this dissertation represents the culmination of my scientific work up to this point, none of this work would have been possible without the hard work and support of countless others. I will undoubtedly fail to acknowledge everyone deserving of credit throughout this journey; this is, however, my attempt to thank everyone who has had a significant impact on me and this work.

First and foremost, this work would not have happened without the continued support and guidance of my advisor, Professor David Kotz. Though this experience challenged me in ways I never expected, I could not have asked for a better advisor. I greatly appreciate all the mentorship and wisdom you have provided me over the past six years, shared through countless meetings, late night/early morning emails, walks to and from "the branch office," and serving as your TA. Sincerely, thank you, Dave, for demonstrating each and every day what it is to be a scientist and educator with integrity and passion. I also want to thank the other members of my committee, Professors Sean Smith, Xia Zhou, and José Camacho for their eagerness to engage with me on my work and for providing invaluable guidance. Thank you all for being teachers and mentors to me.

I have also been fortunate to work with incredible technical mentors over the years. Most notably, thank you Reshma Lal, Srikanth Varadarajan, and Pradeep

Pappachan. You showed me firsthand that there are incredibly intelligent and hardworking people that are actively designing and building the present and future of computing. My research experience with you all at Intel significantly deepened my passion for computer and network security, as well as applied research.

In addition to excellent faculty and research mentors, I was fortunate to work with an extremely talented and supportive group of students, postdocs, and technical staff during my time at Dartmouth. In particular Tim Pierson supported me every step of the way as a friend, collaborator, and mentor. Thank you, Tim, for always entertaining my crazy ideas, for helping to give me frameworks to work through problems, for providing honest feedback, and for demonstrating what it is to be a friend, a husband, a father, and a scientist. I am also grateful to Ron Peterson for his technical expertise and incredibly expansive knowledge in computing. I would also like to say thank you to a number of other people who provided support, encouragement and friendship along the way: Patrick Proctor, Sougata Sen, Reza Rawassizadeh, Xiaohui Liang, Andres Molina-Markham, Varun Mishra, Taylor Hardin, Shengjie Bi, Aarathi Prasad, Shrirang Mare, Tianlong Yun, Cory Cornelius, George Boateng, Emily Greene, Vijay Kothari, Jason Reeves, and Ira Ray Jenkins.

Over these past years my life has also been enriched by my friends at Dartmouth, in the Upper Valley, and back home. Thank you Chris & Erin Audino, James & Dani Pallardy, Paul & Kristen Coats, Mike & Amber O'leary, and Jake and Danielle Allen. You are all sources of inspiration for my faith, my family, and my work, and I greatly appreciate your friendship.

Finally, I consider myself extremely fortunate to have such a loving and supportive family. Each of you have been instrumental in shaping who I am today, and I couldn't ask for a better family. My mother Patricia Peters, has always been an inspiration to me – mom: thank you for teaching me how to work hard, and for always encouraging my education. My mother and father, Jerene and Steve Rogers,

never ceased to be loving and encouraging throughout this journey – thank you both for loving me and for sharing your daughter with me. My siblings, Jenna, Shelby, David, and Laura, have all been catalysts for personal and relational growth – thank you for challenging me to see beyond myself, for loving me, and for growing with me. And my wife, Mary, has been my rock. I honestly could not have done this without you by my side, Mary. Thank you for all of the love, laughter, food, running, and for putting up with all my late nights and nerdy tangents.

My research was enabled by a large suite of open source software. I am indebted to the open source community, in particular the developers of Linux, BlueZ, Python, Scikit-learn, Pandas, Matplotlib, and Quantum Leaps.

This research results from a research program at the Institute of Security, Technology, and Society at Dartmouth College, supported by the National Science Foundation under award numbers CNS-1329686, CNS-1314281, CNS-1314342, and TC-0910842, and by the Department of Health and Human Services (SHARP program) under award number 90TR0003-01.

Sincerely, thank you all for your unwavering support!

Contents

List of Tables

List of Figures

1

Introduction

In the *Internet of Things (IoT)*, everyday objects are equipped with the ability to compute and communicate. These *smart things* have invaded the lives of everyday people, being carried or worn on our bodies, and entering into our homes, our healthcare, and beyond. This trend has given rise to wireless networks of smart, connected, always-on, personal *things* that are constantly around us, commonly known as Wireless Personal Area Networks (WPANs).[1] Devices within WPANs have unfettered access to our most personal data as well as all of the other devices that we own and encounter throughout our day. It should, therefore, come as no

[1]Indeed, the term WPAN was in use before the term *IoT* was popularized. Today WPAN is sometimes seen as a subset of the umbrella term, IoT.

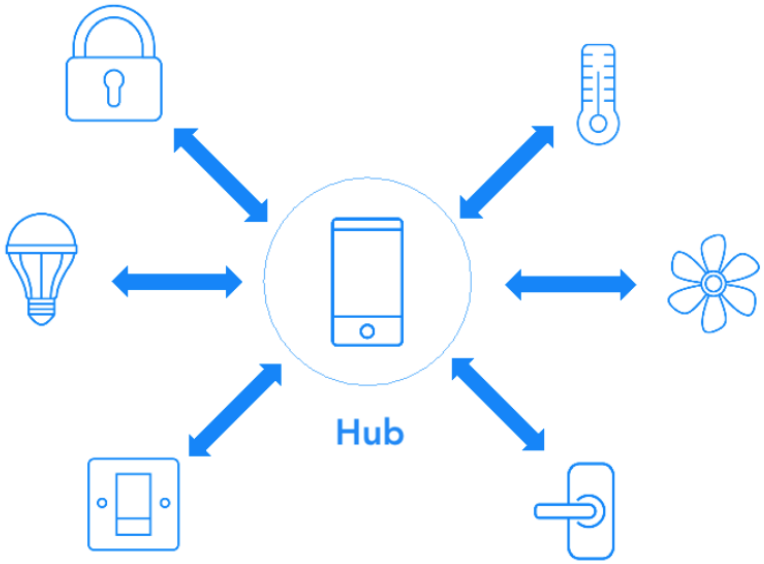

Figure 1.1: An example of a Wireless Personal Area Network, where various IoT devices are connected around a personal hub, such as a smartphone.

surprise that our personal devices and data are frequent targets of ever-present threats. Securing WPANs, however, is challenging.

One specific concern is the fact that *hub devices* (e.g., smartphones, tablets, laptops, and dedicated smart-home "hubs") at the center of WPANs run *applications* (e.g., health applications, finance applications, messaging applications, virtual assistants) that interact with nearby devices and regularly handle sensitive information (Figure 1.1). Threats to these applications and their information (e.g., theft of data, tampering with data) are ever present: they can come from over the Internet, from other software that coexists on the device, or even from other physically-proximate devices.

Past work has largely focused on *remote security*, which aims to protect devices from attacks arriving over the Internet (e.g., attacks exploited over the TCP/IP stack). *Local security*, on the other hand, is concerned with issues such as protecting devices from people and other nearby devices, or protecting applications within a device from malicious software within the device. Local security, unfortunately, is

generally regarded as less of an issue than remote security because localized threats and attacks are expensive and often do not scale in terms of their impact. Indeed, local attacks require that an adversary be physically present to carry out an attack, or at least requires the adversary to compromise another device that will be physically proximate to the target of the attack. However, if an adversary can compromise even one device (be it remotely or locally), they can use that device as a vector to attack other nearby devices. For example, an adversary may compromise a smart light bulb and use it to attack other smart devices in the home; or, an adversary may compromise a smartphone and use it as a vector to attack in-home or on-body devices via local attacks. While these threats were once far-fetched, the rise in the ubiquity and mobility of computers is making them real. Indeed, with the recent interest in IoT, localized threats are possible and becoming more common. Thus, in this dissertation we focus on local security and posit that the attitude of viewing local security as a lesser threat is no longer acceptable.

In this dissertation, we outline three critical problems in the context of WPANs and present our solutions to these problems At its core, *our solutions are largely focused on making WPANs more trustworthy*: re-envisioning how we compose and verify the software, systems, and connections that form the basis for trust in these devices and networks.

1.1 Preliminary Definitions

Throughout this dissertation we specifically focus on WPANs, but use the umbrella term of IoT interchangeably. WPANs are made up of hub devices *(hubs)*, peripheral devices *(peripherals)*, and networks of these devices. Hubs, such as smartphones, tablets, laptops, and smart-home hubs (sometimes referred to as "personal digital assistants"), are so named because of their role as a central device within WPANs.

Hubs generally run end-user applications, and support one or more local networking technologies, such as those that enable Body/Personal Area Networks (via Bluetooth or ZigBee, for example) as well as (Wireless) Local Area Networks (via Wi-Fi, for example).

Peripheral devices, such as mHealth devices, wearables, sensors, actuators, and user-interface devices, are so named because of their ancillary role within WPANs. Some peripheral devices are used for input (e.g., input from an end-user into a hub device) or output (e.g., provide feedback to an end-user from a hub device, administer medication to an end-user per the instruction of an application running on a hub device). Some peripheral devices sense (or actuate upon) the physical world. Peripheral devices generally do not connect directly to the Internet, but rather, connect with a hub device (which may in turn enable the peripheral device to interact with applications on the hub device or over the Internet).

Networks of these hub and peripheral devices are what we refer to as WPANs. Personal Area Networks (PANs) need not be wireless networks, but today, many are (i.e., many PANs are WPANs). Body Area Networks (BAN) and Personal Area Networks (PAN) are all common terms used to refer to networks that connect all devices worn (or carried) on or near the body (or even *within* the body!); these are often WPANs. Body Area Health Networks (BAHN) are a type of BAN that is specifically made up of health and wellness devices. Home Area Networks (HAN) are a kind of Local Area Network (LAN) that connect all devices within the home (PCs, smartphones, televisions, etc.); these could be WPANs.

We state all of this for clarity as there are many ways of defining these networks. All of these networks and ones like it are relevant to our work. Generally speaking, our work aims to address the threats and challenges that come along with everyday end-users operating WPANs.

1.2 The Insecurity of WPANs

Both hub and peripheral devices are commonly used by end-users that have little or no technical insight into how the devices operate, yet they may have significant dependence on their presence and the integrity of their operation. For example, consider a patient that relies on these devices to manage a health condition and interact with her clinician. Thus, throughout this work, we hold that it is vital that security solutions in this context must consider their impact on factors that ultimately impact the end-user; this includes factors such as performance, energy consumption, usability, and compatibility (with respect to the technologies that commonly make up these devices, such as commodity operating systems, network stacks, and the applications themselves). To understand this better, and to illustrate some of the insecurities that our work addresses, let us consider the following scenario.

1.3 Scenario: IoT-based Patient Health Monitoring

Bob is an elderly patient that is clinically obese, has high systolic blood pressure, and was recently diagnosed with type 2 diabetes. To help Bob successfully monitor and manage his health, his primary care physician has prescribed a strict diet and activity regimen, blood-pressure medication, as well as a suite of health devices. The devices are intended to provide Bob meaningful feedback each day, monitor his adherence to the prescribed plan, and enable Bob and his physician to interact regularly between in-person visits. The suite of health devices includes a continuous glucose monitor that will regularly measure Bob's blood sugar levels, a head-mounted eating device that will assist Bob in tracking his food and beverage consumption, and a wrist-worn health device that runs multiple mobile health (mHealth) applications – some

of which interact with the other prescribed health devices.

The wrist-worn device is a critical device for effective treatment as it is intended to be an always-present device in situations where Bob is not near his home computer or smartphone. This device has built-in sensors and wirelessly communicates with the other health devices to aggregate data, which is offloaded to companion applications on Bob's laptop or smartphone when they are nearby. The companion applications communicate with remote services to upload Bob's relevant data as well as render feedback for Bob.

At least three problems arise in scenarios like these, which pose significant threats to Bob, his devices, and his data. We describe these problems briefly next.

Problem #1: Malware and Threats to Sensitive I/O Data

Data from the various health devices is continuously aggregated in the wrist-worn health device and offloaded to Bob's smartphone or laptop whenever possible. To ensure Bob's privacy, all of the health devices employ over-the-air encryption to secure sensitive data in transit between devices. Unfortunately – unbeknownst to Bob – Bob's smartphone was recently the victim of a drive-by-download attack that successfully installed a *privileged malware* – undesirable software with the capability of accessing resources (e.g., protected files, network interfaces) that are generally restricted to privileged entities only; the malware has been surreptitiously siphoning-off any data in and out of the smartphone's Bluetooth network interface.

The reality today is that communication is protected in transit *between* the devices (using over-the-air encryption, for example), but not necessarily in transit *within* the devices (between the intended companion application(s) and the network interface, for example). This lack of protection within the devices leaves internal I/O vulnerable to theft and tampering. Figure 1.2 provides an overview of the components within a hub that pose a potential threat to sensitive I/O data.

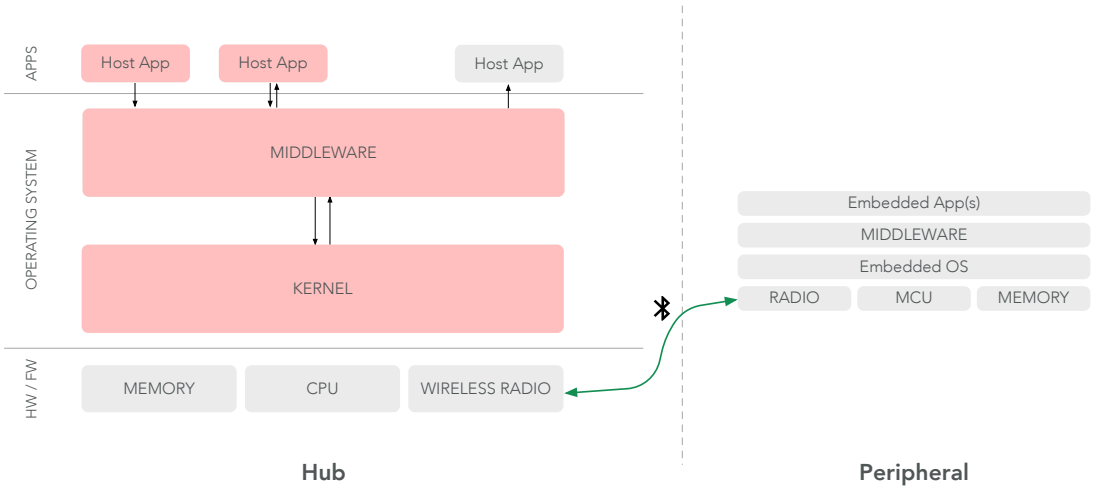

Figure 1.2: An overview of our system model and the threats to sensitive I/O data hub devices. Components within a hub (left) that pose a potential threat to sensitive I/O data are colored red.

Problem #2: The Vulnerability of Peripheral Devices

The wrist-worn health device and other peripheral devices like it run a variety of software, including one or more applications. For instance, the wrist-worn health device in our scenario runs multiple mHealth applications, including an activity monitoring application, a food and beverage consumption tracking application, and a continuous glucose monitoring application; this device also runs software that was not specifically prescribed to Bob for his healthcare treatment, such as an alarm clock application and a sleep monitoring application. While the mHealth applications were developed by reputable companies, the alarm clock application that Bob downloaded from an App Store contained malware.[2] Once installed, the malware exploited a buffer-overflow vulnerability in the system and obtained the ability to execute arbitrary code. The malware has since used its access to the system to steal information from other applications on the device, and to launch attacks against Bob's smartphone (in an attempt to compromise smartphone applications and

[2]In this case, Bob knowingly downloaded the application (though he did not know that it contained malware). While this is one way that malware finds its way into devices, it is by no means the only way.

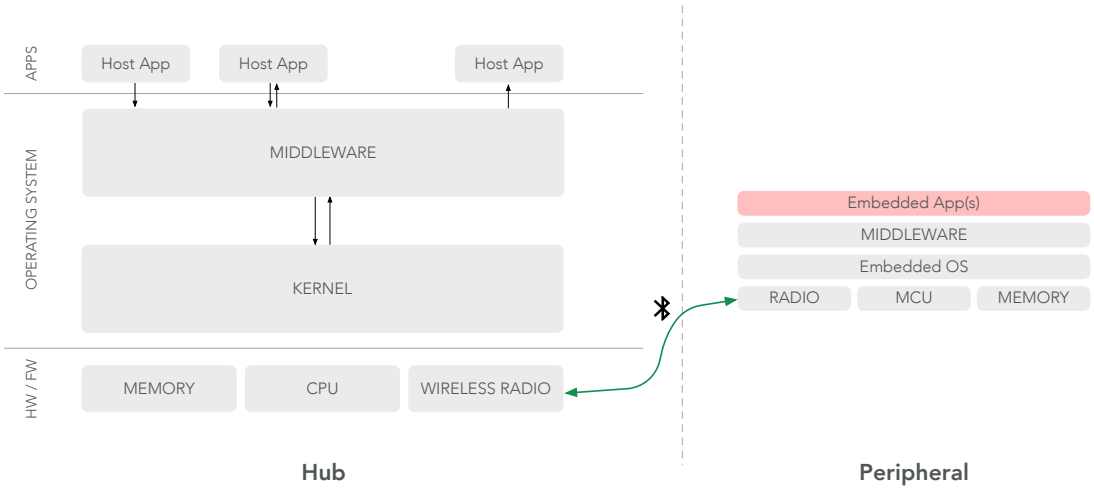

Figure 1.3: An overview of the threats to apps and data on peripheral devices. Components within a peripheral (right) that pose a potential threat to apps and data are colored red.

obtain access to the Internet). Figure 1.3 provides an overview of the components within a peripheral that pose a potential threat to apps and data.

The reality today is that malware and other threats can find their way into peripheral devices through various means. And because of the nature of peripheral devices – which are often small, battery-powered devices, operating on low-cost and functionally-limited hardware – they often lack traditional resources that protect against these threats. This lack of security in peripheral devices makes them ideal targets to adversaries seeking to steal or tamper with data, to commandeer system resources, or to obtain control over the device in order to carry out further attacks on other nearby devices.

Problem #3: The Absence of (Re-)Verification of Trust Relationships

Even if the latter two problems did not exist, and even if hub and peripheral devices employed a variety of state-of-the-art security and privacy solutions to protect against known threats, one unfortunate reality remains: at some point the adversary will be successful in compromising the system or network. For example, at some point an attacker successfully installs malware on one of Bob's devices. Or, at some

point an attacker observes the identifiers of one of Bob's devices and uses them to masquerade as that device in order to establish a connection with his smartphone, which can be used to carry out a subsequent attack on that device.

There exist efforts to address this reality today. For instance, antivirus software installed on a device periodically scans file systems in search of patterns ("signatures") matching known malware. Alternatively (or possibly simultaneously), software agents can run in hub devices or in Wi-Fi access points or in network switches that aggressively monitor network traffic to detect abnormal network activity. Unfortunately, today's solutions face a number of challenges and have been shown to be insufficient by themselves – we list a few of these challenges here. First and foremost, it is not necessarily possible to install software agents or antivirus software on each device, especially peripheral devices, rendering today's solutions inapplicable in our patient monitoring scenario (and scenarios like it). Second, most antivirus solutions today are rule-based or signature-based,[3] and therefore can only detect *known* malware (i.e., malware with a known signature). Unsurprisingly, malware developers have been quite successful in adding slight variations to malware, which changes its signature and enables the slightly-modified malware to evade detection. Third, antivirus software that scans file systems inherently assumes that malware is at some point written to persistent memory such as the hard drive. Recently, however, there have been discoveries of malware that resides only in non-persistent memory, such as RAM or caches, and employs techniques to avoid ever being written to persistent memory. Fourth, even if existing solutions were to scan both persistent memory (e.g., hard drives) and non-persistent memory (e.g, RAM, caches), it is possible that the contents of that memory are encrypted by software or platform security, preventing antivirus software from performing its

[3]While there are other forms of solutions, such as anomaly-based detectors, they are less common. In large networks – where these solutions are often deployed – it is important to avoid false-positives so that short-staffed security teams can focus their attention on real threats. Thus, signature-based systems seem to be the preferred approach, even today.

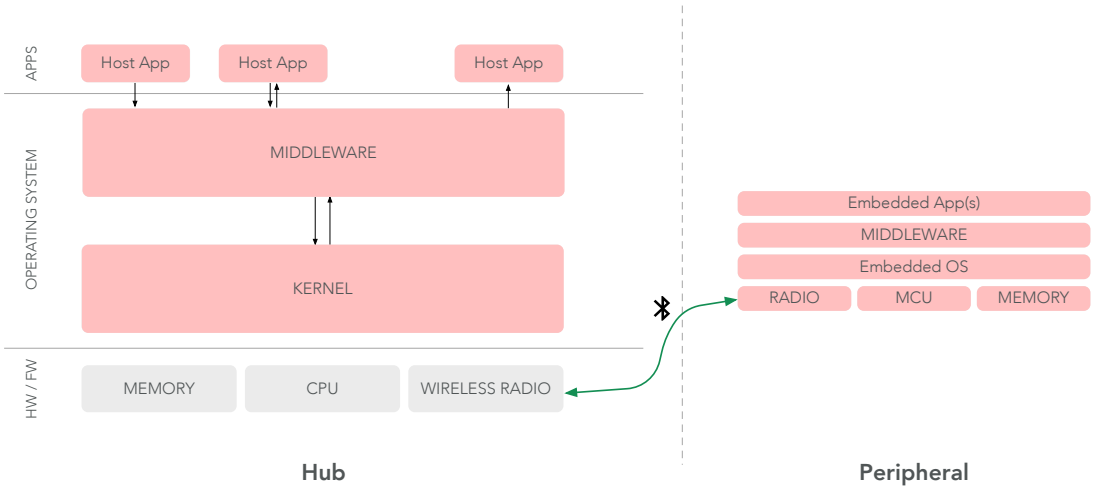

Figure 1.4: An overview of the components within hub and peripheral devices that pose a potential threat to the trustworthiness of Bob's WPAN are colored red.

inspection. Last, not all malicious network traffic is IP-based network traffic. In fact, many communications in the IoT are peer-to-peer, using Bluetooth/BLE, ZigBee, Z-Wave, or others.

These observations amount to an unfortunate reality: there are currently no mechanisms for Bob or his physician to actively detect (or protect against) a compromise within Bob's WPAN. In our scenario, this amounts to Bob and his physician never knowing if there has been a breach within Bob's personal devices. *Has Bob's privacy been violated? Is the data obtained from the devices trustworthy?* Figure 1.4 provides an overview of the components within hub and peripheral devices that pose a potential threat to the trustworthiness of Bob's WPAN. Namely, threats to Bob's WPAN could come from the hub device (e.g., a malicious application attempting to spread and compromise nearby devices), or from peripheral devices (e.g., a device masquerading as another device in an attempt to obtain access to Bob's data). Today, there is no standard mechanism to detect this threat in WPANs.

1.4 Our Vision: Making WPANs More Trustworthy

The trustworthy WPANs we envision are hardened against attacks that target personal devices and sensitive user data. Furthermore, WPANs employ mechanisms to continuously (re-)verify that past trust decisions (e.g., Bob connects and pairs two new devices, $\mathcal{A}$ and $\mathcal{B}$) remain valid over time.

Problem #1 – malware and threats to sensitive I/O data – is addressed by our work, **BASTION-SGX** [156, 184]. BASTION-SGX is a security architecture for protecting applications on hub devices from attacks *within* the device, such as a compromised operating system, drivers, or other variants of malicious software. This work aims to make I/O more trustworthy between peripheral devices and apps running on the hub. We present BASTION-SGX in Chapter 2.

Problem #2 – the vulnerability of peripheral devices – is addressed by our work, **Amulet** [91, 139]. Amulet is a novel platform, which enables developers to create secure and efficient mHealth applications on resource-constrained peripheral devices, such as health and wellness wearables; our work on the Amulet Platform focuses on how these devices and their software can be composed and verified in a more trustworthy way, fortifying them from being compromised by errant applications that might attempt to interfere with other applications or the underlying system itself. We present Amulet in Chapter 3.

Problem #3 – the absence of (re-)verification of trust relationships – is addressed by our most recent work, **Verification of Interaction Authenticity (VIA)**. VIA is an extension of techniques commonly used in anomaly-detection and intrusion-detection systems, which enable authentic interactions between apps and devices to be modeled. The models we produce can be used to verify that future interactions remain consistent with the previously-learned models. Our approach has yielded promising early results, suggesting that VIA can be used to

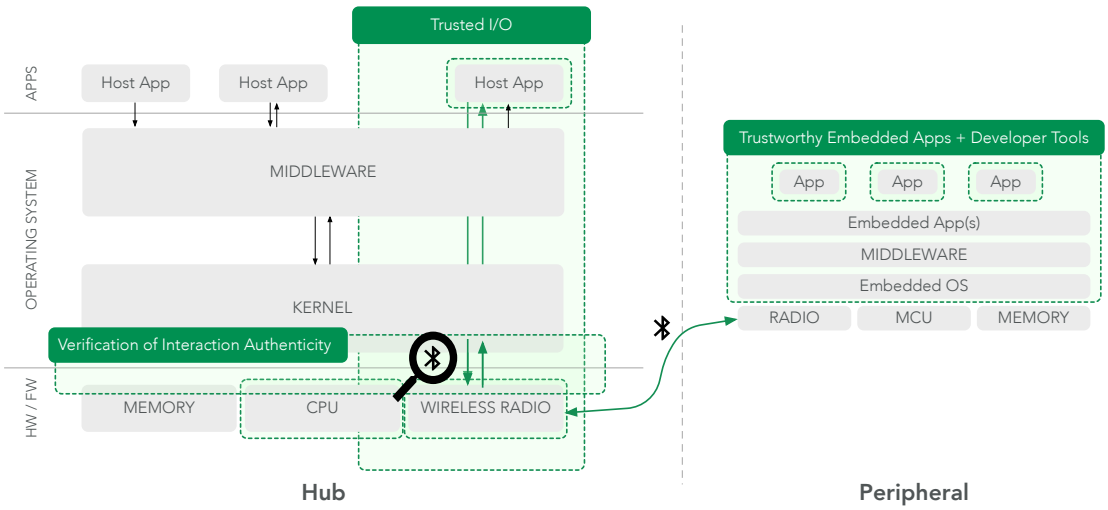

Figure 1.5: Overview of the system model, network model, and contributions presented in this dissertation.

mitigate threats where an adversary attempts to carry out an attack that deviates from the authentic interactions that were previously observed (e.g., when a trust relationship was first established). We present VIA in Chapter 4.

We see our solutions working together in complementary ways (illustrated in Figure 1.5). Specifically, BASTION-SGX (Chapter 2) is a solution that offers more trustworthy I/O between peripheral devices and apps on hubs. Amulet (Chapter 3) is a solution that offers insights into how to design and deploy more trustworthy peripheral devices. And VIA (Chapter 4) is a solution that offers ongoing verification that interactions between apps and devices with a WPAN are authentic, and therefore more trustworthy.

1.5 Contributions

This dissertation makes significant contributions towards the analysis of threats to WPANs, as well as the design and development of tools and system enhancements for protecting apps, securing data channels, and identifying suspicious behaviors within these networks (Figure 1.5).

In aggregate, we make the following contributions in this dissertation.

Establishing Trustworthy Channels for App-Device Communications

In our work on BASTION-SGX [156, 184], we introduce an architecture and methodology for achieving secure and trustworthy I/O on platforms with SGX-enabled processors. We present BASTION-SGX in Chapter 2 and make the following contributions:

1. We identify and solve several challenges in realizing Trusted I/O for Bluetooth on SGX-enabled platforms.
2. We define and present a new Trusted I/O architecture.
3. We present an analytical evaluation of the performance impact of Trusted I/O.
4. We present a prototype and a case study that demonstrates how our solution effectively protects sensitive Bluetooth I/O data from privileged malware.

Designing Trustworthy Peripheral Devices

In our work on Amulet [91, 139], we present a secure, low-power platform and suite of tools for developing and deploying mobile health applications. We present Amulet in Chapter 3 and make the following contributions:[4]

1. We define and present the design and implementation of Amulet's software stack and runtime system.
2. We define and present the design and implementation of Amulet's firmware-production toolchain that guarantees application isolation.

[4]The Amulet project was made possible through the efforts of many researchers at Dartmouth College and Clemson University. Here I emphasize my specific contributions to the project.

3. We define and present the design and implementation of resource models that are deployed in a graphical developer tools that aid developers in developing secure and efficient applications.

4. We present an experimental evaluation of Amulet.

Along with the broader amulet hardware and software, my contributions have been publicly released and are freely available.[5]

Verifying Trustworthy Behavior

In our work on VIA, we introduce a novel approach for ongoing verification of authentic interactions between apps and devices in WPANs. We present VIA in Chapter 4 and make the following contributions:

- We assembled a testbed made up of two distinct device categories (smart health and smart home devices) consisting of 9 different device type, and 20 devices in total. From this testbed, we produced a novel dataset of more than 300 Bluetooth HCI network traces. This new dataset will be made publicly available when this dissertation is published.

- We contribute extensions to open-source Bluetooth-analysis software to enhance the available tools for practical exploration of the Bluetooth protocol and Bluetooth-based apps.

- We present a novel modeling technique (*hierarchical segmentation*), coupled with n-gram models to reliably characterize and verify app-device interactions.

- We present an experimental evaluation of our work using the 20 off-the-shelf devices from our testbed.

[5] https://github.com/AmuletGroup/amulet-project

1.6 Bibliographic Attributions

A majority of the material presented in this dissertation is adapted from previous publications.

- Parts of Chapter 2 appeared in the Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP'18) [156], in conjunction with the 45^{th} International Symposium on Computer Architecture (ISCA'18) – the premier forum for new ideas and research results in computer architecture. Parts of this chapter are also motivated by our patent: System, apparatus and method for providing trusted input/output communications [184]. This paper and patent introduced an architecture and methodology for achieving secure and trustworthy I/O on platforms with SGX-enabled processors.

- Parts of Chapter 3 appeared in the Proceedings of the 14^{th} Conference on Embedded Network Sensor Systems (SenSys'16) [91]. An earlier subset of this work also appeared in the Proceedings of the Workshop on Mobile Medical Applications – Design and Development (WMMADD'14) [139]. These papers presented a secure, low-power platform and suite of tools for developing and deploying mobile health applications.

2

Establishing Trustworthy Channels for App-Device Communications

In Chapter 1 we introduced the idea that there exist threats (e.g., malware) to sensitive I/O data within hub devices. In this chapter, we present BASTION-SGX (Figure 2.1), a novel security architecture for protecting applications on hub devices from attacks *within* the device, such as those that may come from a compromised operating system, drivers, or other variants of malicious software (henceforth malware). While many of the ideas and solutions in this chapter are relevant to all hub devices, the following chapter describes work that is grounded in Intel-based

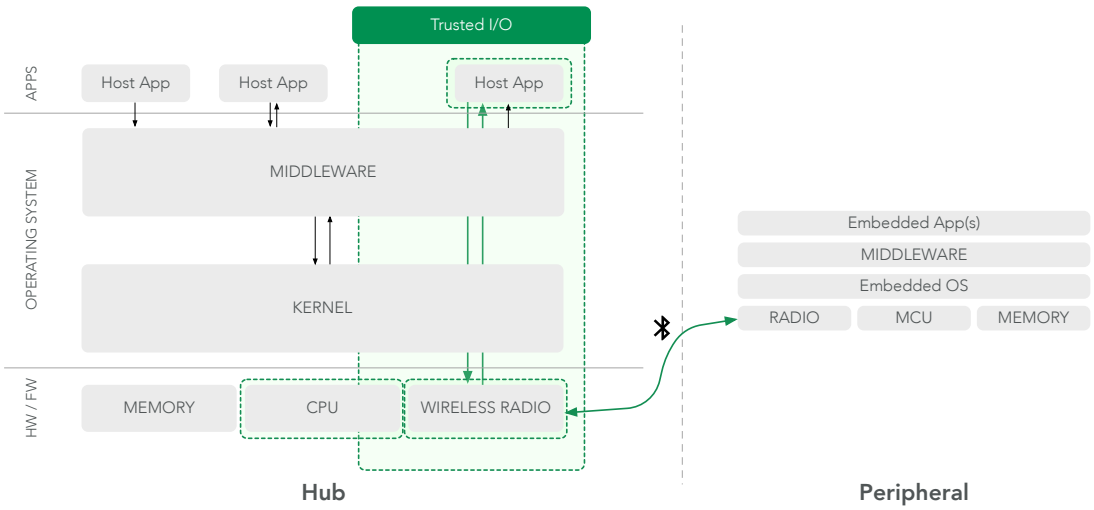

Figure 2.1: Overview of System Model and BASTION-SGX.

devices – specifically those that have CPUs with Intel's Software Guard Extensions (SGX) (see Section 2.2.2 for a brief overview). Moreover, this chapter discusses I/O security with respect to Bluetooth I/O data. Again, many of the ideas and solutions discussed herein are relevant to other I/O technologies; however, this text is grounded in discussion of Bluetooth to demonstrate its value when applied in practice.

This work draws on work that I did as a Security Research intern at Intel Labs in 2015 and 2016. This chapter is a modified version of our paper, *BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX* [156], which was published in June of 2018, and our patent *System, apparatus and method for providing trusted input/output communications* [184], which was published in August of 2019.

2.1 Introduction

Recently, Trusted Execution Environments (TEEs) have generated considerable interest as a means to protect application *code* and *data* from unauthorized access. TEEs are especially promising in light of the ubiquity of malware that threatens to compromise applications and steal or modify security- and privacy-sensitive data

such as personally identifiable information (PII), passwords, credit card numbers, and health data. For example, Intel's Software Guard Extensions (SGX) [100] has been shown to be effective in protecting password databases [43] and even arbitrary containerized processes in Docker [22] from such malware attacks.

For many sensitive applications, input/output (I/O) data has the same level of sensitivity as the data protected inside a TEE-protected application, creating a need to protect I/O data against theft or tampering from a malicious actor. Therefore, many applications that need to use a TEE also need a mechanism to protect user I/O data; this is especially so where sensitive user data is frequently communicated between hub devices and peripheral devices.

To address this need, past work has proposed to construct trusted paths: secure channels between applications and a user's I/O devices. A common approach to construct such a path is to introduce a combination of trusted drivers, middleware, operating systems (OSes), virtual machines (VMs), and hypervisors. As a result, the application's security and the security of its I/O is reduced to the security of the drivers, middleware, OS, VM, and hypervisor, which currently have no way to attest itself to the TEE/TEE-protected application. In contrast, in our work we develop lightweight extensions to the platform itself and describe how those extensions can be attested.

To better understand this issue and the improvement that our solution makes, we must first formalize the *trusted path problem* and discuss the shortcomings of past work.

2.1.1 The Trusted Path Problem

In the canonical trusted-path problem, an application within the hub device (for example, *App1* in Figure 2.2) wants to send/receive data securely to/from peripheral devices outside the hub device, such as Human Interface Devices (HID) and

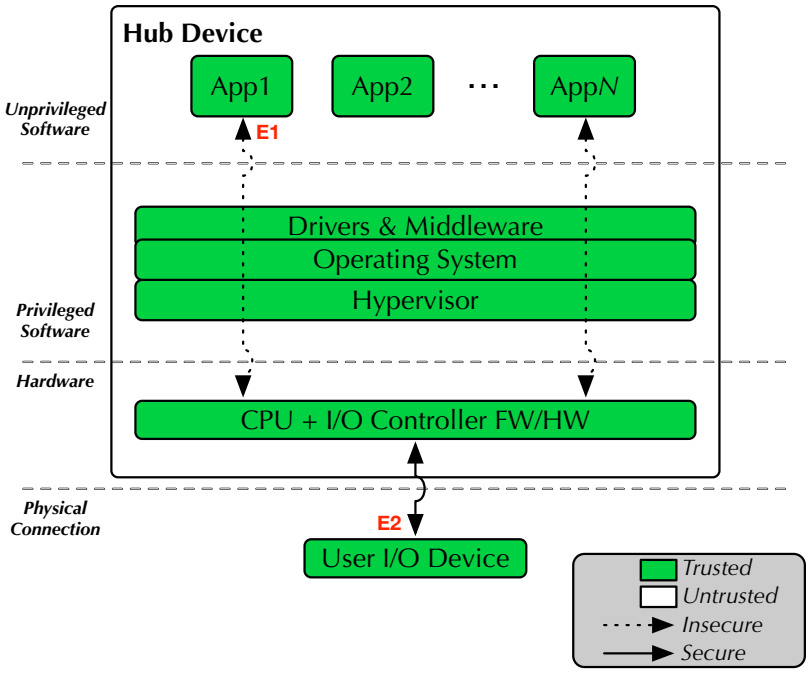

Figure 2.2: Today's solution to the trusted-path problem is to assume that apps, drivers, middleware, OSes, VMs, hypervisors, and hardware are capable and trustworthy with respect to handling sensitive I/O data.

sensors (see the *User I/O Device* in Figure 2.2). More precisely, there exists a collection of applications (apps) that run on a hub device (hub). On the hub is one or more apps that handle sensitive data (e.g., banking app, health app, messaging app). These apps want to securely communicate with peripheral devices (e.g., keyboard, microphone, speakers) connected with the hub, even in light of software-based adversaries (malware) that may have compromised drivers, OSes, VMs, or hypervisors.

Today's solution (Figure 2.2) assumes that any apps, drivers, middleware, OSes, VMs, hypervisors, and hardware are capable and trustworthy with respect to handling sensitive I/O data. Not all of these components of a system are equally worthy of user trust, however.

Figure 2.3 illustrates the general idea behind early research towards addressing the trusted-path problem (e.g., [189, 200]). Specifically, given a hub with TEE

technology (e.g., SGX), TEEs are used as a mechanism to protect the execution of an app (parts of its code and data) by partitioning it into trusted and untrusted parts; this leads to the notion of a *trusted app* [96], which is distinct from other user apps. A combination of new, trusted components – trusted drivers, middleware, OSes, VMs, and hypervisors – are then used to construct trusted paths between the trusted app (*TApp* in Figure 2.3) and user I/O device through untrusted software. For instance, a hypervisor-based solution [200] will run a trusted VM with an untrusted commodity OS (e.g., Windows, Linux). The hypervisor and VM are relied upon to contain the untrusted OS, preventing it from accessing I/O devices or memory regions (memory that belongs to a trusted app, for example) that are meant to be protected. All I/O goes from the hypervisor to the trusted VM, where it is encrypted and sent to the trusted app that runs within the untrusted OS. This approach, however, is not effective for SGX-enabled apps since the software outside of its TEE does not have a way to attest itself to the SGX app. Furthermore, these approaches include complex and error-prone software within the Trusted Computing Base (TCB). As a result, the security of the trusted app is reduced to the security of the (many) trusted components that are relied upon in past solutions – components that are regularly exploited.

This leads to Figure 2.4, which depicts our solution. Rather than introducing non-standard drivers or hypervisors, as past work does, we introduce lightweight extensions to the platform itself. Specifically, we developed new platform features to equip SGX-enabled platforms with Trusted I/O capabilities for specific I/O paths. These features are primarily concerned with ensuring I/O protection from all software adversaries, including privileged software, such as the OS. By enabling Trusted I/O in the platform – removing all system software from the TCB – we significantly enhance I/O security.

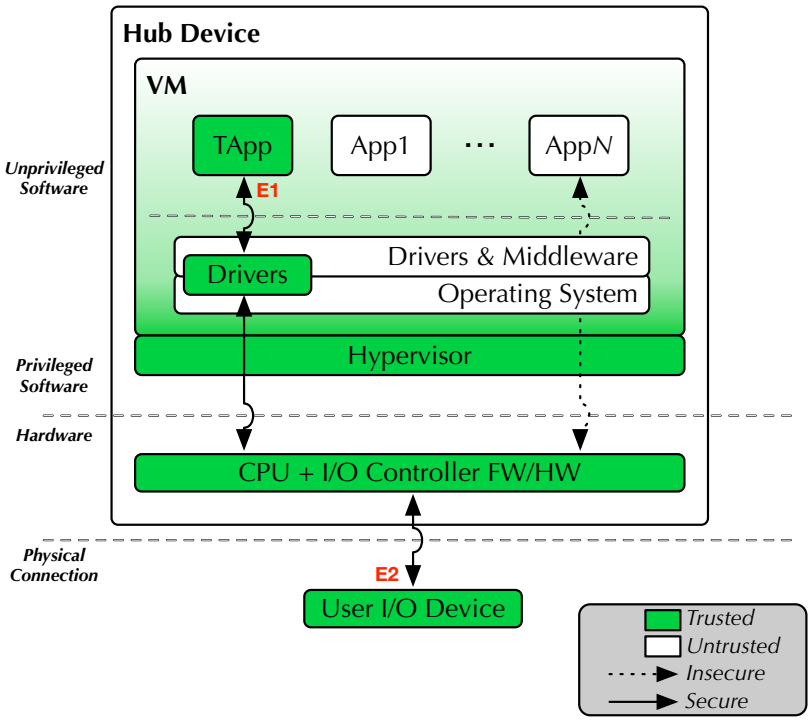

Figure 2.3: Variations of proposed solutions to the trusted-path problem rely on a combination of trusted drivers, VMs, and hypervisors for I/O security. For example, past work [189] describes how secure app-to-driver binding and driver-to-device binding can be achieved via trusted VMs and hypervisors.

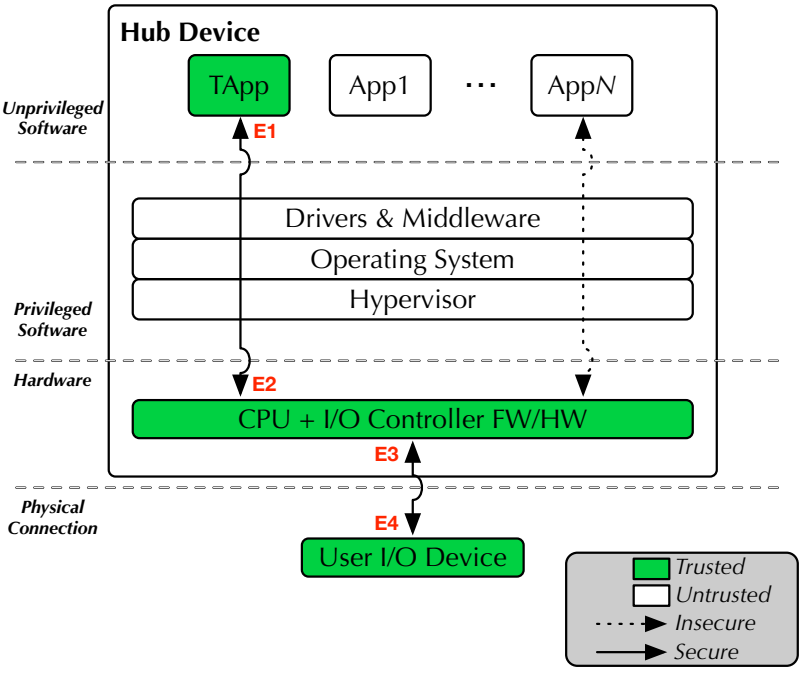

Figure 2.4: Our solution to the trusted-path problem is to implement lightweight features in I/O hardware, securing the path between the TEE and a specific I/O interface.

2.1.2 Scope: Securing Bluetooth I/O on SGX Platforms

In our solution we consider the trusted-path problem for a specific wireless I/O technology – Bluetooth – using a specific TEE technology – SGX. Also, we specifically consider the challenges around securing only the sensitive user data sent to/from Bluetooth devices (e.g., keyboard key presses, health sensor data) on SGX-enabled platforms. Thus, all references to I/O security throughout this chapter are specifically aimed at securing user data over Bluetooth. While we focus our attention on Bluetooth and SGX, we postulate that similar notions apply to other wired (e.g., USB), wireless (e.g., ZigBee, NFC, Wi-Fi) and TEE (e.g., ARM TrustZone [19], AMD Secure Technology [7]) technologies.

Furthermore, in our solution we consider adding security only where it is needed, but does not yet exist. Namely, today, wireless I/O technologies – including Bluetooth – already provide I/O protection at the hardware link level (i.e., between the hub and peripheral device), but leave the hub-side security (i.e., within the hub) up to the OS. All I/O between the hub and its connected devices is already secured with existing security, such as Over-The-Air (OTA) encryption; all I/O *within* the hub, however, is transferred in plaintext between apps and the hub's I/O hardware (e.g., Bluetooth Controller). Our work addresses this hub-side insecurity by securing I/O data between the I/O hardware and trusted apps.

2.1.3 Challenges in Bluetooth Trusted I/O

Today, hub devices run a vast number of apps and connect with a multitude of Bluetooth devices that enable human users to interact with these apps. It is imperative that the hub be capable of reliably managing multiple connections with other Bluetooth devices. Therefore, in light of the trusted-path problem, we need a Trusted I/O mechanism that can selectively secure only user data, and only between

trusted apps and designated devices, while allowing all other apps and devices to communicate as usual; this ensures all other apps can continue to work without modification. We discuss this fundamental challenge and how we overcome it in greater detail in Section 2.4.

2.1.4 Contributions

The contributions of this work are:

1. We identify and solve several challenges in realizing Trusted I/O for Bluetooth. Namely, we describe an approach to connection monitoring that can be applied within a hub's Bluetooth Controller, allowing it to unobtrusively collect Bluetooth device and channel metadata.

2. We present BASTION-SGX, an architecture for realizing Trusted I/O specifically for Bluetooth on SGX-enabled platforms. Our architecture is grounded in lightweight extensions to existing Bluetooth I/O firmware to enable Bluetooth Trusted I/O.

3. We present a prototype of our work in a case study that secures sensitive Bluetooth I/O data between Human Interface Devices (keyboard/mouse) and a trusted app. Specifically, our work shows how it is possible to extend existing over-the-air Bluetooth security all the way to a trusted app with our new security features that secure Bluetooth I/O between the Bluetooth Controller and the trusted app.

We emphasize that all of our contributions significantly enhance I/O security relative to today's solution by protecting user I/O data from software-based adversaries on the hub. Furthermore, our solution only requires modifications to the firmware of the hub's Bluetooth Controller and the trusted apps.

2.2 Background

In this section we present background information on the prominent technologies featured in this chapter: Bluetooth and Intel's SGX.

2.2.1 Bluetooth

A detailed description of the Bluetooth architecture and its protocols are beyond the scope this work. For this, we refer the reader to the Bluetooth Core Specification [35] (more than 2,500 pages!). A basic understanding, however, is required to appreciate the challenges and solutions we discuss. Note that this chapter describes specific insights into our work with Bluetooth Classic. The main ideas carry over to Bluetooth Low Energy (BLE) as well since the the hierarchy of links and channels (defined below) are arranged similarly in both Bluetooth Classic and BLE. Therefore, we make references throughout this text to *Bluetooth*, with the understanding that it applies to both Bluetooth Classic and BLE.

A typical deployment of Bluetooth (within a hub) consists of a Host and one or more Controllers. The Host Controller Interface (HCI) is a command interface between the Host and Controllers. A Bluetooth host is a logical entity made up of all the layers between Bluetooth's core profiles (i.e., Bluetooth applications and services) and the HCI. A Bluetooth controller is a logical entity made up of all of the layers below the HCI, and enables the client to communicate with other Bluetooth devices. In most hub devices, the Host is implemented in software (within an OS such as Linux, for example), whereas the Controller is implemented with a combination of firmware and hardware. The Bluetooth specification identifies several of possible configurations of Hosts and Controllers, but an understanding of this representative deployment is sufficient for the reader. A simplified view of this deployment is shown in Figure 2.5.

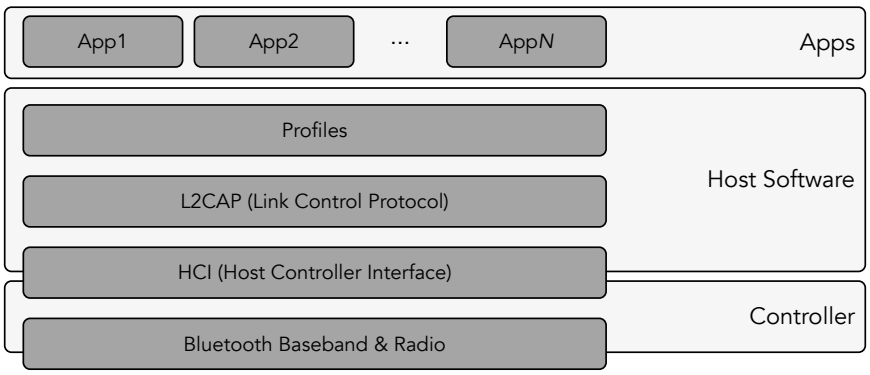

Figure 2.5: A simplified view of the Bluetooth stack.

During normal operation of Bluetooth, a physical radio channel is shared by two or more devices. A collection of devices sharing a single radio channel (synchronized to a master device) is referred to as a piconet. Without loss of generality, the following text describes relevant internals of a physical channel shared by only two devices.

Within the context of a shared, physical radio channel, there is a complex layering of links and channels and associated control protocols that enables coordination amongst the devices as well as data to be transferred between devices. A detailed description of each of these channels and links is beyond the scope of this dissertation (see the Bluetooth Specification [35], Volume 1, Part A). Worthy of note in our work is, however, the Logical Link Control and Adaptation Protocol (L2CAP) channel. L2CAP channels provide a channel abstraction to applications and services. The L2CAP layer of the Bluetooth protocol carries out many functions, including segmentation and reassembly of application data, and multiplexing and de-multiplexing of multiple channels over a shared logical link. Application data submitted to the L2CAP protocol may be carried on any logical link that supports the L2CAP protocol.

Summary

In our work we focus primarily on two aspects of the Bluetooth protocol: the HCI and the L2CAP. The HCI is the interface between software (i.e., the Bluetooth Host software) and Bluetooth I/O hardware (i.e., the Bluetooth controller firmware and hardware); by extending this interface, we can enable trusted software to create secure channels for Bluetooth data within the hub. The L2CAP protocol is the primary protocol for enabling applications and services; by applying Trusted I/O security at the L2CAP layer, we can offer fine-grained, channel-based protection for user data.

2.2.2 Intel SGX

Intel Software Guard Extensions (SGX) [100] is a set of new instructions and mechanisms that can be used by app developers to protect selected code and data from disclosure or modification by partitioning apps into CPU-enforced containers known as enclaves (Figure 2.6). Enclaves offer protected areas of execution in memory that increase security even on compromised platforms. Specifically, an enclave provides confidentiality, integrity, and replay-protection guarantees, even without relying on trusting drivers, middleware, OSes, VMs, hypervisors, firmware, or the BIOS. SGX also provides remote and local attestation capabilities, allowing enclaves to be measured and verified – or in other words, a means for an enclave to provide proof of its authenticity.

More information on SGX is available through Intel's official SGX documentation [100] as well as past academic research (e.g., [130, 189, 59]). Today, however, Intel's SGX does not support Trusted I/O features.

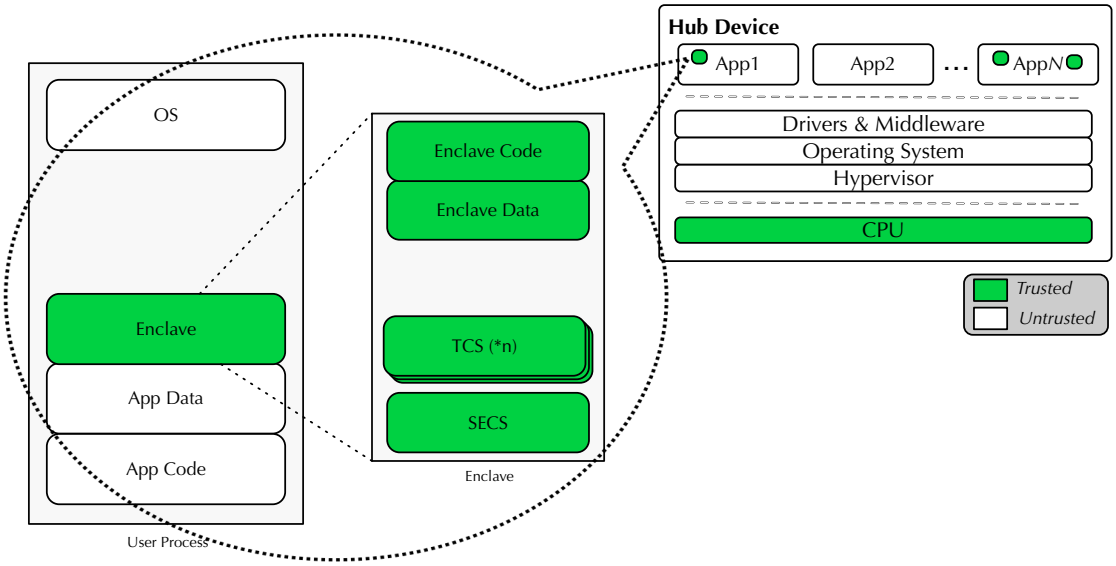

Figure 2.6: Overview of SGX and anatomy of an SGX enclave.

2.3 System & Security Model

In this dissertation, we address the trusted-path problem for Bluetooth I/O, as illustrated in Figure 2.4. We break the overall path between the trusted software and the user (device) into two subpaths: (1) the path between trusted software and the trusted Bluetooth Controller (Figure 2.4, E1-E2), and (2) the path between the trusted Bluetooth Controller and a trusted Bluetooth device (Figure 2.4, E3-E4). As in related work [189, 200], we assume that the latter path is secure today (e.g., through standard Bluetooth OTA security). Our work focuses on the former path and how trusted software and the trusted Bluetooth Controller can secure I/O channels through untrusted Host software *within* the hub. Towards this end, this section describes our goals and security model.

2.3.1 Threats & Adversaries

While there are certainly many types of adversaries and threats that one can imagine, in this dissertation we are primarily concerned with software-based adversaries,

including privileged software. Specifically, we concentrate on preventing two types of software attacks: (1) unauthorized access attacks, which aim to access sensitive user data transported via Bluetooth, and (2) data-injection or replay attacks, which aim to inject data that is not authentic, or replay authentic data for malicious purposes.

To this end, we consider an adversary (malware) that has various capabilities and employs various tactics to successfully perform such attacks. Specifically, adversaries ***can*** read or write the code and data of system software and untrusted apps on the hub. Adversaries can create their own trusted/untrusted apps. Adversaries can also interpose on communication (i.e., intercept, insert, alter, deny messages), either between trusted software and Trusted I/O hardware *within* the hub or between the hub and device (or both). Adversaries ***cannot*** physically access the hub device or any of its connected Bluetooth devices. Adversaries cannot read or write the data or code of trusted software that is protected within a TEE, nor the data or code protected by the trusted hardware. Adversaries cannot break encryption primitives or protocols known to be secure today. Denial-of-Service (DoS) and side-channel attacks are out of scope for our work.

2.3.2 Assumptions & Trust Model

In our work we assume the human user is not an adversary. We assume that the Bluetooth I/O device is implemented correctly and is trusted by the user to faithfully handle I/O on their behalf; furthermore, we assume the device accurately identifies itself to the hub. We assume the hub has SGX-enabled hardware. We assume that system software (e.g., drivers, OS) and other apps are *not* trusted. We assume all trusted software and trusted hardware (including firmware) is implemented and authenticated/loaded/initialized/booted correctly. We assume trusted apps trust that the I/O Controller will comply with security policies (Section 2.3.4), not

disclose user I/O data to other devices or apps, and implement OTA protection between the client and any Bluetooth devices. We assume the physical channel between the hub and device (Figure 2.4, E3-E4) is protected with existing Bluetooth OTA security.

2.3.3 Goals

In light of these threats and assumptions, we seek to achieve the following four goals: **(G1)** We aim to design a hub-side architecture that is not dependent on trusted hypervisors, trusted OSes, or trusted drivers for security. **(G2)** We aim to provide confidentiality, integrity, and replay protection guarantees over select user I/O data between trusted software and hardware. **(G3)** We aim to provide protection against impersonation of trusted software or hardware. And last, **(G4)** we aim to achieve these goals in a way that does not interfere with existing I/O protocols (namely, Bluetooth) or break existing routing mechanisms.

2.3.4 Bluetooth Trusted I/O Security Policies

Bluetooth applications and services rely on L2CAP channels to transport user data. Thus, we apply Bluetooth Trusted I/O security to L2CAP channels that carry user data.

A Bluetooth Trusted I/O security policy specifies what Bluetooth I/O data needs to be protected between a trusted app and the Bluetooth Controller, as well as information used to secure the data. Specifically, a policy is made up of two elements: (1) information to identify the specific channel(s) that should be secured (e.g., user data from a Bluetooth keyboard), and (2) a symmetric key; the symmetric key is used to apply cryptographic protection over the specified channel's data. Here, a *secure* channel is one with the properties noted in goals G2 and G3 described above. We discuss what information is needed to identify specific channels that

carry user I/O data in Section 2.4, and how security policies can be *programmed into* the Controller in Section 2.5.1.

Because our architecture aims to provide confidentiality of user I/O data (G2), a trusted app that configures a security policy has exclusive access to the channel(s) defined in the policy. This step ensures that no other software can access the Bluetooth I/O data that the trusted app aims to secure. The OS enforces exclusive use of some devices today (e.g., keyboards, cameras) for security. For example, by default, the OS ensures keyboard input is only sent to the app that has focus to ensure that no other app can observe passwords or other sensitive data entered by the user. Since the OS and other system software is not within the TCB of our architecture, however, we aim to provide similar guarantees without relying on this untrusted software.

2.4 Channel Selection & Security Policy Enforcement

Today, the hub's Bluetooth Controller is the gateway for all packets transported between the client's Host software and all connected Bluetooth devices (Figure 2.7). An implication of the Controller's current design and role as gateway, however, is that all ingress Bluetooth packets (device-to-hub), from all connected devices, are multiplexed within the Bluetooth Controller into a single stream and delivered to Host software via the HCI. Similarly, all egress packets (hub-to-device) must be demultiplexed within the Controller and sent to the correct Bluetooth device.

In our work we seek to not interfere with the Bluetooth protocol or break existing routing within the hub's software (G4). Therefore, our Trusted I/O solution aims to selectively secure only user data, and only between trusted apps and designated devices, while allowing all other apps and devices not requiring secure I/O to communicate as usual. Realizing Trusted I/O for Bluetooth has raised

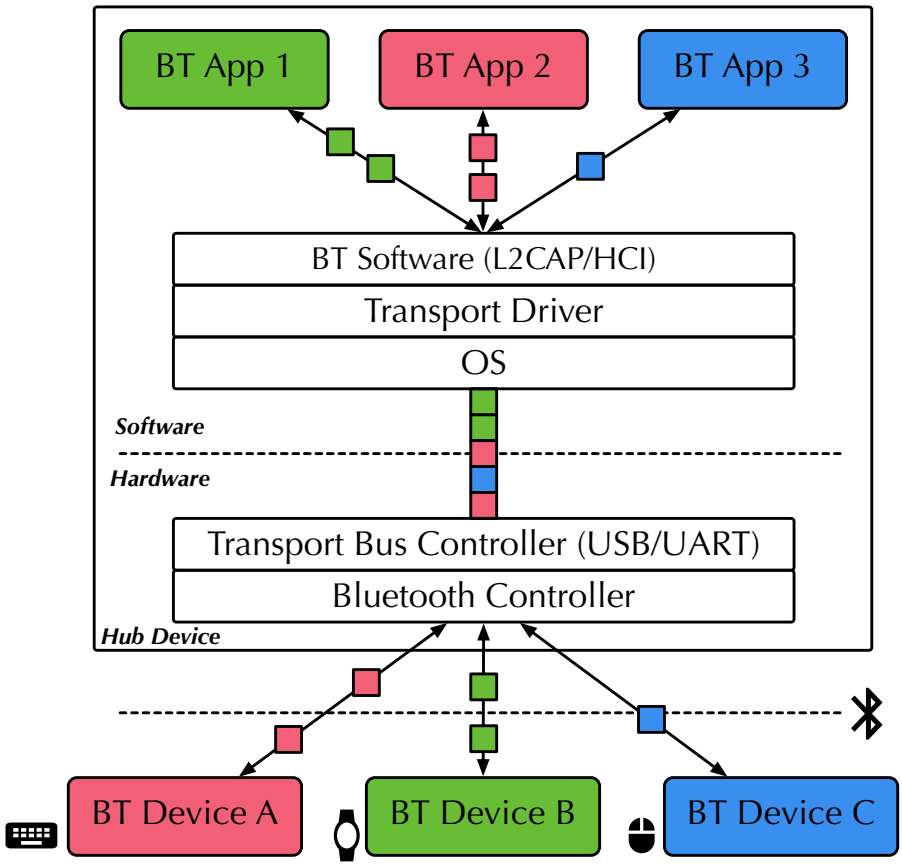

Figure 2.7: Illustrating Bluetooth's packet routing. Packets sent from Bluetooth devices are multiplexed into a single stream and interleaved as they are sent to Host software. Packets sent to Bluetooth devices must be de-multiplexed and associated with their respective device before transmission from the hub.

three main challenges: **(C1)** associating multiplexed and interleaved packets to their respective devices and channels; **(C2)** isolating and protecting only user data (without interfering with control channels or packet headers); and **(C3)** overcoming tension between enforcing high-level security policies given only low-level context. We discuss these issues next and refer to aspects of Figure 2.8 and Figure 2.9.

2.4.1 Packet Multiplexing & Interleaving

Today, a Bluetooth Controller must maintain basic metadata about connected devices and their channels so that it can route packets to Host software. Conceivably,

a Bluetooth Controller need only maintain a mapping between an HCI connection handle and the corresponding device's Bluetooth address, enabling the Controller to know which packets belong to which devices. Knowing how to differentiate packets by device, while necessary, is not sufficient for securing user I/O data. To elaborate, actual user data is transported between Host software and Bluetooth devices using L2CAP channels. All L2CAP packets (Figure 2.8) have a channel identifier (CID), and according to the Bluetooth specification, the L2CAP CID of a packet enables routing software to associate packets with specific L2CAP channels. Unfortunately, the CID used in L2CAP packets is guaranteed to be unique only *per device*. This can give rise to ambiguity. For example, Figure 2.9 illustrates a hub connected with two Bluetooth devices, each with one channel used to exchange user data with the hub. Per the Bluetooth specification, it is actually reasonable for the channels to have the same CID. The disambiguating attribute in this case is the identifier for the physical link (i.e., the HCI connection handle or Bluetooth Device Address, which identifies a unique Bluetooth device), which is not part of the L2CAP packet. While all of this information is not located in a single packet, the combination of an HCI connection handle *and* an L2CAP CID can be used to uniquely identify channels.

2.4.2 Isolating & Securing User Data Only

Distinguishing between channels within one device and between channels belonging to different devices is a necessary precursor for identifying and securing channels that carry user data. The next issue, however, is that the L2CAP packets mentioned above actually come in two different types: those that carry control information and those that carry user data. To ensure all user I/O data is secure, it may initially seem like a good idea to secure *all* packets – regardless of type –

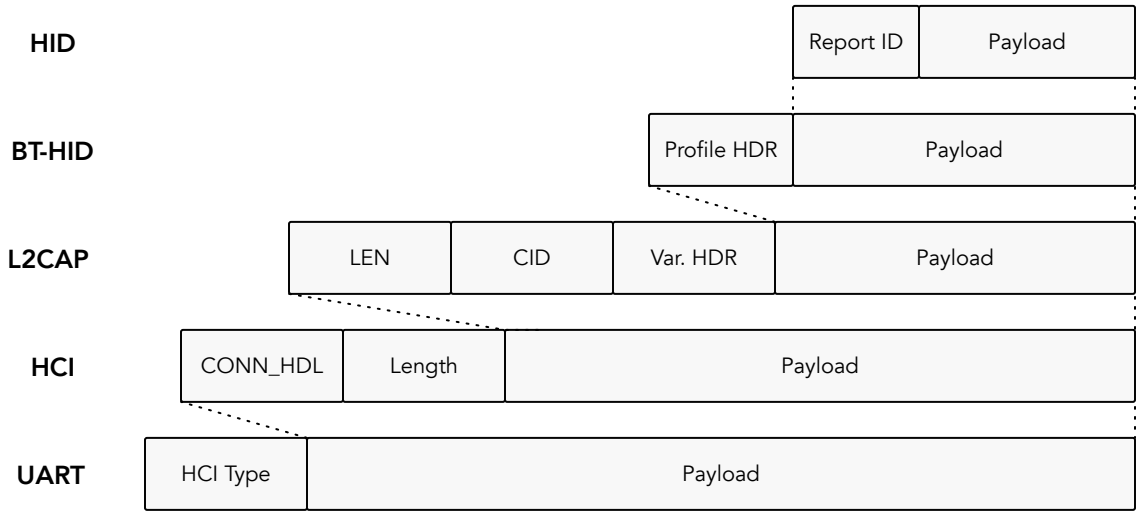

Figure 2.8: An example of the Bluetooth packet hierarchy. In our prototype work we show how input data from HID devices can be secured. User data (e.g., key presses) is transported in HID packets (top level). These packets are nested in multiple layers of the Bluetooth protocol for transportation from a device to a client's Host software.

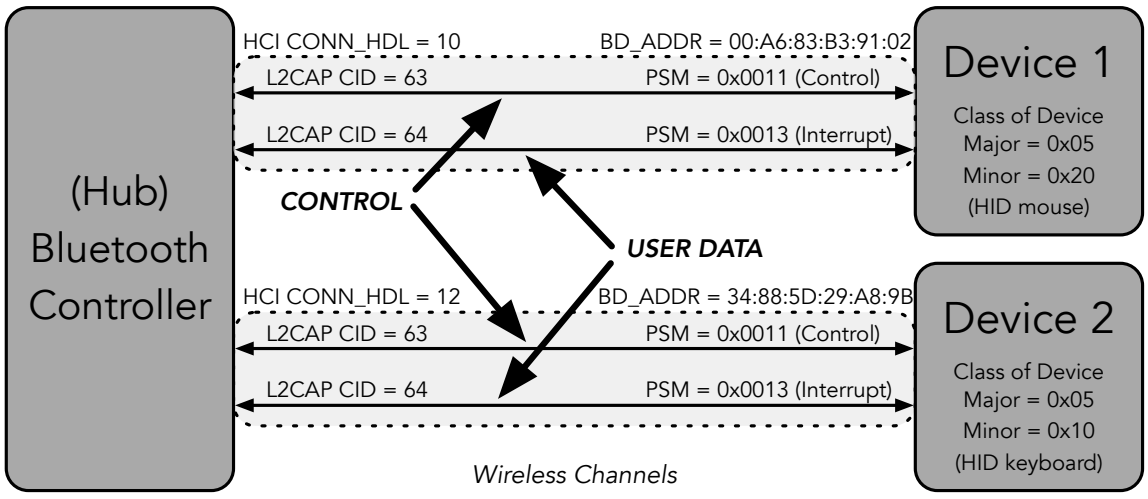

Figure 2.9: An example of two Bluetooth devices connected with a hub. Trusted I/O requires fine-grained channel selection. For each device and channel, a Trusted I/O-enabled Bluetooth Controller maintains information about: the physical connection (HCI Connection Handle), logical channels (L2CAP CIDs) and their respective protocol/service multiplexor (PSM; control vs. data), and Class of Device (COD; composed of Major and Minor numbers).

belonging to communication with a specific device. This approach, however, can have atrocious effects on existing Bluetooth functionality. For example, as depicted in Figure 2.9, Device 1 has one channel (L2CAP CID = 63) dedicated to handling signaling (e.g., enable device options, determine current device state) between the hub and device; as a consequence of securing *all* packets, Trusted I/O security would obfuscate (encrypt) this signaling channel and "break" application-level functionality. For other, primary signaling channels, this can be even more destructive, breaking functionality that controls how packets are routed, and how logical channels are created, maintained, and destroyed. In general, essential Bluetooth functionality relies on access to HCI connection handles, L2CAP channel identifiers, and in some cases, even information in the L2CAP packet payloads (e.g., control parameters). Thus, in order to not break existing Bluetooth functionality, packets that contain *control information* should not be secured, and packets containing *data* should only be secured if a relevant security policy exists.

The underlying issue here is that L2CAP packets have no type descriptor in the packet that can be used to disambiguate control packets from data packets. Such a descriptor is, however, present at the time that new L2CAP channels are created. During L2CAP channel creation, a Protocol and Service Multiplexor (PSM) value is exchanged. PSM values are useful for securing channels as they provide higher-level insight into the purpose of the channel and the type of information that will be exchanged over the channel. Furthermore, some data and control channels are defined in the specification and are allocated reserved channel identifiers that indicate their purpose (data vs. control). Thus, while channel type information is not contained within all packets, such information is either standardized (and therefore need not be directly observed) or available during the creation of channels (and is therefore observable within the Controller).

2.4.3 Understanding Device Types

More information may yet be required to enforce meaningful security policies. Thus far we have worked to disambiguate devices from one another, to disambiguate channels from one another, and even to disambiguate types of packets from one another. What we are lacking, however, is a higher-level understanding of connected devices. Specifically, we seek a mechanism to map security policies that are meaningful to humans and apps, to any connected devices. For instance, a Trusted I/O security policy may require that all I/O from/to a particular device, or particular *class* of device (such as all keyboard devices; see Device 2 in Figure 2.9), should be secure. Fortunately, Bluetooth defines the notion of Class of Device (COD), which provides higher-level information about the purpose and function of a device. Each Bluetooth device belongs to some class, represented by major and minor class information. Bluetooth currently defines 32 major classes (e.g., Computer, Phone, Peripheral, Health). There are many minor classes that describe subclasses of a particular major class; for example, given a major class of Peripheral, minor class information further describes if that Peripheral is a keyboard, mouse, or something else. According to the Bluetooth specification, "The major device class segment is the highest level of granularity for defining a Bluetooth device. A device's main function determines its Major Class assignment." Using Bluetooth Class of Device information to represent devices in security policies likely maps well to the high-level understanding of devices that humans and apps have.

2.5 BASTION-SGX

In this section we present BASTION-SGX: our Trusted I/O architecture for Bluetooth on SGX (Figure 2.10). BASTION-SGX is comprised of the following components: the hub's trusted software, the hub's Trusted I/O hardware, and some

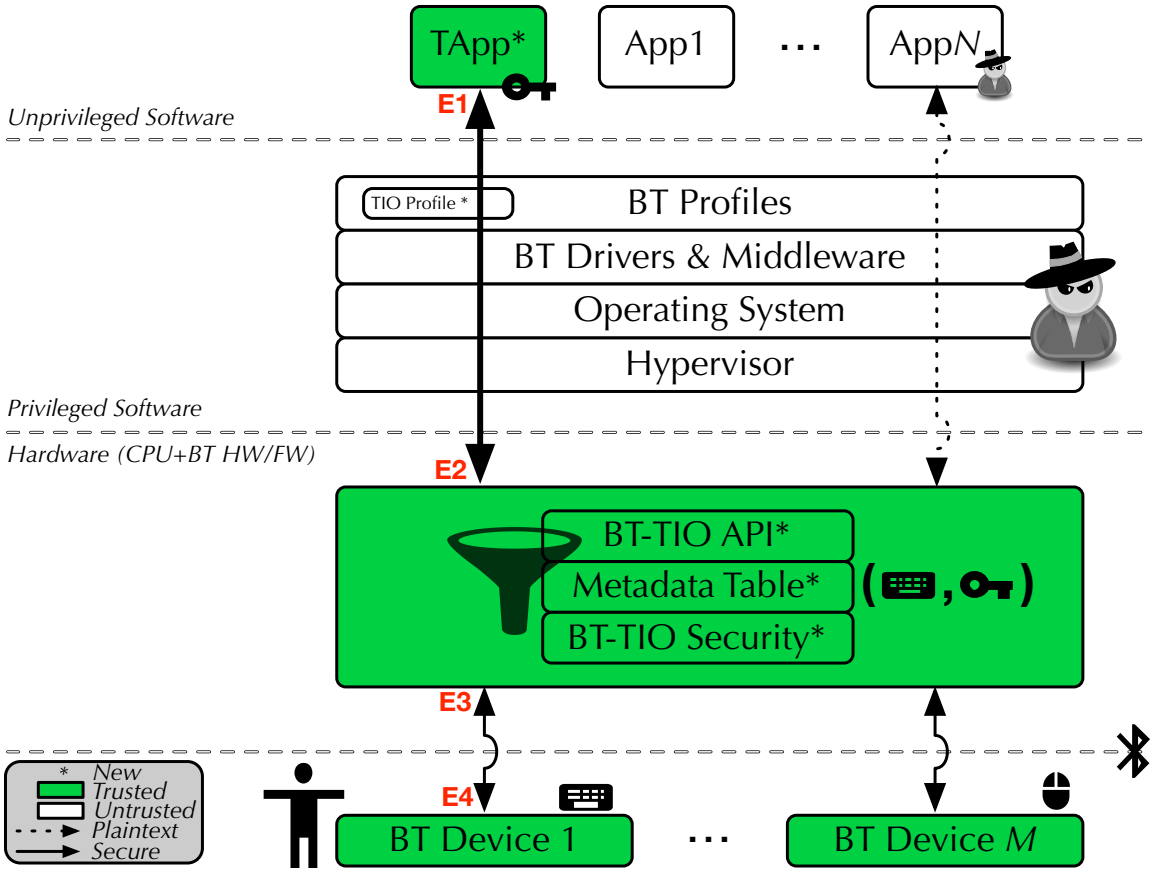

Figure 2.10: Overview of our Bluetooth Trusted I/O architecture. Our goal is to ensure data is secured in its transportation between two endpoints: trusted software (E1) and a Bluetooth device (E4). We assume the path between the hub's Bluetooth Controller and the device (E3-E4) is secure via Bluetooth OTA security that exists today. Our work shows how I/O channels between trusted software and the trusted Bluetooth Controller (E1-E2) can be secured. Together, these paths achieve our goal.

number of wirelessly connected devices. Specifically, the trusted software consists of standard SGX software and a trusted app. The trusted hardware is an SGX-enabled CPU and a Bluetooth Controller with our Trusted I/O extensions. Wireless devices connect with the hub and communicate with apps.

2.5.1 Bluetooth Trusted I/O Controller

BASTION-SGX is centered around a Trusted I/O-enabled Bluetooth Controller, which implements the following features: (1) monitor connection events and main-

tain a Metadata Table to store information for connected Bluetooth devices and their respective channels; (2) expose an API to support Trusted I/O-related interactions with Host software; (3) filter packets in accordance with the Metadata Table; and (4) apply cryptography to provide the desired security properties over a channel. We discuss these components in greater detail next and in Figure 2.10.

Connection Event Monitoring & the Bluetooth Trusted I/O Metadata Table

The solutions we envision for the various challenges described in Section 2.4 rest in our ability to obtain metadata about devices, and their respective HCI and L2CAP channels. We assert that it is possible to obtain most of the necessary information simply by extending the Bluetooth Controller's firmware. (Our solution also relies on information obtained from the trusted app via our new Trusted I/O-related APIs. We discuss this information in the next section.) Specifically, BASTION-SGX realizes new features to monitor HCI and L2CAP connection/disconnection events, and maintain a Metadata Table that contains the information alluded to in Section 2.4. Figure 2.11 provides a summary of the relevant HCI and L2CAP events, and the metadata that the Controller captures.

When a Bluetooth device first connects with a hub device, the Controller creates an *HCI Connection Handle* (`CONN_HDL`) that the client's Host software can use to communicate with that specific device. As an example, when a device connects, the Bluetooth Controller generates an HCI Connection Request event to inform Host software that a device wishes to connect. The device is described initially by its Bluetooth Device Address (`BD_ADDR`) and Class of Device (`COD`). Upon completion of the physical connection between the hub and device, the Controller generates an HCI Connection Complete event that provides Host software with the `CONN_HDL` that can be used for future communication with the device. At this point, the Host and Controller have an active HCI connection that can be used for subsequent

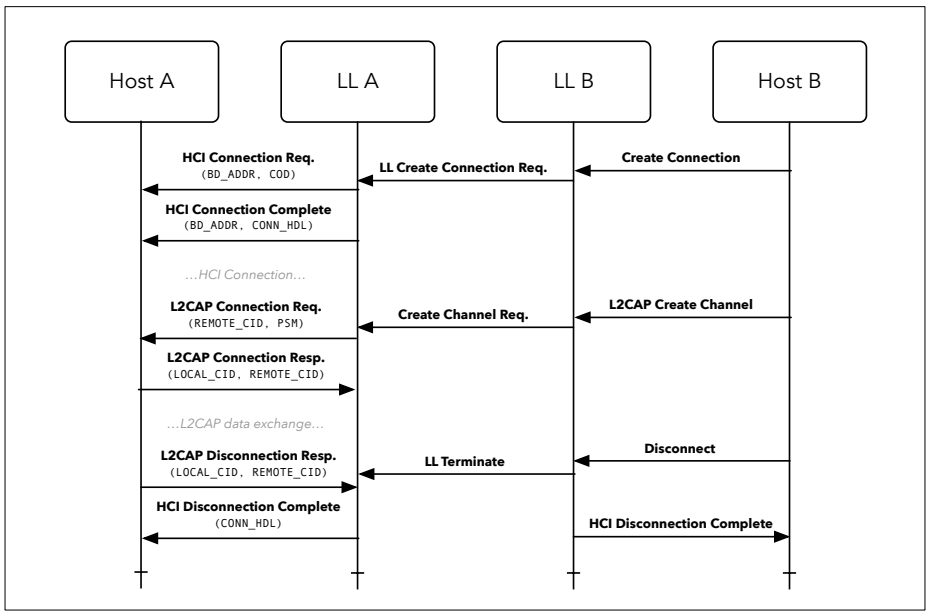

Figure 2.11: An example flow of a hub-device connection with a summary of the relevant HCI and L2CAP connection-related events. These events are standard in Bluetooth's HCI and L2CAP protocols today. Our solution adds extensions to the Bluetooth Controller to monitor and capture device- and channel-specific metadata during connection/disconnection events; the result is a Trusted I/O-enabled Bluetooth Controller. The relevant metadata is shown in parenthesis.

communications between the Host software and device.

Once connected, these entities can exchange L2CAP connection requests and responses to create logical links for exchanging control and data packets. For example, a device that sends a request to the Host to create a new L2CAP channel sends two pieces of information: a `REMOTE_CID` and `PSM`. The Protocol/Service Multiplexor (`PSM`) indicates the purpose of the channel; i.e., what protocol or service will operate over the new L2CAP channel. The L2CAP channel has two endpoints: one in the Host (`LOCAL_CID`) and one in the device (`REMOTE_CID`). While the Host is free to assign any `CID` for its local endpoint, it does not control the `CID` that the device uses for its endpoint. Thus, when new L2CAP channels are formed, the Host software and device carry out an acknowledgement of the `CID` to be used for their respective endpoints. Events related to HCI and L2CAP disconnections can be similarly observed.

BD_ADDR	CONN_HDL	LOCAL_CID	REMOTE_CID	COD (minor)	COD (major)	PSM	KEY
34:88:5D:29:A8:9B	0x0046	0x0041	0x0041	0x05	0x10	0x0011	0xXXXXXXXX...
34:88:5D:29:A8:9B	0x0046	0x0042	0x0042	0x05	0x10	0x0013	0xXXXXXXXX...
00:A6:83:B3:91:02	0x0047	0x0041	0x0041	0x05	0x20	0x0011	0xYYYYYYYY...
00:A6:83:B3:91:02	0x0047	0x0042	0x0042	0x05	0x20	0x0013	0xYYYYYYYY...
...	...	...	...	...	...	...	...

Figure 2.12: A simplified example of the Trusted I/O Metadata Table.

By monitoring these HCI and L2CAP events, the Controller can be made to capture and maintain fine-grained information about each connected device (`CONN_HDL`, `BD_ADDR`), its logical channels (`LOCAL_CID`, `REMOTE_CID`), its type (`COD`), and the protocols or services (`PSM`) operating over each channel. This information can then be used in accordance with security policies (Section 2.5.1) to filter (Section 2.5.1) and secure packets (Section 2.5.1). A simplified example of the Trusted I/O Metadata Table is shown in Figure 2.12.

Bluetooth Trusted I/O API

Trusted I/O features are aimed at giving trusted software the ability to create secure channels to protect specific I/O data channels. Thus, in BASTION-SGX, security policies (Section 2.3.4) are driven by the requirements of trusted software. There are two issues worth considering here: first, how trusted software can configure security policies within the Controller, and second, how trusted software can describe security policies, based on metadata the Controller independently maintains.

Configuring Security Policies. The HCI can be used to address the first issue. The HCI is already used for communication between the Host and Controller. Furthermore, the HCI protocol supports an extensible interface, often referred to as the Vendor Specific Debug Command (VSDC) interface. This interface enables vendors to add non-standardized features to Bluetooth Controllers and to enable apps to use those features. Thus, we can use this interface to support new Bluetooth Trusted I/O APIs – such as policy specification APIs for *adding* and *removing* security

policies – in Trusted I/O-enabled Bluetooth Controllers.

Class-of-Device Policy Specification. One approach to specifying security policies is to identify a class of devices that should be secured along with a key (`COD`, `KEY`), and rely on the Controller to determine which channels carry sensitive data versus those that do not. As noted in Section 2.5.1, upon connecting, `COD` information is exchanged; therefore the Controller can know the `COD` of each of its connected devices. Furthermore, because `PSM` information is exchanged during the creation of any L2CAP channel, the Controller can know the purpose of each L2CAP channel, enabling it to identify (and subsequently secure) channels that carry user data (using the policy's `KEY`). This approach for defining security policies is conservative in that it allows trusted software to secure I/O between it and *any* device matching the `COD` in its policy.

Bluetooth Trusted I/O Filtering

The Bluetooth Trusted I/O Filter is responsible for (1) identifying packets containing sensitive user data based on known devices (Section 2.5.1) and security policies (Section 2.5.1), and (2) securing these packets (Section 2.5.1) using the policy's `KEY`. Thus, as L2CAP packets are transported between the Host and Bluetooth devices, the filter examines *each* packet to determine if some security policy applies to that packet. Essentially, given an L2CAP packet, `PACKET`, an HCI connection handle, `CONN_HDL`, and information about the *direction* of the packet (host-to-device or device-to-host), `DIR`, the filter must decide to either apply Trusted I/O security to `PACKET` or allow it to pass through unaffected.

Each L2CAP `PACKET` (recall Figure 2.8) contains a `CID` and packet length (`LEN`). Also, the Controller knows the `CONN_HDL` to which the `PACKET` belongs, and which direction the packet is being transported. Therefore, the filter can check the Metadata Table to see if the `PACKET` belongs to a secure channel and apply the

appropriate security operations (encrypt, decrypt, etc.) over the `PACKET`'s payload based on `DIR`. Note that the boundary of the `PACKET`'s payload can be determined from the `LEN` field in the header of `PACKET`; all Trusted I/O security is applied to the payload. We discuss this step next.

Bluetooth Trusted I/O Security

The details of the security applied to user data (e.g., encryption and decryption algorithms, key sizes, MACs) are implementation and security-model specific. As noted in Section 2.3, BASTION-SGX aims to provide confidentiality, integrity, replay protection, and mutual authentication guarantees (G2 and G3). In BASTION-SGX, any authenticated encryption algorithm can be used to secure the channel. Such an encryption algorithm is applied to user I/O data within trusted software and the trusted Controller so that the data can be routed in the normal way (G4) via untrusted software; by securing the I/O data before moving it out of the trusted software or Controller, we ensure the data is opaque to untrusted software (G1). Furthermore, in doing so, we protect user I/O data against the malware attacks defined in Section 2.3 – which raises the bar significantly from today's solution. Thus, assuming trusted software has a secure mechanism to share keys with the Controller, BASTION-SGX can achieve its security goals. We note that this key-sharing step is critical to Trusted I/O security, yet distinct from the contributions we describe in this dissertation. In fact, there is nothing especially novel behind how this key sharing can be done, and as such, is not an emphasis of this dissertation. To convince the reader that this step is not an issue, we briefly describe two approaches next.

Dynamic Key Provisioning. This approach requires further extensions to the Bluetooth Controller that enable it to attest itself to an enclave, enabling a Controller to prove to an enclave that it is authentic Bluetooth hardware, executing authentic

Bluetooth firmware; the attestation we envision is similar to what is proposed in the USB Type-C Authentication Specification [70] and the PCIe Device Security Enhancements Specification [58].[1] These specifications define nearly identical authentication architectures that allow USB and PCIe devices, respectively, to have their identity and capability cryptographically verified. These architectures provide a specific example where cryptographic verification can be used to subsequently exchange secrets to set up a secure channel between software and a device; such a channel, in our work, would be used by trusted software to configure security policies (Section 2.3.4) within the Controller.

At a high level, these authentication architectures adapt common industry paradigms (i.e., PKI) for identity and capability verification. Specifically, a trusted root certificate authority (CA) generates a root certificate; the root certificate is used by an authentication initiator (verifier) to verify the validity of signatures generated by a device (prover) during authentication. The root certificate is also used to endorse vendor certificates, which are then used to endorse some combination of intermediate certificates and model certificates; these certificates are ultimately used to endorse per-device certificates.

Authentication happens in two steps: (1) Authentication Provisioning, and (2) Runtime Authentication. In the authentication provisioning step, the root certificate is provisioned to the authentication verifier (in our case, trusted software) to enable the verifier to verify the validity of signatures generated by a device (in our case, a Bluetooth Controller) during the runtime authentication step. Furthermore, a public/private key pair is generated for each device; the private key is provisioned into the device at the time of manufacturing along with a certificate that contains its corresponding public key, along with a signature that can be verified using the root

[1] At the hardware level (i.e., within the hub), components connected with the CPU via PCIe, USB, UART, etc., are commonly referred to as "parts" or "devices." Thus, references to "devices" in this context are distinct from how we use the word "device" throughout the rest of this dissertation, which is to refer to Bluetooth devices.

CA's public key in the root certificate. In the runtime authentication step, the verifier queries the device to obtain its certificate, and sends a unique challenge (nonce) to the device; the device can authenticate its identity and capability by signing the challenge along with other authentication data (e.g., a measurement of its firmware) with its private key. The verifier can verify the device's response/signature using the device's public key and the root CA's public key (as well as any intermediate public key), and use the result of its verification to make a trust decision.

Assuming the device (again, in our case this is the Controller) successfully attests to its authenticity, standard protocols such as DAA-SIGMA [186] can be used to share a secret and establish a secure channel between an enclave and the Controller; the enclave can then use the secure channel to share Trusted I/O keys with the Controller.

New Platform Capability. An alternative approach envisions a new platform capability, similar to previously-envisioned capabilities. For example, the authors of the Secure Input/Output Device Management patent [129] describe a scenario similar to ours: a processor has secure execution environment support (e.g., SGX) and wishes to establish a secure connection to an I/O controller. The I/O controller includes an integrated Trusted I/O component that can receive (unencrypted) requests to configure the Trusted I/O component. In the patent, the authors provide details for how a USB controller can be equipped with trusted I/O capabilities and how encryption keys can be established between an enclave and the USB controller. Our work can use similar features for a Trusted I/O-enabled Bluetooth Controller. This new capability would make secure key sharing a feature of the platform (via ISA extensions), and allow enclaves to send Trusted I/O keys securely to an authentic Bluetooth Controller.

2.5.2 Trusted I/O Host Software

In BASTION-SGX, trusted apps are implemented as SGX enclaves. Trusted apps are therefore subject to the same security model as SGX, and benefit from existing work towards resources for enclave software development [96, 130]. Trusted software uses enclaves to protect select code and data within the enclave, and uses our Trusted I/O features to secure I/O data between itself and the Bluetooth Controller.

In theory, a trusted app can be quite simple. To create secure Bluetooth I/O channels, a trusted app needs to configure security policies within the Controller using the Bluetooth Trusted I/O API (Section 2.5.1). For each new secure I/O channel, a trusted app should use a new symmetric key, which can be generated using third-party libraries such as mbedTLS [20], for example. The trusted app can then use one of the mechanisms described in Section 2.5.1 to securely configure security policies (which include the key) into the Controller. A trusted app also needs to perform cryptographic operations on incoming/outgoing data; the SGX SDK [100] offers various functions to help developers with these sorts of operations, though again, third-party libraries (e.g., [20]) can also be used.

In some cases there may be a need for additional trusted software (e.g., Trusted Bluetooth Profiles) to support trusted apps that use Trusted I/O features. (See Section 2.7 for an example.) In today's solution, the OS and various drivers are trusted, so it is not a problem to have them process and interpret the contents of I/O packets to make them useful for apps. Our security model rules the OS and these subsystems out of the TCB; our work is therefore unable to rely on them for these services. Alternatively, a trusted app can opt to implement any data processing/interpretation that is needed (as we do in our work). Both of these options are viable.

We envision extending the existing SGX SDK [100] in future work to include

our Bluetooth Trusted I/O extensions for SGX. Specifically, these further extensions would implement common Bluetooth Trusted I/O operations such as key generation and encryption/decryption for securing I/O data, a security policy API, and so forth, alleviating the need for developers to implement these features in their apps.

2.6 Analytical Evaluation

In this section we present a analytical evaluation of BASTION-SGX. We are most interested in two evaluating two aspects of our solution: the memory overhead of managing additional metadata for Trusted I/O, and impact on network performance (latency and throughput).

Trusted I/O Memory Overhead

Recall the metadata table from Figure 2.12. Each row in the table consumes at most 32 bytes (`BD_ADDR` = 6 bytes, `CONN_HDL` = 2 bytes, `LOCAL_CID` = 2 bytes, `REMOTE_CID` = 2 bytes, `COD` = 2 bytes, `PSM` = 2 bytes, `KEY` = 16 bytes). In practice, Bluetooth controllers support connections from three to eight devices, though some Bluetooth hardware is known to support as many as twenty devices.

Figure 2.13 illustrates that the memory overhead for our Trusted I/O solution scales linearly with the number of channels requiring security. In a scenario common to today, where no more than eight devices are connected with a hub, and each device has only one channel that requires Trusted I/O security, the metadata table that is maintained for our solution would consume a modest $8 \times 23 = 184$ bytes of memory. In fact, even if we envisioned a more capable Bluetooth controller that supported up to 100 devices, the metadata table would consume just over $2K$ bytes of memory. Ultimately, the cost of our solution in terms of additional memory consumption is dependent on the hardware's resources – specifically, the number

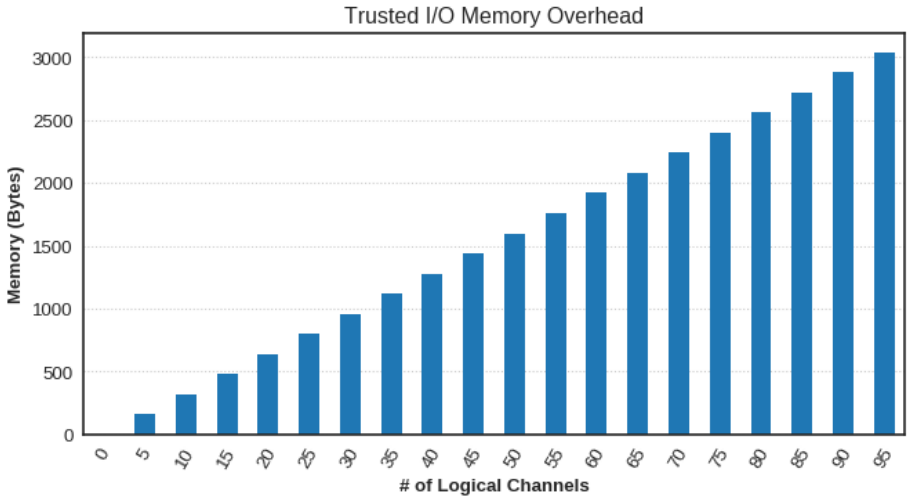

Figure 2.13: Trusted I/O memory overhead. Memory consumed by the Metadata Table is ultimately limited by the number of connected devices that the hardware supports. The memory consumed scales linearly with the number of logical channels utilizing Trusted I/O security.

of connected devices that the hardware supports.

Trusted I/O Latency & Throughput

To obtain insights into the impact of BASTION-SGX on network performance, we draw from measurements performed in related work [128, 76, 152]. In particular, we utilize timing measurements for BLE data transfer, as well as encryption, decryption, and MAC operations, which are used to protect data channels. For the purpose of this analysis, let us suppose that two devices are already connected and we are interested in computing the additional per-packet performance impact of Trusted I/O.

To reason about the impact of Trusted I/O, observe that, for packet ingress, a Trusted I/O-protected channel incurs the additional cost of a table look-up (negligible), one packet (re-)encryption within the Bluetooth controller, and one packet decryption in the receiving app. For packet egress, a Trusted I/O-protected channel incurs the additional cost of an encryption in the app and a decryption in the Bluetooth controller. Thus, regardless of the destination of the packet, each data

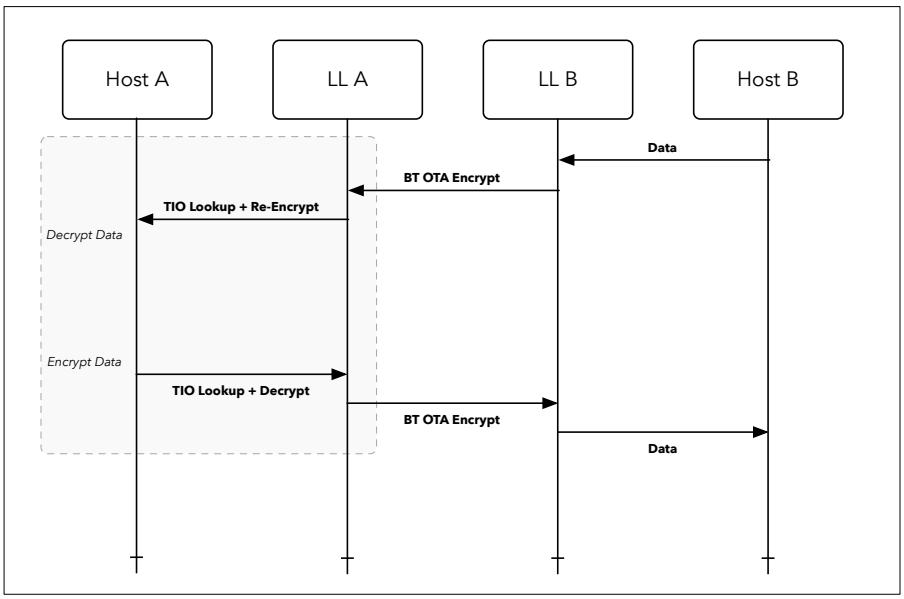

Figure 2.14: Trusted I/O Operation Overhead.

packet protected by Trusted I/O is encrypted and decrypted one additional time. For clarity, this observation is illustrated in Figure 2.14.

Common approaches to measure throughput are to use Round-Trip Time (RTT) or One-Way-Trip (OWT) Time. The RTT simply measures the elapsed time between when a packet is sent from $\mathcal{A}$ to $\mathcal{B}$ and back. Assuming the delays in both directions are approximately the same, the OWT is half of the RTT. Using RTT to provide analytical insight into achievable throughput with Trusted I/O, one can (in theory) send as much data as possible, apply the expected (average) latency, and evaluate how much data gets through. This raises two questions: *How much data can be sent per second?*, and *What is the latency?*

To proceed, we can use the following equation to calculate throughput:

$$\begin{aligned} \text{Throughput} &= \text{Packets Per Second} * \text{Data Per Packet} \\ &= \frac{N_{connInterval} * d}{connInterval} \end{aligned} \tag{2.1}$$

where $N_{connInterval}$ is the number of packets exchanged per connection interval,

d is the amount of data per packet, and $connInterval$ is how often two devices communicate (in milliseconds; a minimum of 7.5 milliseconds and increases in steps of 1.25 milliseconds).

The maximum application layer throughput for a connection between two devices has be obtained in prior work through both simulation and mathematical analysis [76]. To summarize: the theoretical data rate of Bluetooth varies depending on the version. For instance, in Bluetooth v4.2, the maximum BLE data rate is $1Mbps$. In practice, however, the data rate at the application layer is substantially lower – the maximum theoretical application layer throughput is $236.7Kbps$, but real-world experiments suggest that the application layer throughput is more like $58.48Kbps$ [76]. This disparity is due to a variety of factors (e.g., protocol overhead, connection interval (CI), Inter Frame Space (IFS), the number of devices in the piconet, implementation limitations) that are beyond the scope of this analysis.

In a BLE connection, each device communicates with the other at least once during a period known as the CI. By default, even when neither device has data to send, each device will transmit an empty Link Layer packet. The minimum CI per the Bluetooth specification is $7.5ms$. Within the CI, devices are allowed to transmit as many packets as desired, restricted only by the IFS (required time between packets), and the fact that each packet sent from $\mathcal{A}$ to $\mathcal{B}$ results in a response (ACK) from $\mathcal{B}$ to $\mathcal{A}$. While it is theoretically possibly to transmit more packets per CI, most devices transmit at most four data packets per CI. Even in ideal scenarios (e.g., extremely low Bit Error Rate (BER), using the smallest possible CI), past work evaluating BLE observed that the average one-way trip latency is $1ms$ [76]. For reference, we recreate figures (Figure 2.15) from PunchThrough's article on BLE throughput [162], which illustrates the maximum throughput, minimum connection interval, and maximum number of packets per connection interval for various iOS and Android devices.

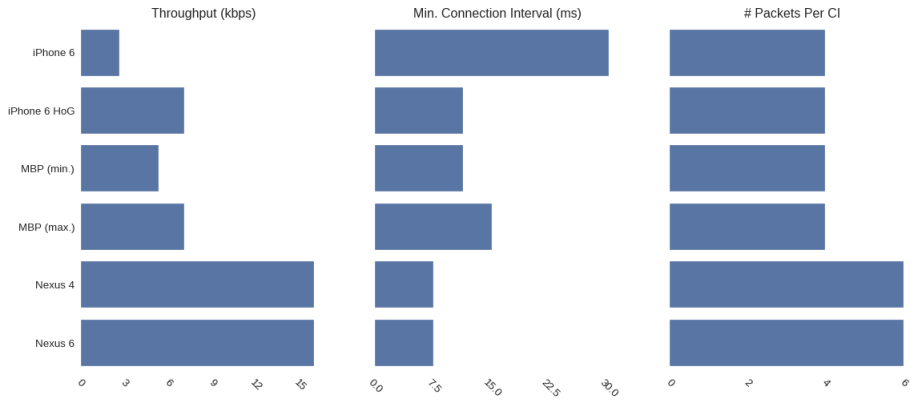

Figure 2.15: Examples of commodity devices and their maximum throughput, minimum connection interval, and maximum number of packets per connection interval. We recreate these figures using data obtained from PunchThrough's article on BLE throughput [162].

BASTION-SGX is technically agnostic to the security parameters and operations for Trusted I/O. For illustration, however, let us assume a deployment of BASTION-SGX uses strong, standard techniques to protect the channel, such as AES-CTR, AES-CCM, or AES-EAX.[2] Let us consider the use of AES-EAX. In recent work evaluating the performance of cryptographic algorithms across various IoT platforms and operating systems [152], it was observed that one encryption operation for AES-EAX consumes $< 100\mu s$ of time on average for a message sizes of less than 100 bytes. (The associated decryption displayed similar performance.) Since each packet in Trusted I/O, regardless of ingress or egress, incurs the cost of one additional encryption and decryption, which adds $200\mu s$ on top of the average latency of a BLE packet.

Combining all of this information suggests that Trusted I/O adds approximately 20% overhead to the transfer of a single packet. Notice, however, that if the average latency for BLE is $1ms$, and Trusted I/O adds $200\mu s$ to protect a single packet, and four packets are sent per CI,[3] the total latency is still less than the CI.

[2]AES-CCM and AES-EAX are schemes for achieving Authenticated Encryption with Associated Data (authenticated encryption with associated data (AEAD). Counter Mode (CTR) is a mode of operation that turns a block cipher into a stream cipher by including successive values of a counter value.

[3]Past work has observed that the maximum number of packets per connection interval is

Thus, the impact to latency is quite small, and since the number of data packets per CI remains the same, there is no impact to the overall throughput.

2.7 Case Study: Securing Bluetooth I/O on Intel's SGX

Here, we validate the Trusted I/O Controller and its role in our architecture (Section 2.5.1). Specifically, we seek to validate: (1) that our metadata table can be built and enables unique channel selection for Trusted I/O security; (2) that security policies can be added/removed to/from the Controller by creating new Vendor Specific Debug Commands (VSDC); (3) that packet filtering can isolate data-carrying packets and encrypt only packet data; and (4) that only the trusted app can recover (decrypt) I/O data over the secure channel it establishes with the Controller.

To this end, we built a prototype of our architecture and an example trusted app on an SGX-enabled platform running the Linux OS (Figure 2.16). The official Bluetooth Host Software used by Linux is BlueZ [36]. In our current prototype, we modified Bluetooth firmware that runs within an Intel Bluetooth Controller, adding the features we describe in Section 2.5.1. We describe our prototype in more detail next.

2.7.1 Implementation of the Trusted I/O Controller

On initialization of the Controller, we allocate space for the metadata table. We added hooks into the existing firmware to monitor HCI and L2CAP connection/disconnection events, and update the metadata table accordingly. We also extended the Controller to support two new VSDCs for adding/removing security policices: `TIO_SET_KEY` and `TIO_CLEAR_KEY`. As packets arrive in the Controller, the

dependent on the BLE stack and chipset, and is limited to four packets per interval on iOS and six packets per interval on Android. Thus, even if it is theoretically possible to transfer more packets during the interval, the BLE stack or chipset enforces a limit [162].

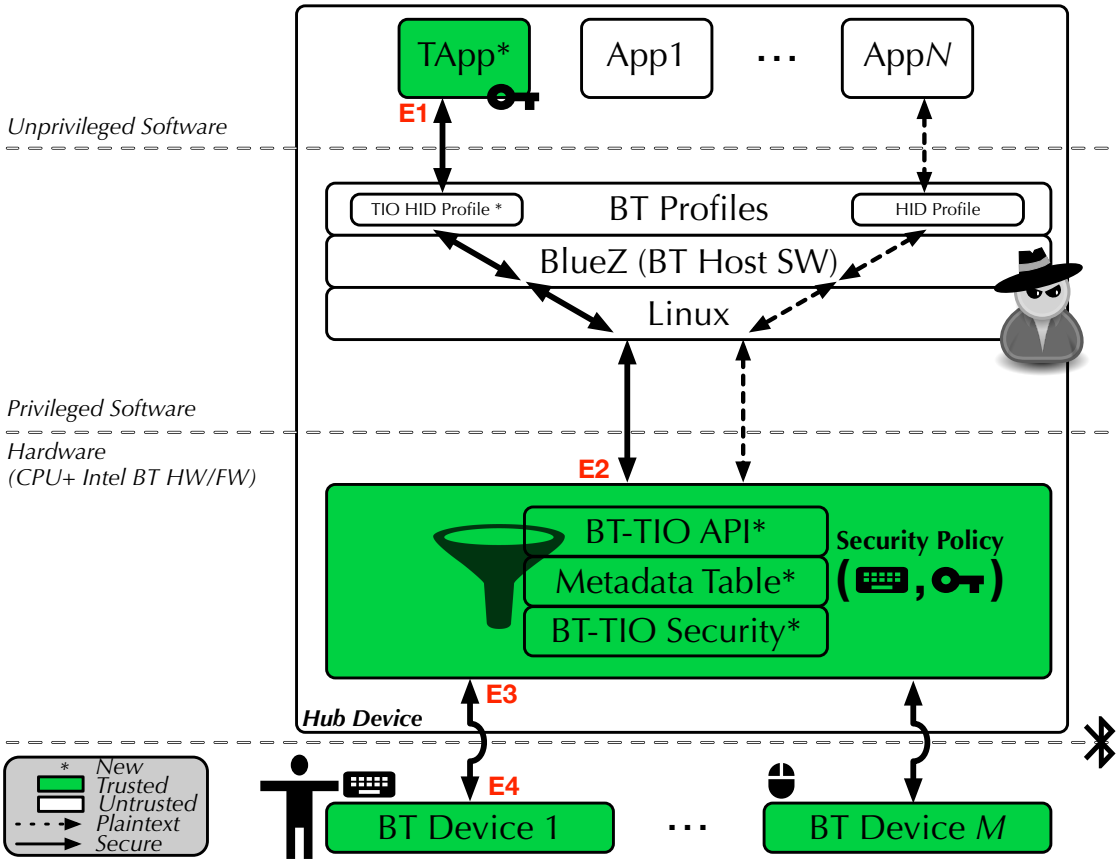

Figure 2.16: Adaptation of our Trusted I/O architecture to illustrate our prototype work.

Controller looks up information about the packet (i.e., to which channel it belongs) to determine whether further action is necessary (Section 2.5.1). We use the `KEY` programmed via `TIO_SET_KEY` to secure the relevant channel between trusted software and the Controller. Because the metadata table maintains information about each connected device and all of their logical channels, our current implementation uses the presence of a `KEY` for a particular channel as a flag to indicate whether that channel is currently a secure channel. Together, these steps enabled us to validate the above-mentioned objectives.

2.7.2 Validating Trusted I/O

We also implemented a trusted app (TA). We specifically considered a TA that prompts the user for a password and wants to ensure that password entry is secure. We installed a privileged keylogger on the hub to verify that the channel is in fact secure during password entry; the keylogger monitored all transactions over the HCI and logged all HID data.

At the time a user enters the password field context, the TA generates a symmetric key and uses the Bluetooth Trusted I/O API (Section 2.5.1) to send it to the Controller, indicating that it wants to secure input from the connected keyboard device. As a user types her password, the Bluetooth device generates packets containing the key presses. Because the device and hub were previously paired, they share a symmetric key and use it to protect user data in the OTA segment of the I/O path.[4] As L2CAP packets arrive in the Controller, the Controller uses the OTA key to decrypt them. *Without* Trusted I/O, the Controller need only map the link identifier to the HCI connection handle, and transport packets to the Bluetooth Host software, which in turn routes the packets to the appropriate apps. *With*

[4]The OTA symmetric key is negotiated between the hub's Controller and device's Controller. Host software (trusted or untrusted) does not have access to the OTA key.

Trusted I/O, however, the Controller's filter first checks to see whether the packet belongs to a Trusted I/O channel (Section 2.5.1). If the packet belongs to a Trusted I/O channel, channel security is applied (Section 2.5.1) using the `KEY` programmed by the TA previously. If the packet does not belong to a Trusted I/O channel, no channel security is applied. In either case, the packet is encapsulated within an HCI packet and sent to Host software (and ultimately routed to the appropriate app) via the normal transport (e.g., UART). When a TA receives packets, it decrypts and verifies the contents.

One technical challenge that arises in our prototype work is the handling of Human Interface Device (HID) input. Because HID devices are an important part of modern computers, there are drivers and other middleware that help to process and interpret HID data. For example, keyboard input is sent through a HID subsystem that translates scan codes into text characters. In reality, this translation is fairly simple, and the TA can implement it in its own code – indeed, the TA in our prototype handles the translation itself. Alternatively, one can envision trusted middleware that handles these sorts of standard operations. In light of this, we need to prevent Host software from trying to interpret certain (HID) packets that have been secured as part of a Trusted I/O channel.

In Bluetooth, HID packets are encapsulated within Bluetooth HID packets, which are then encapsulated within L2CAP packets (recall Figure 2.8). The Bluetooth HID layer serves as a lightweight wrapper of the HID protocol defined for USB; this enables the re-use of Host software that already exists to support USB-based HID. By default, when a Bluetooth HID device connects, Host software routes its HID packets through the relevant HID subsystems, processes the packet contents, and then makes the data available to apps. To prevent the Host software from routing Trusted I/O HID packets through these HID subsystems – and erroneously interpreting packet contents – we installed a new Bluetooth profile:

the *Trusted I/O HID Profile*. This profile is functional software that exists solely to prevent premature interpretation of data, and instead, passes data to the intended trusted software for handling. We emphasize that this "glue" software is not part of our TCB: it is untrusted *functional* software that is needed only to prevent Host software from incorrectly handling certain packets.

2.8 Related Work

Addressing the trusted path problem for Bluetooth I/O raises a number of challenges that we confront in this work. Our approach bears some resemblance to the trusted path work by Zhou et al. [200]. They propose to build a trusted path between a program endpoint (trusted app) and a device endpoint (I/O hardware); they rely on a non-standard hypervisor to offer trusted-path isolation from untrusted software. In our work, we eliminate any need to rely on trusted drivers, OSes, hypervisors, and so forth, for security; all data is secured within the Bluetooth Controller and the trusted app, and is therefore opaque while in transit through untrusted software. In another related work, researchers present SGXIO [189]. In SGXIO, the trusted path must be built from a user app (enclave) to a Trusted I/O driver, and from the driver to the respective I/O device. Again, this work relies on a hypervisor to realize a secure binding between the Trusted I/O driver and the actual I/O hardware. In our architecture, a specific I/O Controller (Bluetooth) is modified, enabling an SGX app to create a secure binding with the Controller directly.

2.9 Summary

In this chapter, we provide an in-depth analysis of Bluetooth and various challenges in realizing Trusted I/O for Bluetooth. In response to those challenges, we present

BASTION-SGX: a Trusted I/O architecture for Bluetooth on SGX. We also discuss our prototype implementation of BASTION-SGX, which adds new, lightweight features to the Bluetooth Controller and demonstrates its utility in securing user data input from keyboard devices.

Our prototype examines an implementation of BASTION-SGX in a real-world case study that effectively mitigates a priveleged keylogger malware. We implement our solution primarily in firmware; to reduce performance costs, we recommend implementing some (or all) of these operations in hardware. We also present an analytical evaluation of BASTION-SGX's impact on memory and network performance. Because the the maximum application-layer throughput supported by Bluetooth is no more than a few kilobits per second, we show the Trusted I/O-related cryptographic operations will not introduce any perceptible latency and will certainly not have any impact on the throughput.

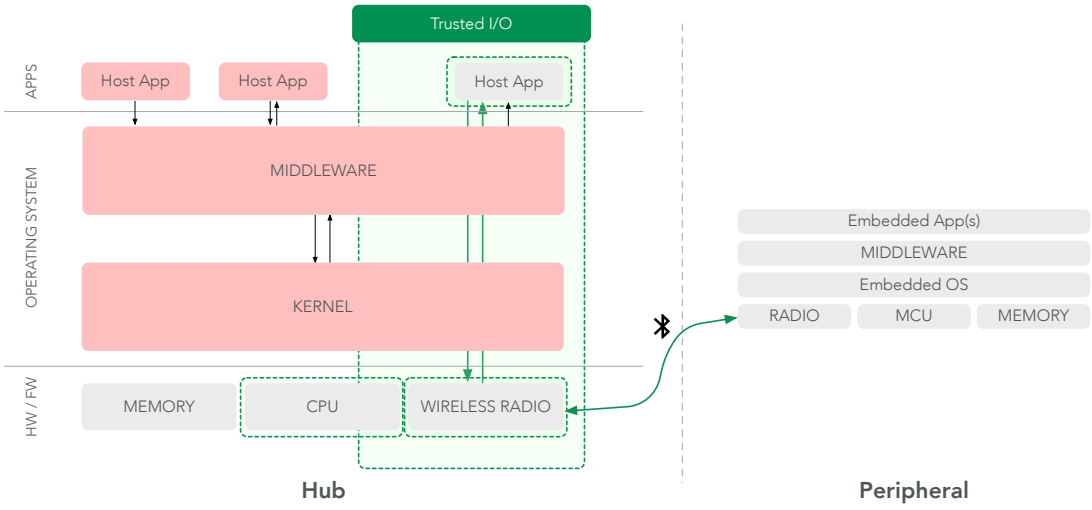

Figure 2.17: An overview of our system model and the threats to sensitive I/O data hub devices illustrated alongside our solution.

3

Designing Trustworthy Peripheral Devices

In Chapter 1 we introduced the idea that many peripheral devices are expected to be useful personal devices, to be secure against threats (such as malicious applications), and to do all of this "on a budget." In this chapter, we present our work on Amulet: a software and hardware platform that enables developers to create secure and efficient mobile health (mHealth) applications on resource-constrained devices, such as wearable devices and other peripheral devices; our work focuses on how these devices and their software can be composed and verified in a more trustworthy

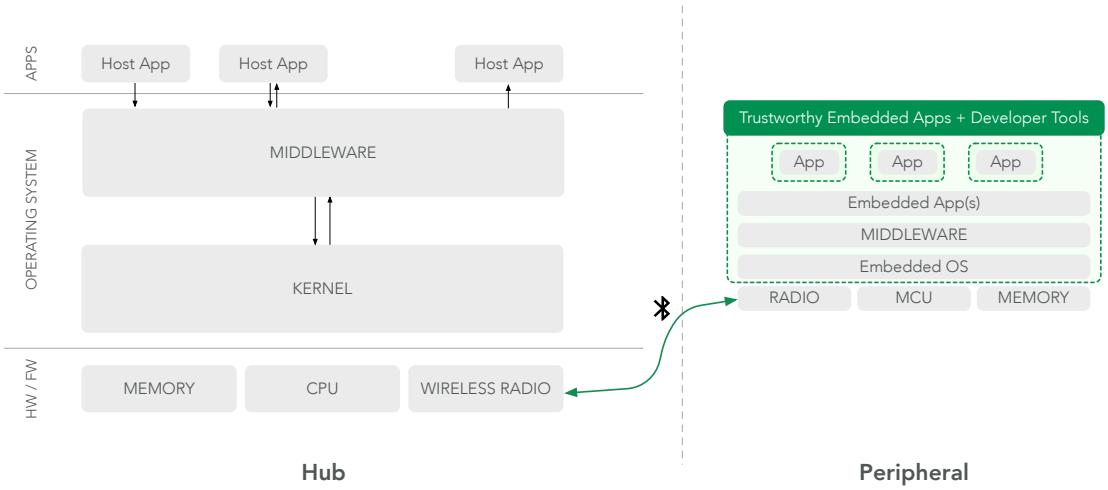

Figure 3.1: Overview of System Model and Amulet.

way, fortifying them from being compromised by errant applications that might attempt to interfere with other applications or the underlying system itself.

Although the Amulet Platform has potential to support a broad range of applications, on a broad range of devices, we focus our design on mHealth applications that run on a smartwatch form factor because they are increasingly prevalent, and their need for a robust, secure, long-lived platform poses important design challenges. While we focus our attention on mHealth applications on smartwatches, our contributions (Figure 3.1) can be generalized to any embedded platform (including many IoT devices) that needs (1) to support the secure operation of a device with one or more third-party applications, and (2) do so with extremely low power consumption.

This chapter is a modified version of two of our papers that were published in 2014 and 2016, respectively: *Amulet: A secure architecture for mHealth applications for low-power wearable devices* [139], and *Amulet: An Energy-Efficient, Multi-Application Wearable Platform* [91].

3.1 Introduction

Wearable wristbands are increasingly popular devices for health and fitness sensing, and the increasing variety of applications is driving the market from single-function devices (like the Fitbit Flex) toward multi-application platforms (like the Apple Watch or Pebble Time). These devices enable new sensing paradigms; they are worn continuously, they can provide at-a-glance information to the wearer, and they can interact through a body-area network with computers, smartphones, and other wearables (such as an ECG chest strap) or even *implantables* (such as an implanted insulin pump). Some existing devices are flexible and full-featured, with supportive development environments, but have inadequate battery life (about a day). Most others are single-purpose devices with better battery life that users cannot easily reprogram or customize for different applications and conditions.

Although the line between "smartband" and "smartwatch" products is blurring, we think of the former as having great battery life but limited flexibility, and the latter as having programmability but limited battery life. Battery life is a critical feature for mobile and wearable devices – by far the most-important feature as rated by users of today's smartphones and wearable gadgets [23, 182]. We aim to enable devices that have the week-long or month-long battery lifetimes of a smartband with the multi-application flexibility and full-featured development environment of a smartwatch. To support multiple applications, especially in critical domains like health, the platform must also provide strong security properties, including isolation between apps. To realize these goals, wearables must effectively manage energy, share resources, and isolate applications on low-power microcontrollers that cannot support hardware memory management units (MMUs).

In this chapter, we present Amulet, an open source[1] hardware and software

[1]You can find the open-source, open-hardware release of the Amulet Platform and its tools at https://github.com/AmuletGroup/amulet-project.

platform (Figure 3.2) for developing secure and efficient mHealth applications on resource-constrained devices. (Detailed illustrations of the relevant software architectures are shown in Figure 3.3 and Figure 3.5; an illustration of the hardware architecture is shown in Figure 3.7; images of our hardware prototype are shown in Figure 3.8.) This platform, which includes the Amulet Firmware Toolchain, the Amulet-OS Runtime, the ARP-View graphical tool, and our custom hardware design, efficiently protects applications from each other without MMU support, allows developers to interactively explore how their implementation decisions impact battery life without the need for hardware modeling and additional software development, and represents a new approach to developing long-lived mHealth applications.

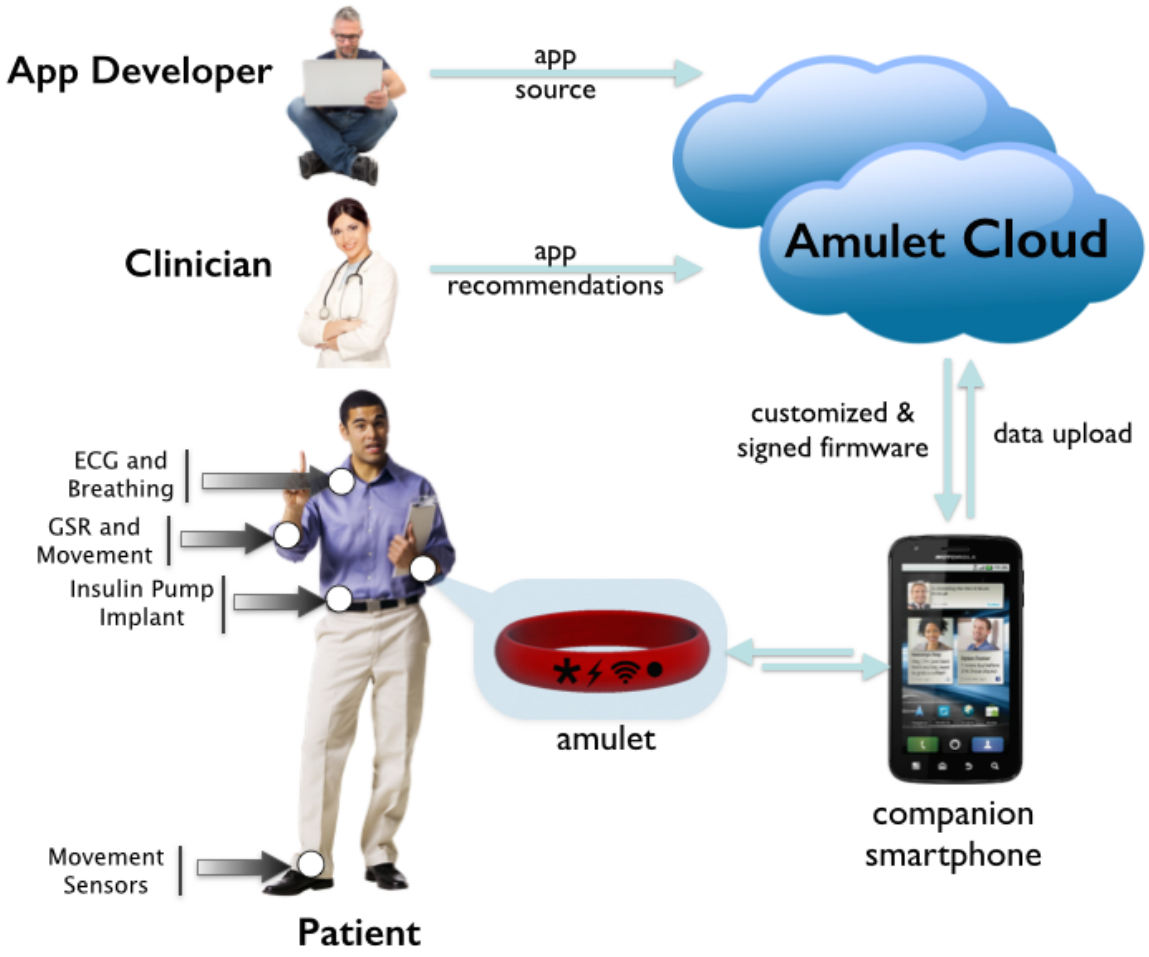

Figure 3.2: An overview of the Amulet Platform and vision. Figure adapted from its initial publication [181].

Devices with the properties that the Amulet Platform offers are of great interest to research communities in sensing, health, and other domains. In fact, we note that the Amulet Project has stimulated follow-on work by others including developing mHealth applications [37, 38, 39], new techniques and systems to secure mHealth data [80, 88] (between clinician and patients, or coaches and athletes, for example), user studies and human-computer interaction (HCI) studies [137], sensor design [159], further enhancements to the underlying design and security of the platform [106, 86], and patents [109].

3.1.1 Contributions

The Amulet work that we present in this dissertation was done as part of the Amulet Project: a multi-year, multi-disciplinary collaboration between researchers from Dartmouth College and Clemson University.[2] As such, and for the purpose of this dissertation, we understand that it is necessary to distinguish *my* contributions from the contributions of others on the Amulet Project. The contributions of the platform as a whole have been discussed in our past work [139, 91]. For clarity, a break-down of my contributions on this project are detailed next.

As a member of the Amulet Project (2014-2016), I made significant contributions to the design, implementation, and evaluation of the system and its security properties. Specifically, this dissertation highlights the following contributions:

1. Collaborated with others to design the Amulet software stack (see Figure 3.3), including the "Amulet APIs" used by applications, "Core Services" offered by the platform, and the interface with the physical components ("Board Support"). Based on the Amulet software stack, I collaborated with others to implement the Amulet-OS Runtime, a multi-application runtime system for resource-constrained wearables that is built on a low-power variant of the QP runtime [114].

2. Collaborated with others to design and implement the Amulet Firmware Toolchain (AFT), a firmware-production toolchain that guarantees application isolation (Section 3.3.3 and Section 3.5). Though work on what is now the AFT was started by Andrés Molina-Markham (Dartmouth), Bhargav Golla (Clemson), and Jacob Sorber (Clemson), I re-wrote and significantly extended their initial work. Specifically, I implemented the Resource Profiler (which

[2] Many people that I refer to in this section have moved on to other schools or companies. For this reason, I note their original affiliation.

profiles system and per-app memory and energy usage), the ARP-View (an app development tool that visualizes the Resource Profiler's findings for app developers), the Authorization Module (which performs compile-time analysis of apps according to Amulet security goals), and the compiler translations (which inserts lightweight runtime checks for apps, among other things).

3. Collaborated with Josiah Hester (Clemson) and David Kotz (Dartmouth) to design the resource models (Section 3.4). Based on these resource models, I implemented the Resource Profiler to aggregate the data needed to compute the models, and implemented the models in the ARP-View.

4. Collaborated with Josiah Hester (Clemson), Tianlong Yun (Dartmouth), and Andrés Molina-Markham (Dartmouth) to evaluate a prototype Amulet. Worthy of note is the evaluation of battery lifetime, the accuracy of the Resource Profiler predictions, and the system overhead (Section 3.6). This work also led to two user studies: one investigating the utility of our ARP-View tool to developers while developing a continuous sensing app, and one investigating the feasibility of using Amulet as a platform for continuous monitoring applications in human subject research. While I did not directly participate in the running of either study, I designed and implemented the tool studied in the former, and supported the app developer of the latter.

In all of this work, I would be remiss if I did not specifically acknowledge the exemplary work of fellow doctoral-student researchers, Josiah Hester (Clemson) and Tianlong Yun (Dartmouth), who also made significant contributions to the overall design, implementation, evaluation – and ultimately – success of the Amulet Platform. Furthermore, I acknowledge the efforts of George Boateng, Eric Chen, Emily Greene, David Harmon, and Anna Knowles – Dartmouth undergraduates with whom I worked – that helped to demonstrate the viability of the Amulet

Platform through developing interesting apps and uses of Amulet. Last, but certainly not least, I acknowledge the direct leadership and technical support of David Kotz (Dartmouth), Jacob Sorber (Clemson), Ryan Halter (Dartmouth), Ron Peterson (Dartmouth), Andrés Molina-Markham (Dartmouth) – without whom this project would not have achieved the success that it did.

3.2 Background & Motivation

In this section we provide relevant background on mHealth wearable devices and mHealth applications.

The current generation of mHealth wearables, such as the Fitbit Flex and the Withings Pulse, are single-application devices that focus on specific health goals like physical activity or sleep quality. These devices run one application, created by the device developers; they cannot run multiple applications nor be extended with applications from third-party developers. As a result, an individual with multiple health-monitoring goals may need to wear many such devices. Amulet intends to encompass many of these single-application devices by combining input from a variety of internal and external sensors such as heart rate, skin conductance, physical activity, and air quality. Amulet would allow interested adults to customize their wearable device with the applications that are relevant to their health needs and lifestyle. Meanwhile, "smartwatches" like the Apple Watch and the Samsung Gear are general-purpose wrist wearables that support multiple applications and third-party developers. Neither class of devices address our goals, for several reasons.

First, we are not convinced that all users want a general-purpose large-screen smartwatch with a battery life measured in hours. Our architecture aims to enable smaller wristbands (and other constrained wearables) with battery lifetimes mea-

sured in weeks or months and support for critical and sensitive applications like those related to chronic disease and behavioral health.

Second, developer tools for these wearables are in their infancy. Battery lifetime (i.e., amount of time between battery charges) is a critical concern for any wearable; although some developer frameworks provide general guidelines for writing efficient applications, developers are unable to accurately predict how their applications will perform when deployed. The Amulet Platform includes tools that forecast battery lifetimes and an application's resource usage. More importantly, these tools help developers conceptualize how their design decisions impact energy consumption and identify specific opportunities for improvement.

Third, current solutions do not provide open-source hardware and software; the Amulet Platform is fully open-source and open-hardware, enabling new opportunities for innovation by health and technology researchers alike.

Finally, the Amulet platform is noteworthy for its focus on security. While a device that can run third-party applications is inherently more difficult to secure than unmodifiable single-application devices (i.e., many IoT devices today), most current wearable devices that run third-party applications are not designed with security in mind (for example, the Pebble Watch). Newer wearables, including most smartwatches, provide limited support for third-party applications and run a heavy-weight operating system (such as Android's Wear OS or The Linux Foundation's Tizen, for example) at high energy cost and with little emphasis on security. Our proposed software architecture, Amulet, can run multiple third-party mHealth applications simultaneously and provides strong security properties.

3.3 System Overview

In this section we discuss our goals for the Amulet Platform, and outline the two major building blocks of the Platform: the *Amulet-OS* and the *Amulet Firmware Toolchain* (AFT).

3.3.1 Amulet Goals

We designed Amulet to support a multi-developer, multi-application vision, aiming at four goals not faced by single-purpose wearables, single-developer wearables, or power-hungry platforms that need to be recharged daily.

Goal 1: Multiple Applications. *Amulet platforms enable sensing applications written by third-party developers,* even on resource-constrained wearable devices. The Amulet Platform masks the complexity of embedded-system development, and supports a variety of internal and external sensors, actuators, and user-interface elements. Since users are unlikely to wear multiple single-function devices, the Amulet Platform aims to support multiple concurrent applications.

Goal 2: Application Isolation. *Amulet platforms isolate applications from each other and from the system.* With multiple concurrent applications, sensitive user information must be protected and applications must be prevented from interfering with the system or other applications. Amulet uses creative compile-time and run-time isolation mechanisms to achieve these properties on ultra-low-power microcontrollers that do not provide memory virtualization or memory protection. In this dissertation we focus on memory isolation and resource management.

Goal 3: Long Battery Life. *Amulet platforms enable wearable devices with battery life measured in weeks.* Today's multi-application wearable devices have poor battery life, including research devices like ZOE [113] and commercially significant devices like the Apple Watch [13]. Even the longest-lived commercial devices, like

the Pebble [150], have lifetimes measured in days. When wearables can run for weeks or months, new applications are enabled and users are likely to benefit from applications that support long-term 24/7 health monitoring and interventional behavior change. The Amulet hardware supports ultra-low-power operation and longer battery lifetimes, and the Amulet Platform helps developers see how their application's behavior influences battery lifetime.

Goal 4: Resource-usage Prediction. *Amulet platforms include tools that provide interactive analysis of resource usage*, including energy impact and memory usage for applications and the underlying system. Existing tools for third-party application developers on wearable platforms are very limited, focused on documenting best practices and measuring resource usage of running applications; they do not provide compile-time, app-developer tools for predicting the battery impact of an application or combination of applications.

We next describe our initial implementation, focusing on the above goals in two parts: the Amulet-OS and the Amulet Firmware Toolchain (AFT).

3.3.2 Amulet-OS

The Amulet-OS software architecture (Figure 3.3) achieves all the above goals by providing a low-power, event-driven programming model, a rich API, and efficient application isolation and optimization through compile-time and run-time techniques.

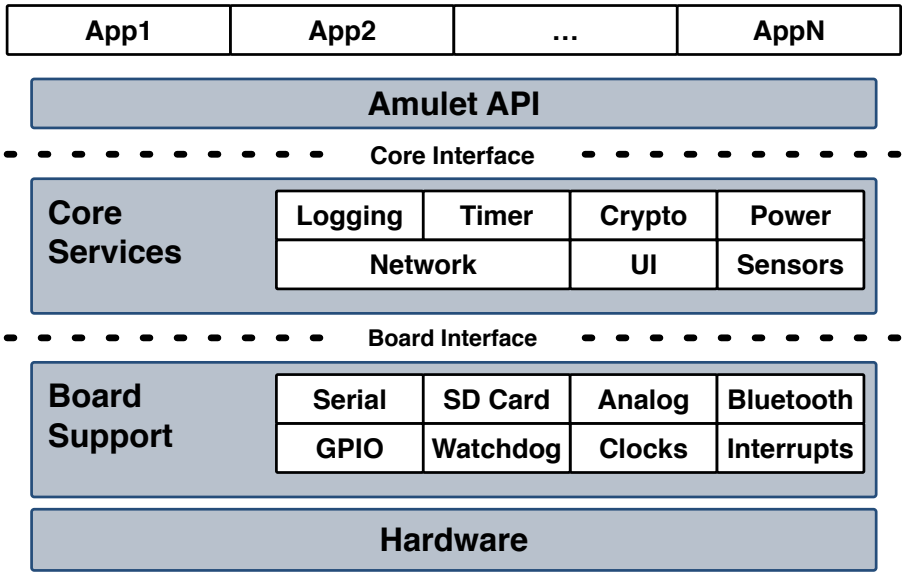

Figure 3.3: The Amulet-OS software stack; applications access core services through the Amulet API layer.

Event-driven programming

Many sensing-based applications, including the health-oriented applications that motivate our work, tend to remain idle waiting for user interaction or new sensor data. Thus, Amulet uses an event-driven programming model to simplify developer tasks and enable low-power operation. Each application is represented as a state machine with memory; that is, each application consists of a set of *states* (the boxes shown in Figure 3.4) and *transitions* between states (the arrows shown in Figure 3.4), as well as a small set of persistent *variables*. Each transition is triggered by the arrival of an *event*, which themselves result from expired timers, user interactions like a button press, or data arriving from internal and external sensors. Applications can specify optional *event handlers* for each state and each event type. Handlers are non-blocking functions that may consume data arriving with the event, update application variables, call Amulet APIs, or send events, in any combination.

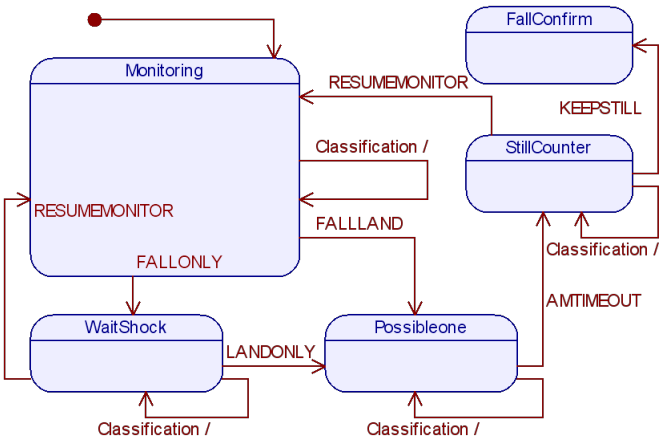

Figure 3.4: An example state machine for a simple event-driven application.

This state-machine approach makes application state explicit, easing analysis and optimization. Application code, state, and variables are kept in persistent storage. Handlers run to completion, so there are no threads with stack-based state information to preserve between events, let alone across processor reboots. The Amulet-OS leverages this simplicity for deep power savings; when there are no events to handle, the processor can go into deep sleep or even shut off. The Amulet Firmware Toolchain leverages this structure to enable the analysis and profiling tools described in the next section. This approach is also a major advantage over the alternative of running a larger operating system, such as embedded Linux or a real-time OS, in which applications are represented as processes or threads with complex state and limited opportunity for deep sleep. (For more information, see Section 3.7.)

Amulet API

Amulet-OS provides an application programmer interface (API) that allows for sensing, storage, signal processing, communication, timing, and user interaction. As with any OS, this API provides abstraction (hiding low-level complexities like interrupt vectors, analog-digital converters, and radio communication) and resource

management (isolating apps from each other and allowing them to share data and devices). Amulet-OS simplifies data gathering by providing applications the ability to *subscribe* to internal and external sensors; multiple applications can share a single data stream, each receiving an event when new data arrives. Amulet-OS also includes a logging API so applications can log sensor information to files on an internal microSD card, and a timer API so applications can arrange for an event in the future. Finally, the API provides applications access to interface elements (display, LEDs, buzzer, buttons, and capacitive touch, in our implementation) and multiplexes access across apps. These APIs call into the Amulet-OS core services shown in Figure 3.3. All such calls are non-blocking because event handlers must run to completion; where needed, a response is delivered to the application later as an event.

The Amulet-OS is more of a lightweight run-time system than an operating system, but nonetheless supports multiple applications on a low-power microcontroller without memory protection or management. Thus, Amulet uses compile-time analysis to support application isolation and access control, and to minimize its memory footprint. These challenges are the focus of the Amulet Firmware Toolchain.

3.3.3 Amulet Firmware Toolchain (AFT)

The Amulet Firmware Toolchain (AFT), shown in Figure 3.5, manages the analysis, translation, and compilation of firmware. By building a custom firmware image for each user, the AFT can optimize the image for the user's device and its applications. Here, we focus on two critical AFT roles: application isolation and resource profiling. First, the AFT ensures that applications can only access Amulet hardware by sending a well-formed request to the Amulet-OS core via the Amulet API, and prevents malicious or buggy applications from reading or modifying the memory of either the OS or another app. Second, with the AFT's profiling tools an application

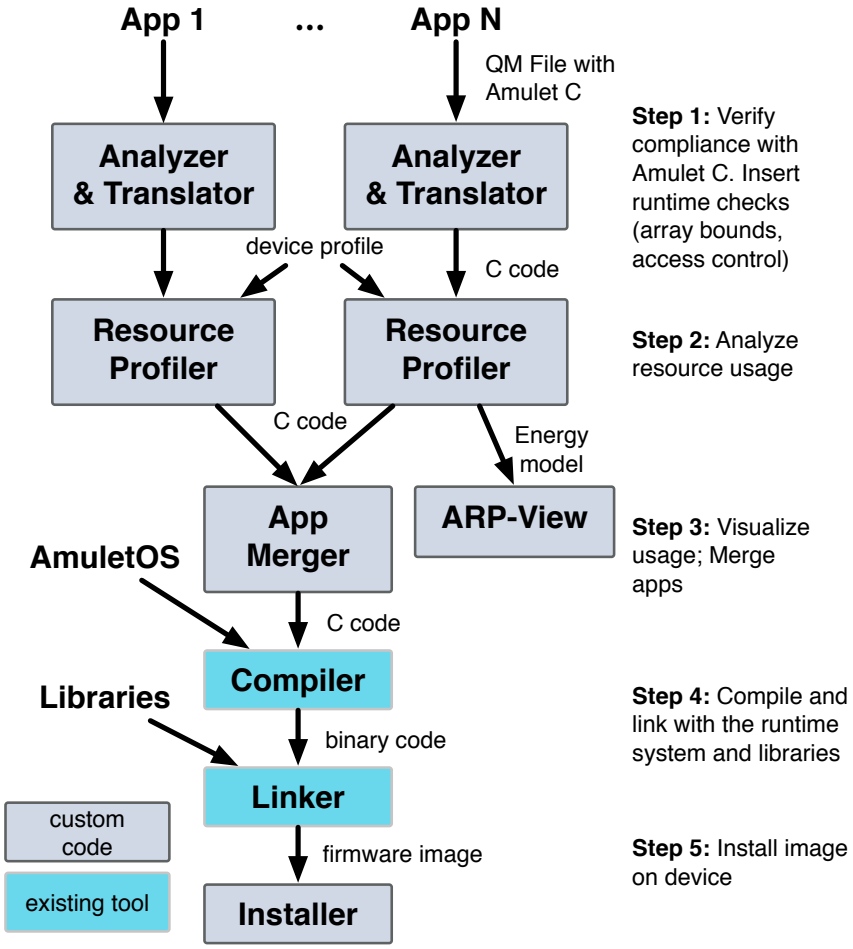

Figure 3.5: Architecture of the Amulet Firmware Toolchain, showing the steps in producing the firmware image for a given Amulet device.

developer can predict her application's resource usage. The following text refers to the numbered steps illustrated in Figure 3.5.

Analysis and Translation

Our approach leverages compiler-based translation and static analysis: application developers pass their code to the Amulet Firmware Toolchain, which translates and analyzes the source code, rejecting any code that is either not well formed or

violates the isolation principle (Goal 2). Application designers implement Amulet state machines using a simple variant of the C programming language, "Amulet C," which offers programmers familiar programming constructs and facilitates efficient code generation, while excluding many of C's riskier features (Section 3.5.4). These modifications, and the addition of loop invariants and automatic annotations by the AFT, allow rigorous analysis of an application's memory safety. The AFT uses static-analysis tools to examine the code to identify memory-safety violations and present the developer with compile-time errors. The AFT inserts run-time validation code where static analysis is inconclusive; for example, the AFT inserts code for array-bounds checking when the array index is not computable at compile time.

The Translator also implements authorization policies that define which applications can access which resources. For example, these policies may say that a fall-detecting application may subscribe to data from the accelerometer, that an emergency-response application may write a file to the SD card, or that the EMA application may use the buttons and the display. By addressing these policies at the time of application translation and analysis, the AFT can flag illegal access at compile time, and insert run-time checks only when needed. The resulting code is smaller, faster, and safer. Along with the applications, a list of application services are supplied in a configuration file. The AFT uses this list to determine if applications call functions that are not permitted, and to determine which set of drivers need to be included for this set of apps, to reduce code size.

Amulet Resource Profiler (ARP)

Although our experiments in Section 3.6 characterize the performance of the Amulet hardware and software under a representative application workload, and those results show it is possible to develop efficient applications for the Amulet hardware, new application developers may not have access to the inner Amulet hardware or

to our measurement infrastructure. With effective compile-time tools that predict an application's resource usage, developers can effectively manage critical resources and protect the user's experience even when combining applications from many developers.

The Amulet Resource Profiler (ARP) tool leverages the other modules in the Amulet Firmware Toolchain to predict an application's resource usage; here, we focus on memory and energy. Profiling these resources gives insight into the application's impact on battery lifetime, and the application's impact on constrained, shared, device computing resources. Lifetime is the most important consideration for a wearable device; thus, giving developers insight on how their design decisions affect lifetime is of great value to the developer and the user. The ARP captures information about each application's code space and memory requirements, using a combination of compiler tools and static analysis. To profile energy, the ARP builds a parameterized model of the application's energy consumption, as described in the next section. The results are then exported for use by the ARP-View.

ARP-View

This interactive tool presents developers a graphical view of the resource profile and sliders that allow them to immediately see the battery-life impact when they adjust application parameters. The tool guides developers towards a better understanding of their application and enables them to explore trade-offs of different design decisions. See Figure 3.6 for a snapshot of the ARP-View in action.

In ARP-View, developers are presented with an annotated visualization of their application that resembles their original finite-state machine. Annotations on edges (representing state transitions) provide immediate feedback about the rate in which that event handler executes and the energy usage of that particular event handler. In addition, ARP-View includes a set of 'sliders' that can be individually

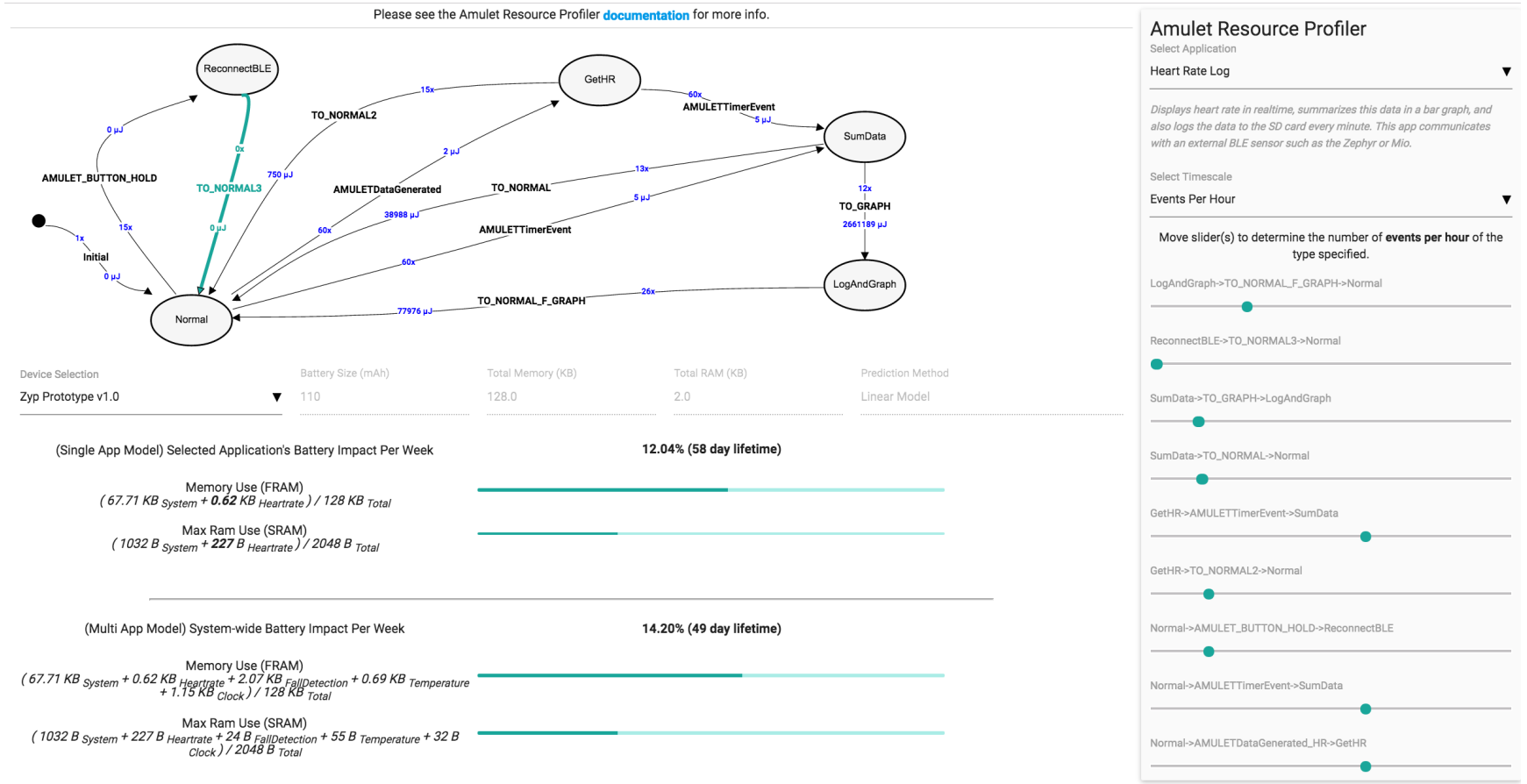

Figure 3.6: Screenshot of the ARP-View tool. A selected application is rendered on the screen where the transitions have been annotated with both the estimated cost of executing that transition and the rate at which the transition executes. The sliders along the right-hand side of the window allow the developer to adjust the rate of specific transitions. The bottom area of the screen displays information about an application's impact on the overall battery lifetime per week and memory usage. To view a full-size interactive demo of the ARP-View, please visit https://arpview.herokuapp.com/.

scaled to simulate a particular handler running more/less frequently (R_t); this allows developers to evaluate the impact of specific application activities on the overall battery lifetime.

For perspective, ARP-View imports the device profile for information about the device (such as battery and memory capacity) obtained earlier by the Profiler; see the next section for more details.

Merge, Compile, Link

The next step in producing a final firmware image, referring back to Figure 3.5, is for the AFT to merge the desired set of applications, then compile and link them with the Amulet-OS code. For Amulet we leverage an event-driven programming framework (QP) [114] that automatically generates the C code for state transitions from each application's state diagram. The AFT Merger combines this C code with the translated application code (now in C), and the underlying event-driven application framework, into a single C file. This file is compiled and linked with the Amulet-OS code, incorporating only the system components needed for this particular set of applications. (We anticipate users will select applications for their personal Amulet device from a store hosting applications from many developers – an "App Store" – and which compiles a custom firmware image comprising their selected applications.)

The result of the AFT pipeline is a single firmware image for upload or distribution. The AFT is currently agnostic about methods for delivery of the firmware; we acknowledge that the delivery method (perhaps through the above-mentioned App Store or some other cloud service) involves complex security and privacy concerns.

Table 3.1: Model notation.

Device profile	
M_d	storage capacity of data memory
M_c	storage capacity of code memory
E_B	energy capacity of full battery
E_ℓ	average energy consumed by a line ℓ
E_f	average energy consumed by one call to API function f
P_0	baseline power draw
P_s	average power draw for subscription to sensor s
S	the set of all sensors on this device
F	the set of all API functions available in Amulet-OS
Application parameters	
A	the set of all applications on this device
a	an application in the set of all applications; $a \in A$
f	a function in the set of all Amulet API functions; $f \in F$
F_a	the set of all API functions used by application a; $F_a \subseteq F$
s	a sensor in the set of all sensors; $s \in S$
S_a	the set of sensors used by application a; $S_a \subseteq S$
t	transition t in the set of all application transitions, T
T_a	the set of all transitions in application a; $T_a \subseteq T$
L	the set of all lines of code outside Amulet-OS
ℓ	line of code in the set of all lines of code; $\ell \in L$
$N_{\ell,t}$	number of times line ℓ is executed by transition t
$N_{f,t}$	number of times function f is called in transition t
R_t	rate transition t executes (transitions per second)
Energy estimates	
E_t	estimated energy of each occurrence of transition t
E_a	estimated energy consumed by application a
E_A	estimated energy consumed by set of applications A

3.4 Resource Model

As noted above, the Amulet Resource Profiler (ARP) constructs a predictive energy and memory model for each application. For clarity, we first discuss the modeling and prediction for a single application (Section 3.4.2), and then conclude this section with an extension of the model to accommodate multiple applications (Section 3.4.3). Before delving into this discussion, we detail the assumptions we make in our

resource model. The notation for our model is summarized in Table 3.1.

3.4.1 Model Assumptions

To predict resource usage we make the following assumptions in our resource model:

1. The baseline power P_0 captures the baseline power draw of our custom board, the display, the buttons, the scroll wheel, the haptic buzzer (off), and the LEDs (off).

2. The radio chip consumes its baseline power when it is inactive; all additional costs of its activity related to external sensors, and its internal accelerometer, are captured by the relevant sensor-subscription power draw P_s. The radio chip incurs no other energy.

3. The application chip consumes its baseline power when it is inactive, and we assume it is sleeping whenever no application code is running. All other app-board energy is captured by API calls E_f, lines of code E_ℓ, or sensor subscriptions P_s.

4. All powers P are an average power draw over time.

5. Power drawn by input devices (buttons, scroll wheel) are included in the baseline power.

6. Energy consumed by active output devices (LED, buzzer) costs are captured by the E_f where f is an API function using those devices.

7. Display (static) cost is buried in the baseline P_0.

8. Display (change) cost is captured by the display-affecting API calls.

9. We currently ignore data-dependent conditional statements and loop conditions.

10. $N_{\ell,t}$ includes lines of code in the exit block of the source state, the main block of the transition, and the entry block of the destination state; all three blocks execute (in sequence) when the transition occurs.

3.4.2 Single Application Model

The ARP proceeds in four phases, which we describe next.

Phase I: Import Device Profile

The ARP imports a *device profile*, specific to the target Amulet model but independent of any particular application. The device profile lists the amount of energy consumed for each API call and for other fundamental operations, based on empirical measurements collected earlier on a given hardware and system software configuration. Each Amulet component (sensor, user-interface element, storage, and processor) must be represented in the model as an instantaneous energy cost. Each component has different states, each drawing different amounts of power, that must also be captured. The energy cost of each API call must be captured in the form of average power.

Altogether, the device profile includes information about the device capacity (memory, battery), empirically derived measures of average energy consumed E_f for each Amulet API function f, the average energy consumed E_ℓ for executing a specific line of C code ℓ; and the average power draw P_s for a subscription to sensor s (see Table 3.1). Energy is measured in joules (J); power draw is measured in watts (J/s).

A device profile would be prepared and provided by the Amulet manufacturer with each new hardware and Amulet-OS release. The AFT (Section 3.3.3) can assist

by automatically producing the code to create this profile by generating a specially instrumented *Amulet Device Profiler app* that exhaustively tests each of the Amulet API functions that draw significant amounts of energy, for example, sampling the Gyro, writing to the SD card, or turning on the radio. Using simple monitoring hardware, Amulet manufacturers can gather these statistics once and distribute the profile to application developers for their own testing, similar to current Android manufacturer practice [9].

Phase II: Analyze Code

The ARP examines the application's state diagram – a graph in which nodes represent states and directed edges represent transitions from one state to another. The result is a set of transitions T_a for application a. For each transition $t \in T_a$ the ARP identifies all non-system code executed when transition t occurs (using static analysis to count the number of executions $N_{\ell,t}$ of each line of code ℓ, summing across loop iterations and recursively examining code in helper and library functions). That is, $N_{\ell,t}$ counts the number of times line ℓ will be *executed* during the handling of transition t, accounting for loops and function calls.[3] Similarly, for each transition t the ARP determines the number of times the code for transition t will invoke each Amulet API function f, which we denote $N_{f,t}$. Finally, the ARP examines the sensor-related API calls to identify the set of sensors S_a to which application a subscribes. The constraints of Amulet C (no recursion, no pointers, no dynamic memory allocation) make this static analysis feasible.

[3]We count lines of code as a proxy for code complexity; to improve accuracy we could use the code generator to count instructions of assembly. Since instruction execution has a relatively small impact on power consumption, our implementation assumes E_ℓ to be the same for all lines of code; we focus on modeling the API calls and sensor usage.

Phase III: Construct Model

The ARP constructs a parameterized model of the total energy cost for the app. For each transition t, it estimates the average energy consumed E_t for an occurrence of that state transition, incorporating the cost of executing the code and API calls in that transition:

$$E_t = \sum_{\ell \in L} N_{\ell,t} E_\ell + \sum_{f \in F} N_{f,t} E_f \tag{3.1}$$

If the app subscribes to any sensors to feed it sensor data, we must also account for their average power draw:

$$\sum_{s \in S_a} P_s. \tag{3.2}$$

Finally, the Amulet hardware incurs a baseline power draw when it is inactive; we use P_0 to represent the average power draw of the baseline system (the microcontrollers, the display, and the input devices). Because Amulet-OS has no background activity, this baseline power draw represents all of the Amulet-OS power draw not captured in the above equations.

To estimate the total energy consumption for application a, the ARP needs to know how often each transition t will occur. While some of these rates may be discerned from static analysis on the code, for others the ARP needs advice from the developer – which the developer provides through annotations on the app's state machine. These rates R_t are the 'knobs' for the energy model – knobs the developer can tweak to explore the power draw for various design options (for example, the period of a timer that duty-cycles a key part of the application behavior). The total energy consumed for application a, over a time period τ, is thus predicted from the baseline power and the above equations, factoring in the rate of every transition t:

$$E_a(\tau) = \tau P_0 + \sum_{s \in S_a} \tau P_s + \sum_{t \in T_a} \tau R_t E_t \tag{3.3}$$

Over a week, then, application a consumes a fraction of the total battery capacity, $E_a(\omega)/E_B$, where $\omega = 1$ week; we leverage this calculation in our evaluation below.

Phase IV: Count Memory Usage

Every application requires memory for storage of its code and its data; like any embedded system, low-power wearable platforms have severely limited memory space. After parsing the application's code and generating its firmware image, the ARP reports the amount of memory to store the application's code and determines an upper bound on the amount of data memory consumed by the application. These numbers are presented as fractions of M_c and M_d. An application's data memory comprises global variables and local variables (on the stack). Amulet C does not allow dynamic memory allocation, so the ARP easily counts the size of all global variables; Amulet C does not allow recursion, so the ARP can compute the maximum stack depth (including local variables).

3.4.3 Extending the Model to Multiple Applications

The ARP is also capable of estimating the energy consumption for some mix of applications A. (Figure 3.6 shows one app's state diagram along with total energy consumption for all apps.) To estimate the total energy consumed by A, we cannot simply sum the energy consumed by all of the individual applications. That is, the total is not simply $\sum_a E_a$, because we need to avoid double-counting the baseline power draw as well as the sensor subscriptions (which are shared across all apps). Instead, we need to account for the union of all sensors used by the mix of apps: $S_A = \bigcup_{a \in A} S_a$. The total energy consumed for the application mix A is therefore estimated by a variant of Equation 3.3:

$$E_A(\tau) = \tau P_0 + \sum_{s \in S_A} \tau P_s + \sum_{a \in A} \sum_{t \in T_a} \tau R_t E_t \tag{3.4}$$

3.5 Implementation

We developed an Amulet reference device and implemented the Amulet-OS and Amulet Firmware Toolchain software described above. In this section we describe the details of each, as well as nine applications we wrote to demonstrate and evaluate the Amulet Platform.

3.5.1 Amulet Device Prototype & Ultra Low Power Operation

A detailed description of the hardware prototype and the low-power capabilities of the Amulet was discussed in detail in our SenSys'16 paper [91]; we refer the interested reader to Section 5 of our paper. Suffice it to say that the hardware prototype includes a variety of interesting sensors (e.g., microphone, light sensor, temperature sensor, gyroscope, accelerometer), storage options (e.g., SRAM, FRAM[4], microSD), and I/O components (e.g., buttons, capacitive touch sensors, LEDs, a haptic buzzer, a small display), and we use a variety of techniques to operate an MSP430FR5989 microcontroller (the main computational device where applications run) and a Nordic nRF51822 (where radio-based communications, such as BLE, are managed) in a way that enables us to achieve low-power operation. (The hardware architecture of the prototype is shown in Figure 3.7, and images of the perspective and interior views of our prototype are shown in Figure 3.8.) We exclude these details here as they are not necessary to understand the more security-focused work that we highlight in this dissertation.

[4]FRAM (Ferroelectric RAM) is an increasingly common non-volatile memory technology that offers persistent storage without power, and is 100 times faster than flash memory; during use, FRAM uses 250 times less power than flash (about one microamp at 12kB/s).

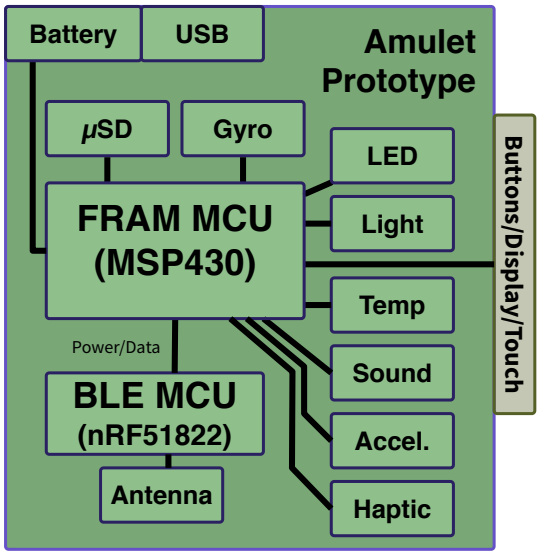

Figure 3.7: The hardware architecture of our two-processor Amulet prototype: the MSP430 runs applications, and the nRF51822 manages communication.

Figure 3.8: Perspective and interior views of our open-hardware wearable device (circa 2016), part of the open-source Amulet Platform. The platform supports development of energy-efficient, body-area-network sensing applications on multi-application wearable devices.

3.5.2 Amulet-OS

We implemented the Amulet-OS run-time system on top of the QP event-driven framework [114]. As shown in Figure 3.3, the Amulet architecture (and our implementation) has three major layers: (1) a board-support layer, running directly on the hardware and abstracting some of the hardware-dependent nuances; (2) a set of core services that provide core functionality like networking, time, logging, and power management; and (3) a set of application services accessible through a thin set of functions in the Amulet API. The AFT static-analysis tools recognize Amulet API functions and verify an application's authorization to use specific application services (much as Android uses the Manifest file to determine application permissions).

The Amulet-OS is fully event-driven: there are no processes or threads, so all application code runs to completion without context-switching overhead. Indeed, an application's code is comprised only of event handlers, and all such handlers execute quickly (enforced by static analysis and OS-imposed time limits, and enabled by non-blocking Amulet API functions). Quick interrupt and event handlers allow the system to stay in low-power sleep mode most of the time. For the purposes of our experiments below, and to support the Amulet Device Profiler app, Amulet-OS can toggle GPIO pins when executing application event handlers, Amulet API functions, or interrupt handlers; our external measurement chassis (Section 3.6) monitors these pins to derive detailed energy and temporal measurements about application and system modules.

3.5.3 Amulet Apps

We implemented nine applications to demonstrate and evaluate the Amulet Platform. Amulet application developers construct their applications using the QP

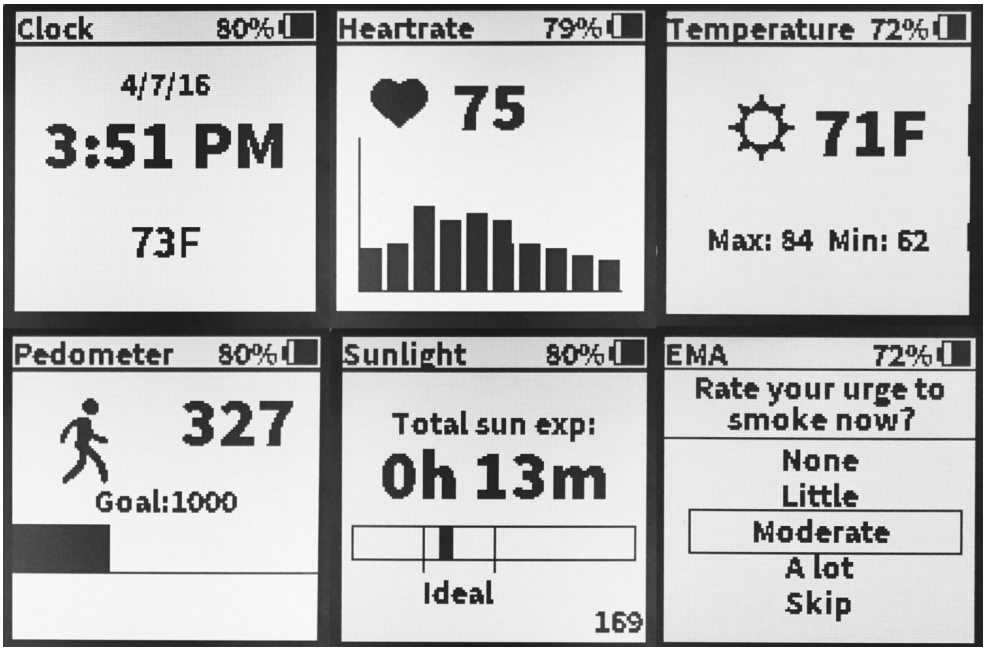

Figure 3.9: Screenshots of six of the nine applications we implemented.

event-based programming framework. Following QP, each application is defined as a finite-state machine; for each possible state, the application can respond to a set of events by providing a handler function for each type of event. We used vanilla QP with the non-preemptive kernel (version 5.3), and required applications to use the Amulet API to request services from Amulet-OS. Screenshots of six applications (apps) are shown in Figure 3.9.

The QP framework exports the application in the XML-based QM format, which embeds all of the programmer-supplied C code along with information about the application's finite state machine.

3.5.4 Amulet C

As mentioned earlier, application isolation is an important goal of the Amulet Platform. To achieve application isolation without the support of memory-management hardware, and without incurring excessive run-time overhead, the AFT conducts most application isolation at compile time. Applications (that is, their handler

functions) are written in a custom variant of C that removes many of C's riskier features: access to arbitrary memory locations (pointers), arbitrary control flows (goto statements), recursive function calls, and in-line assembly. Since array access in C is implemented using equivalent pointer operations, we modified the array syntax so that arrays can be passed to functions explicitly 'by reference' (not as pointers). In Amulet C, arrays also have an associated length that allows for run-time bounds checking whenever access behaviors cannot be adequately checked statically. Although this approach imposes some effort on the developer (to adapt their code for Amulet C) it allows us to estimate the runtime of code executed in the state machine, giving tighter bounds on energy predictions made by the Amulet Resource Profiler.

3.5.5 Amulet Firmware Toolchain

We implemented the AFT as a series of programs that translate, analyze, validate, and profile apps. Each application includes (1) a state machine, (2) event handlers (written in Amulet C), and (3) attributes specifying the application's global variables. The QP framework combines application information into XML-formatted QM files. AFT tools are written in Java and use its built-in XML libraries to parse the submitted apps. Our tools translate the Amulet C code to safe C code using a modified C grammar and the ANTLR parser generator [12]; they then use the Frama-C static analysis framework [30] to ensure that array and other memory accesses are valid, that problematic integer operations do not occur (e.g., division by zero), and that programming techniques such as recursion, goto statements, and pointers (including unary address operators) are not employed. Also noteworthy is the fact that the AFT tools verify an application's permissions based on an XML file that lists application services that the application is permitted to use. Violations against Amulet C coding rules and non-authorized requests to the core API trigger

AFT compile-time errors.

After all these steps, applications are merged together into a single QM file, which is then converted to C using QP. Immediately before the merge, all application resources (e.g., variables, handlers, helper functions) are isolated by mapping them to a unique namespace based on the application's name (no two applications installed on the system can have the same name). This code is then compiled and linked using Texas Instrument's open-source GCC for MSP430. This firmware image can then be installed onto the application chip (MSP430) of our Amulet device prototype.

3.5.6 Resource Profiler

We implemented the Amulet Resource Profiler (ARP) in Java and integrated it with the other tools of the Amulet Firmware Toolchain (AFT). After validation and translation, the ARP uses an ANTLR-generated parser to extract model parameters from the application's code and QM file. Specifically, from the application code it extracts estimates of lines executed per transition ($N_{\ell,t}$), sensor subscriptions (S_a), and Amulet API calls (F_a and $N_{f,t}$). From the QM file it extracts the state machine and its transitions (T_a), and the developer's annotations about transition rates (R_t). From the firmware's symbol table it extracts the code size for both the application and the core, and the amount of FRAM memory used by the application and the core.

Separately, the ARP uses the Amulet Device Profiler app, and the measurement chassis described in the next section, to extract the parameters for the device profile (Table 3.1). The Device Profiler application also informs the ARP about the scaling factors for certain operations, allowing for fine-grained estimates of function cost (for example, assigning lower energy costs to drawing a 2x2 rectangle as opposed to drawing a 128x128 rectangle). The Resource Profiler combines these measurements

to build a parameterized model of the energy cost for each application, following Equation 3.3. Our prototype hardware has two kinds of internal memory: SRAM used for the execution stack and local data, and FRAM used for code and global data; thus our ARP implementation reports on SRAM and FRAM usage (rather than code and data usage as in Section 3.4).

3.5.7 The Amulet Resource Profiler Developer View

Our developer-facing tool, ARP-View, leverages the ARP's fine-grained data about the structure and behavior of applications to (1) give developers insight into how certain user actions, sampling rates, and blocks of code consume energy, enabling developers to make concrete the links between certain parts of code and energy draw; and (2) provide meaningful battery-lifetime estimates for an application or suite of applications. The ARP-View currently presents a wealth of information including system and OS level details (e.g., FRAM available and its usage), sliders to adjust event frequency and view results in real-time, and the battery impact (percentage of battery consumed per week) and lifetime in days for a selected application as well as an entire suite of applications. Our implementation provides **real-time** feedback to the developer regarding application impact on the battery by running a daemon process that monitors application files and re-profiles applications upon detecting changes to those files; note that application translation, analysis, validation, and profiling all happens prior to actually compiling the source code and, thus, runs fast even on common laptop and desktop machines. Providing memory stats to the developer via the ARP-View requires compiling all of the application/system code and parsing the resulting binary, which incurs a noticeable – but small – amount of time (approximately 1-2 seconds) on common laptop and desktop machines; thus, we consider memory-usage feedback to the developer to be "near real-time."

3.6 Evaluation

In this section we evaluate the performance of Amulet-OS and the Amulet Firmware Toolchain against the goals from Section 3.3.1. Goal 1 (Multiple applications) and Goal 2 (Application isolation) are met by design of Amulet-OS and the Amulet Firmware Toolchain. We thus focus on an experimental evaluation of Goal 3 (Long battery life) and Goal 4 (Resource-usage prediction). We examine (1) the battery life of a typical Amulet device, (2) the accuracy of the Resource Profiler's predictions, and (3) the various sources of overhead.

3.6.1 Experimental Setup

A detailed description of the experimental infrastructure that allowed us to collect precise controlled measurements of the Amulet system (and its applications), without incurring measurement artifacts or overhead on the Amulet itself, is discussed in detail in our SenSys'16 paper [91]; we refer the interested reader to the first part of Section 6 of our paper. Suffice it to say that we developed nine applications (apps) for use in the evaluation of Amulet (Table 3.2), selected to represent a range of compelling applications and exercise many of the features available in the current prototype. Furthermore, we built a programmable chassis (and made minor modifications to system code) that allowed us to stimulate the user interface and trigger state changes in the applications under test. Thus, we could automatically and repeatably test the Amulet prototype under controlled and consistent conditions.

3.6.2 Battery Lifetime Under Various Application Workloads

In this section we quantify the average power draw of our Amulet prototype for multiple loads. We represent this power draw in terms of battery lifetime using

Table 3.2: Amulet applications used for evaluation.

Application Name	Description	BLE?	Sensors	Log?	UI elements
Clock	Display time of day and temperature	No	Temp.	No	Display, Buttons
Fall Detection	Detect falls using the accelerometer	No	Acc.	No	Display, Buttons
Pedometer	Record number of steps walked	No	Acc.	No	Display
Sunlight Exposure	Monitor exposure to light over time	No	Light	No	Display
Temperature	Monitor ambient temperature over time	No	Temp.	No	Display
Battery Logger	Monitor, display, and log battery statistics	No	ADC	Yes	Display, Buttons
Heart Rate	Monitor and display heart rate from BLE sensor	Yes	HR (external)	No	Display, Buttons
Heart Rate Logger	Monitor, display, and log HR statistics	Yes	HR (external)	Yes	Display, Buttons
EMA	Deliver surveys to users	No	No	Yes	Display, Buttons, Touch

the 110 mA h battery currently encased in the prototype. (Larger batteries up to 570 mA h are used in current smartwatch products [177].) We acknowledge that the accuracy of battery lifetime estimates depends heavily on battery wear, quality, leakage, and other factors. These estimates serve to place the actual measured power draw in an understandable form.

For our first experiment, we installed a firmware image containing a single application and measured the average power draw using the measurement chassis. We used the programmable chassis to emulate user sessions, where a 'session' lasted as long as it took to exercise states in the critical path of an application. We repeated this experiment for each of the nine applications, for a single session, determining average power by summing the energy of the session and dividing by the time. The results of this experiment are shown in Figure 3.10. This figure shows the power draw of an application was highly dependent on the hardware components used, and the frequency of their use, further motivating the use of ARP-View. Lifetimes for all of the apps exceeded two weeks.

On our current prototype, an application load comprising Clock, Fall Detection, Pedometer, Sun Exposure, and Temperature would allow an Amulet battery to last **more than four months**. With constant BLE communication to the heart-rate sensor (we used commercial Zephyr and Mio heart-rate sensors) our Amulet's power draw allows for nearly a **two-month** battery life. Two energy-hungry applications (Battery Log and HR Log) used significant amounts of energy logging data

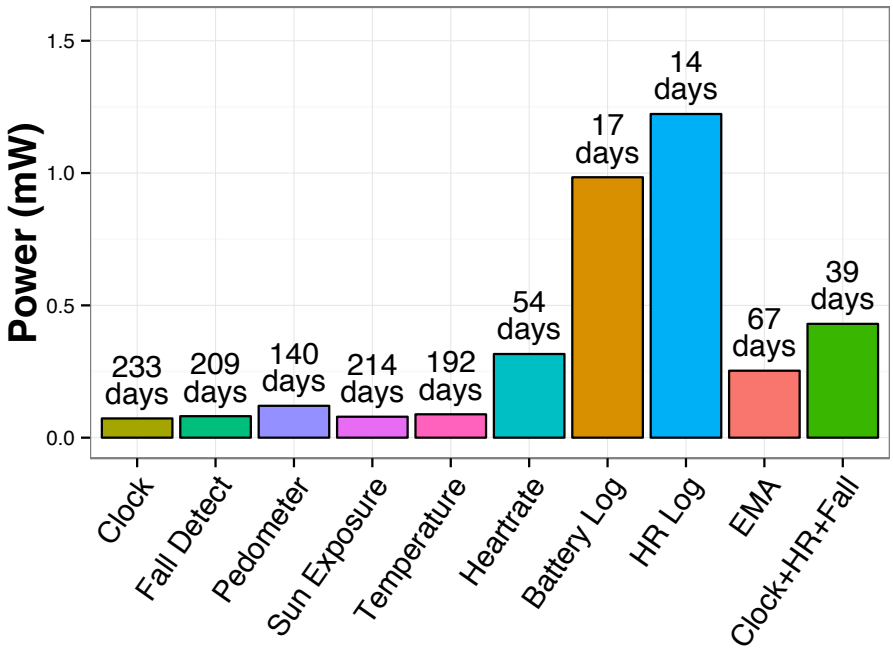

Figure 3.10: Average power draw for each app on the current prototype. The expected lifetime with the 110mAh battery is shown above each bar.

to the microSD card, drastically reducing battery life. The EMA application leaves the capacitive touch sensor on continuously, further reducing lifetime. By duty cycling the capacitive touch sensor and reducing logging operations or moving those operations to Bluetooth, lifetimes could be increased.

To determine the effect multiple applications have on the device lifetime, we assembled a firmware image that included three applications (Clock, Heart Rate, and Fall Detection), and measured the steady-state power draw as above. With this configuration, using the accelerometer, Bluetooth communication, and updating the display, the expected lifetime was **39 days**. In summary, an Amulet with applications using on-board sensors will last for many months, an Amulet using an external BLE sensor feeding regular heart rate data will last for 1-2 months, and applications making heavy use of the logging operations or capacitive touch features will last for a few weeks. This lifetime enables long-term usage for many application

domains. This lifetime is equal or better than the manufacturer-listed lifetimes of popular consumer platforms, such as the Pebble, FitBit, Apple Watch, and many others. (Direct comparisons are difficult, of course, because these products have different battery sizes and different purposes.) In any case, the Amulet Platform seeks to enable energy-focused development regardless of device. Our current prototype device sits comfortably in the high end of the lifetime wearable spectrum, enabling us to develop interesting apps with long lifetimes.

3.6.3 Accuracy of Amulet Resource Profiler Predictions

Recall that the Amulet Resource Profiler (ARP) tool predicts the effect of an application on the battery lifetime of an Amulet device. Specifically, ARP predicts the app's total energy cost per week, which can be used to determine the impact an application has on the battery lifetime. To evaluate the accuracy of these predictions, we compared the ARP battery-impact prediction, for each app, with the actual battery impact computed from the measured average power draw presented in Figure 3.10.

Each app's event frequencies were set to match the event frequencies used by our programmable chassis in collecting the measurements that resulted in in Figure 3.10. For some apps, specifically Clock, Temperature, and Sun Exposure, these event frequencies were gathered from the developer's code. These apps sense intermittently, on a timer whose value is set by the developer inside the code for a transition. These apps only respond to the timer, not to user or environmental input, so their energy impact depends completely on the timer value. Event frequencies were set carefully for Apps like Pedometer and Fall Detection, but their energy impact was dominated by the Amulet's baseline energy, paired with the number of times they compute over a series of acceleration values. Table 3.3 presents the predicted (and observed) percent impact on battery life, for each application, along

Table 3.3: ARP battery-%impact predictor.

Application Name	Obs. (µW)	Pred. (µW)	Error (µW)	Error (%)
Clock	72.73	77.45	4.72	6.1
Fall Detect	81.20	89.21	8.01	9.0
Pedometer	120.47	129.98	9.51	6.3
Sun Exposure	79.19	85.13	5.94	7.0
Temperature	88.37	92.89	4.52	4.9
Heart Rate	316.16	320.34	4.18	1.3
Battery Log	984.10	1089.63	105.53	9.7
Heart Rate Logger	1223.11	1318.68	95.57	7.2
EMA	253.07	265.05	11.98	4.5
Multi	430.01	438.86	8.85	2.0

with the percent error in the ARP prediction. (We quantify the error as the difference between expected average power draw and observed average power draw.) The results indicate that our Resource Profiler was reasonably accurate at estimating the battery life for these applications.

Applications can have different battery lifetimes depending on the amount of user interaction, the data rates or sensing schedules, the environment, and of course, the choices the developer makes in implementation. We capture four types of events in the ARP: (1) User interaction, (2) Data delivery, (3) Timers, and (4) Programmer-defined. While data delivery events are static (sensor subscription schedules are determined beforehand in our current system), each of the other three can be modified by the developer to explore the effect on battery lifetime. The ARP predictions are heavily dependent on the accuracy of the underlying device profile. Small measurement errors in generating the device profile can compound as event frequency increases. The ARP has trouble quantifying certain types of operations. For instance, the energy required for microSD card writes is heavily dependent on the size of the write. The ARP does not currently account for parameter length in SD writes, contributing to the higher error for applications that use SD functions

(Battery Log, HR Log). Nonetheless, in our experiments, the highest error rate was only 9.7%, and we expect it will improve.

However, most developers will not care if their application is predicted to last 90 days and instead lasts 80 or 100 days; they care about how the code they write, and the frequency of events, proportionally affect the lifetime. Of course, further tests 'in the wild', with a wider variety of apps, will be necessary to generalize this conclusion.

3.6.4 Amulet Overhead

We also conduct experiments to understand the overhead of the system.

Runtime

Energy consumption is significantly impacted by the fraction of time the application microcontroller (MSP430) is active; the rest of the time it can be in a low-power deep sleep mode. The MSP430 is active whenever it is running application code (handler) or system code (Amulet-OS and QP system code). (In modern operating systems, this is termed *system time*, whereas time spent running application code is termed *user time*.) The former time (system time) is *overhead*, from the application's point of view. To measure this overhead, we instrumented the Amulet-OS to trigger I/O pins whenever it (1) puts the system to sleep, (2) was active and executing Amulet-OS code, and (3) was active and executing application code. We then used our measurement chassis to obtain precise measurements of the time spent in each mode, for each of our applications, as shown in Table 3.4. For each application listed, we ran three short sessions while the application was conducting its normal duty cycle, but not being triggered by user-interface events. For all apps this meant polling sensors, and waking up for timer events. The low temporal overhead of

Table 3.4: Temporal overhead.

Application Name	%Sleep	%OS	%App
Clock	98.1	0.9	1.0
EMA	98.2	1.0	0.8
Heart Rate	91.1	0.9	8.0
Pedometer	93.8	2.2	4.0
Pedometer+HR	87.5	1.9	10.6
Pedometer+HR+Clock	85.4	2.8	11.8

0.9-2.8% confirms the efficiency of our approach.

Memory

Amulet-OS uses a portion of the limited memory space available to applications, limiting the quantity and size of apps that can be installed in a single firmware image. In our prototype, applications and the OS must share the limited FRAM memory space (128 KB). For a firmware image comprising five applications that used most of the functionality available, Amulet-OS consumed 55.91 KB of the 128 KB available FRAM code space, while applications consumed 14.48 KB; the OS consumes nearly half of the current FRAM. Meanwhile, Amulet-OS claims 1.078 KB of SRAM, leaving 0.922 KB of SRAM for applications; recall that apps use SRAM only for their execution stack, and only when actively executing an event handler; only one application is active at a time. Moreover, FRAM and SRAM are continuous on the MSP430; we anticipate making larger blocks of FRAM available (to be treated as RAM) to the app.

The memory and runtime overhead of our Amulet implementation – while sufficient to develop multiple interesting apps on constrained hardware – could be improved. Memory limitations could be sidestepped by adding 256KB of external FRAM on a secondary storage board to "swap" applications. We expect runtime overhead to further improve as we tune the display driver, which dominates the system overhead.

3.6.5 User Studies

Our SenSys'16 paper [91] also featured insights into two user studies: one investigating the utility of our ARP-View tool to developers while developing a continuous-sensing app, and one investigating the feasibility of using Amulet as a platform for continuous-monitoring applications in human-subject research. While I did not directly participate in the running of either study, I designed and implemented the tool studied in the former, and supported the app developer of the latter. Both studies were approved by our Institutional Review Board (IRB).

The takeaways from these studies are encouraging. Namely, the first user study involved recruiting ten students with at least some experience with low-power embedded systems. Participants were taken through an exercise of mapping out an embedded system application that required some sensing task. Subsequently, each participant was provided with our ARP-View tool and asked to repeat the exercise. Through a structured interview at the end of the study, the investigators found that our tool was helpful to developers, obtaining feedback that our tools provide rich insight into their decisions and how these map to energy cost within their application.

A second user study involved recruiting six medical-school students to participate in a study that was investigating the feasibility of using Amulet as a tool for conducting Ecological Momentary Assessment (EMA) studies – often used in behavioral-medicine research. This preliminary study was intended to evaluate user acceptance of the Amulet wearable, and technical feasibility of multi-source data collection (i.e., EMA, HR data, and battery stats) with an mHealth EMA application. While there were some issues with the prototype (specifically the device's case), this early study demonstrated the feasibility of the Amulet prototype for EMA, and for continuous-monitoring applications on human subjects.

Overall, these studies serve to highlight that our work on the Amulet Platform is a promising approach for designing platforms and devices that make it easy for application developers to develop applications, that are useful in healthcare and health research, and that all of this can be done on a platform designed with security in mind. For more information on these user studies, we refer the reader to our paper for more details [91].

3.7 Related Work

The collection of software and hardware techniques in Amulet draw extensively from the wireless sensor-network literature. In this section we address related work in open wearable platforms; software architectures for sensor-focused devices and other constrained devices; approaches for isolating the execution of application code; and techniques for modeling an application's energy and resource usage.

3.7.1 Open Wearable Platforms

Current commercial platforms such as Pebble, Android Wear, and Apple watchOS only document best practices and measure resource usage of running applications [149, 8, 14], and are closed source, closed hardware, or both. Other open platforms were being developed concurrently with Amulet as the wearable hardware ecosystem evolved; some of these projects have been discontinued, such as Sony's Open SmartWatch Project [179] and the Angel Sensor [10]. Others projects and platforms are still being developed, such as BLOCKS [183] and ZWear [201], but they are not entirely open to users or developers, or do not address one or more of the Amulet goals.

Little information about their developer tools are available; none of them appear to provide compile-time, app-developer tools for predicting battery life for a

given app or combination of apps, and none are engineered to give battery lifetime measured in months. Hexiwear [94] is completely open source and open hardware, and built for the mbed platform. However, it lacks comprehensive developer tools that allow application isolation and evaluation of energy costs. Hexiwear was not built specifically for low-power operation like Amulet. The choice of high-powered components like a color OLED screen and ARM Cortex-M4 make lifetimes significantly less than the Amulet.

Wearable platform such as ZOE [113], Mercury [122], and Opo [97] have been designed by the research community to address specific sensing problems, or to explore specific research areas (like BodyScan [68]).

In contrast, Amulet provides open-source hardware and software in a general platform allowing wearable system designers to innovate at both the operating system and application level – importantly, Amulet provides a tool that gives application developers insight into the tradeoffs between energy and utility. The Amulet Firmware Toolchain and Resource Profiler provide novel capabilities essential to app development for long-running multi-application multi-developer wearables.

3.7.2 Software Architectures

Software architectures and operating systems exist for sensor networks and the Internet of Things, including TinyOS [116], Contiki [65], RIOT OS [26], and mbed OS [18]. TinyOS programs are written in a dialect of C (nesC), Contiki programs are written in a subset of C, and mbed OS programs are written in C++. Amulet programs are currently written in a subset of C but the techniques could be extended to support a subset of C++. SafeTinyOS [56] allows for the static analysis of code to improve reliability and programmer confidence in a solution for single TinyOS applications. Amulet uses similar static analysis techniques to provide application isolation and energy prediction for *multiple* applications. All but RIOT OS are

event-driven (like Amulet), and only one (mbed OS) provides application isolation and access control.

The mbed OS provides app isolation using the target processor's Memory Protection Unit (MPU). (Hardin et al. recently demonstrated [86] that a similar approach is viable on the MSP430, which is the primary app processor for the Amulet.) The mbed OS uses a single-threaded execution model for objects written in C and C++, which is similar to Amulet's state-machine approach (although mbed plans to add multi-threading in the future). Security is managed using the mbed OS uVisor which allows setting and altering fine-grain permissions and memory access for system modules and objects, offering object isolation. However, the mbed OS only offers this isolation on ARM Cortex-M3, M4 and M7 microcontrollers with a Memory Protection Unit (MPU), all of which are larger, higher-power microcontrollers than the target processors for Amulet. It may be possible to use our Amulet-OS techniques to extend mbed OS security for smaller microcontrollers.

Amulet's architecture combines high- and low-power components to achieve our lifetime, performance, and availability goals. This design is similar to other hierarchical power-management systems [180, 28, 1] that seek to provide resource-rich computing platforms with wide dynamic power ranges, by combining a hierarchy of functional "tiers" into a single integrated platform. By using the right tier for the right task, and aggressively powering off high-powered tiers when not needed, these platforms are able to combine high performance and network availability with low average power consumption and long battery life.

None of the other IoT frameworks in development appear to offer the multi-application and app-isolation features of Amulet.

3.7.3 Application Isolation

Traditionally, hardware memory management units (MMUs) prevent applications from interfering with each other and with core system functions. Software fault-isolation techniques extend the MMU by isolating malicious code and have been implemented for x86 and ARM architectures [169, 198, 199] or require hardware (MMU) support. All of these approaches require hardware (MMU) support not available on low-power processors, and incur significant runtime overhead. They are not tenable on the constrained hardware platforms that enable long-lived wearable deployment.

Language-based techniques provide a more efficient alternative, by changing the programming model [102, 71] to make dangerous actions impossible or easy to detect, and using a combination of compile-time static analysis and inserted run-time checks to detect the dangerous actions that remain [51, 56]. The Amulet architecture (at the time of our work) builds on these techniques, with a new focus on safely combining multiple applications without hardware memory-protection support and providing insight into each apps' share of system resources.

We note that, recently, Hardin et al. [86] presented a memory-isolation technique that improves upon our security isolation for Amulet. Specifically, in their work they leverage both compiler-inserted code and MPU-hardware support to realize efficient isolation without resorting to language limitations.

3.7.4 Energy and Resource Modeling

Previous research on energy modeling for constrained devices has focused on techniques for smartphones and wireless sensors. Tools like Sandra [135], eDoctor [123], Eprof [147], Carat [146] and PowerForecaster [134] help users make decisions about energy efficiency at install time, observe the effect of use from day to day, or di-

agnose abnormal battery drain. Other research has tried to identify how users and batteries interact – investigating the battery mental model of users [69]. These works focus on meeting user needs while ARP-View is focused on enabling the developer. Further, these systems are all in-situ; gathering information about usage and then presenting information to the user. However, integrating these tools with ARP-view could provide interesting avenues for future work.

Developer-focused tools have also emerged for smartphones. Tools have focused on identifying energy bugs at compile time [192], in-situ energy metering using kernel additions for energy model building [196], and then using those models to predict smartphone lifetime [105]. These tools, if carefully applied to constrained wearable platforms, could complement the developer insights gleaned from the Amulet Resource Profiler and ARP-view.

Other tools profile changes in the frequency and amount of system calls as a proxy for hardware profiling, to estimate changes in application energy efficiency [2]. Some tools estimate energy cost per line of source code [119]. These tools provide insight for the developer into energy usage and efficiency, but are not tuned to the specific needs of wearable development. Wearable applications are more energy constrained than cell phones, and rely on periodic sensing activities for their function. The developer must be able to easily identify energy expensive code segments that are periodically executed. This is why ARP-view exposes timers to the developer, allowing them to manipulate the frequency of sensing actions in a GUI, without additional profiling or emulation steps.

Simulation and emulation techniques have attempted to provide forecasts of energy usage, starting with tools from the wireless sensor network literature like Power TOSSIM [174]. WattsOn [138] extends the idea of emulation to provide insights and what-if analysis for Android applications, after running the application in an emulator. WattsOn specifically focuses on empowering developers to under-

stand energy efficiency, and is most closely aligned with the goals of ARP-view. However, WattsOn is phone-focused, and does not account for periodic sensing tasks that are crucial to wearable operation. Nor does it provide real-time feedback on code and duty cycling changes. Wearables require special attention to be paid to periodic sensing tasks, timing and duty cycling, and costs of each API function. The Amulet Resource Profiler seeks to empower wearable developers without requiring specialized hardware knowledge, or costly profiling and emulation, all while accounting for sensing and user-interaction costs.

3.8 Summary

In this chapter we present our work on the Amulet Platform, which comprises a toolchain (the Amulet Firmware Toolchain) and runtime (the Amulet-OS) for development of secure and resource-efficient applications on low-power multi-application devices, such as wearable devices. In our past work [139, 91] we demonstrated that our platform offers developers security (e.g., application isolation) and feedback to empower them to develop resource-efficient applications. In this chapter we specifically emphasize security-related aspects of the Amulet platform and – to some extent – its usability.

In our work, we designed, implemented, and evaluated a system (and techniques) that provide application security guarantees in a way that is cognizant of its impact on energy and memory consumption. Our specific contributions presented in this chapter include substantial contributions to (1) the design and implementation of Amulet's software stack and runtime system, (2) the design and implementation of Amulet's firmware-production toolchain that guarantees application isolation, (3) the design and implementation of resource models that are deployed in Amulet's developer tools; these models aid developers in developing secure and efficient applications, and (4) an experimental evaluation of Amulet.

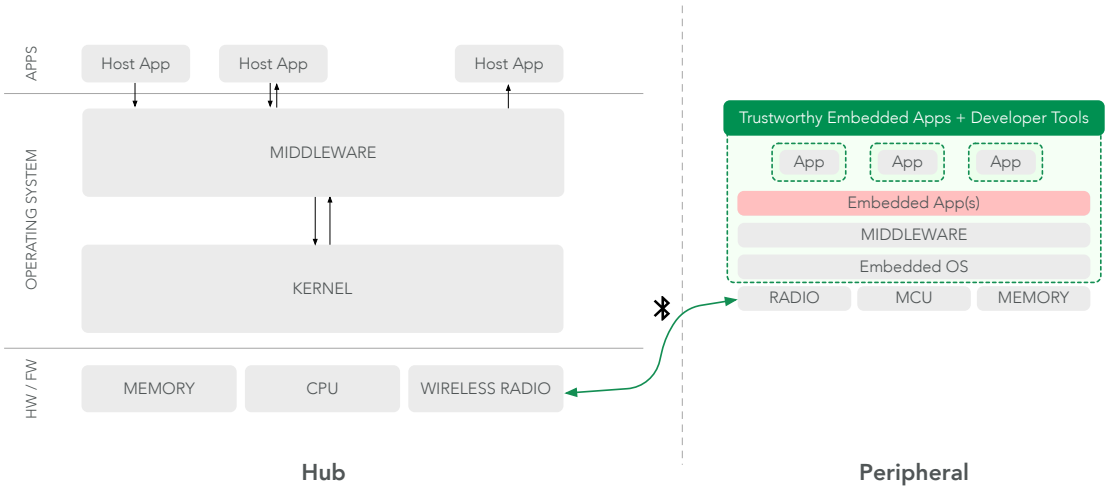

Figure 3.11: An overview of our system model and the threats to apps and data on peripheral devices illustrated alongside our solution.

4

Verifying Trustworthy Behavior

4.1 Introduction

In the preceding work, we have primarily focused on *prevention* – how devices and networks can be composed to provide trustworthy foundations for Wireless Personal Area Networks (WPANs). While these efforts are necessary, they are not enough by themselves. If history is our guide, at some point the adversary will be successful in compromising the system or network. The question then becomes: *now what?* We argue that prevention (through hardening devices and networks, for example) is not enough, and that there must exist some mechanism

to detect inauthentic interactions between applications and nearby devices. Such a mechanism could, for instance, alert users to potential risks and take actions to mitigate those risks. We address this challenge in this chapter.

At its core, we envision a system that wants to determine whether all of the devices in question are still *authentic*. Or in other words, are they still in desirable state, running code they are intended to be running, and operating (*behaving*) in the normal, expected way? Unfortunately, verifying this property is not as straightforward as one would hope. In Section 4.2 we present background information and related work towards this direction. This brief survey also helps to motivate our approach (Section 4.4) to verifying device authenticity and – ultimately – securing WPANs.

4.1.1 Overview of our Approach

To verify trustworthy behavior between apps and devices as they interact, we present our approach: *Verification of Interaction Authenticity (VIA)*. VIA introduces models that characterize typical, authentic interactions (network communication) between apps and devices. The modeling techniques used in VIA are inspired by past work in anomaly detection and Intrusion Detection Systems (IDSs) [104, 124, 188, 112, 82, 32, 48, 46, 121, 95, 140, 141]. Our implementation of VIA resides within a trusted region of a Bluetooth host (e.g., within a trusted OS; within the Secure World on a TrustZone-enabled platform; within an Enclave on an SGX-enabled platform) and requires privileged access to all network traffic to and from the Bluetooth host. Access to all ingress and egress network traffic enables VIA to construct hierarchically-segmented models based on n-grams and other features extracted from packet headers and payloads.

VIA is distinct from past work in multiple ways. For instance, past work has primarily focused on IP-based communications [82, 133], Wi-Fi [176], ZigBee [197],

and Z-Wave [197]; focused solely on a particular platform, such as Samsung SmartThings [197]; or requires invasive modifications to *each* Internet of Things (IoT) device [75, 141, 42], which is rarely (if ever) practical. (Indeed, not a single device in our testbed of more than 20 smart health and smart home devices is "open" to modifications.)

In its current form, VIA is completely agnostic to higher-level protocol semantics and internal device state. VIA's models are only concerned with constructing models for typical communications, which can be used as a reference to verify consistency with these models in future interactions.

Furthermore, unlike other approaches to capturing Bluetooth network traffic, VIA does not require additional hardware or depend on unreliable techniques to sniff Bluetooth connections over-the-air [5, 84, 31]. To the best of our knowledge, this is the first academic work that makes use of raw Bluetooth HCI traces collected on a smartphone – which only requires access to the easy-to-enable HCI logging feature under the Android "Developer Options." (This access does not even require the phone to be rooted.) In our current implementation, all HCI logs are written to the SD card and are easily accessible via the Android Debugger Bridge (ADB) utility.

VIA is also the first in-depth work to focus on characterizing Bluetooth devices based solely on Bluetooth network communications. Given the market dominance of Bluetooth and its increasing popularity for IoT use cases, we believe this work has broad applicability to areas of active research, such as mobile security and trustworthy computing.

4.1.2 Assumptions

We reiterate that that VIA resides on the Bluetooth host, within a trusted region of a Host, and requires privileged access to all network traffic (over Bluetooth)

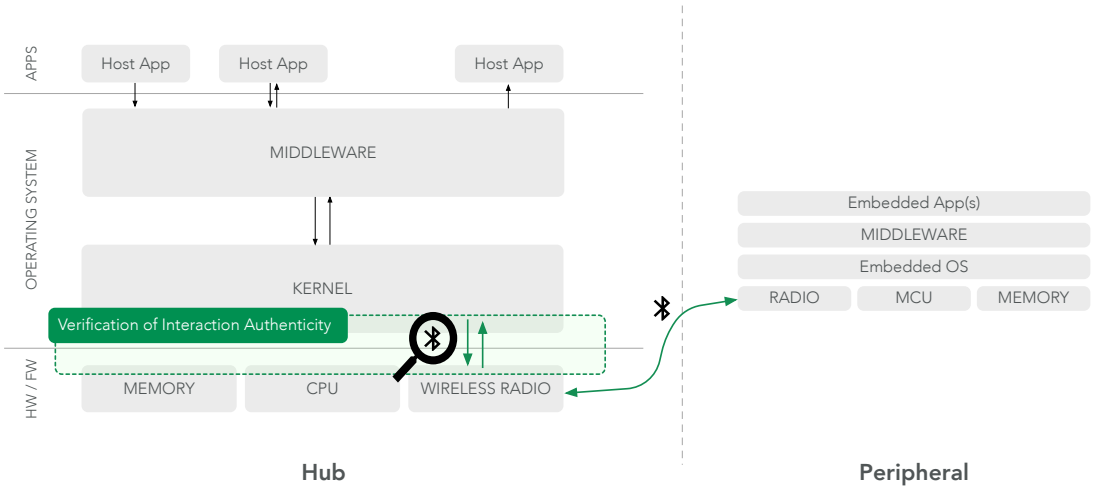

Figure 4.1: Overview of System Model and VIA.

to and from the Bluetooth host. Indeed, our larger vision is that VIA would be deployed within a Trusted Execution Environment (TEE), such as an Software Guard Extensions (SGX) Enclave on IA platforms [100], or within the "Secure World" on Arm platforms [19], or integrated directly into firmware within an I/O controller, alongside our work described in Chapter 2.

For simplicity, VIA can be viewed as a trusted third party that resides *within* the Host, and sits between apps running on the Host and the devices that connect with these apps. Thus, for illustrative purposes, we visualize VIA as sitting among the lowest layers of a Bluetooth host system (Figure 4.1). (This topic is discussed in more detail in Section 4.3).

4.1.3 Contributions

VIA is a novel approach to verifying and enforcing trustworthy behavior within WPANs. We make the following contributions:

- We collected and present a new, first-of-its-kind dataset, which captures Bluetooth HCI traces for app-device interactions between more than 20 smart-health and smart-home devices. (According to a recent survey [3] and much

effort to find any sort of dataset on Bluetooth network traffic, there appear to be no such Bluetooth datasets; we believe this dataset is extremely valuable in its own right). To equip future researchers with better data for investigation, our new dataset will be made publicly available after this dissertation is published.[1]

- We contribute extensions to open-source Bluetooth analysis software to enhance the available tools for practical exploration of the Bluetooth protocol and Bluetooth-based apps. To equip future researchers with better tools for investigation, the Vagrantfile and Dockerfile for the respective VirtualBox VM and Docker Container will be published along with our dataset. Furthermore, most of the code we wrote is already publicly available: `btsnoop`[2] and `bluepy`.[3]

- We present a novel modeling technique (*hierarchical segmentation*) for characterizing and verifying authentic BLE app-device interactions.

- We present an experimental evaluation of our new modeling technique, which presents strong evidence that our technique can realize effective characterization of, and distinction between, smart devices within WPANs for smart homes and smart healthcare.

4.2 Background & Related Work

The peripheral devices that we consider are essentially nothing more than embedded systems (which have been around for decades) coupled with some networking technology. These are generally not *open systems*, meaning they run one application

[1] https://crawdad.org/dartmouth/bluetooth-hci
[2] https://github.com/traviswpeters/btsnoop
[3] https://github.com/traviswpeters/bluepy

(or a few) created by the device developers; they generally cannot be extended with applications from third-party developers. Thus, typical solutions for protecting systems, such as installing third-party security software (e.g., antivirus software, security agent software), are not applicable; in fact, in some cases – such as in FDA-approved medical devices – altering the software might invalidate the certification!

This limitation – not being able to modify the peripheral devices – has generated a lot of interest and work in *non-invasive* approaches to measuring the state of devices, and using the measurement results to determine whether devices are *authentic*. The detection of non-authentic devices (through measurements that indicate non-authentic device *behavior*) may be an indication of the presence of threats to other personal devices or networks; by detecting such threats, there is an opportunity to take some action to secure personal devices and networks. In the remainder of this section, we discuss some of these non-invasive approaches and relate them to our approach.

4.2.1 Authentication

The overarching objective of our work (the verification of authentic interactions within WPANs) is largely motivated by past work in authentication [108, 57, 144, 74, 24, 126, 125, 133, 33, 99, 141, 127]. **Authentication** is a process to verify (i.e., establish the truth of) an attribute value claimed by or for a system entity [173]. In computer systems and networks, authentication is used to verify that a person (or another system) is in fact who or what it claims to be. This relates to our work in that VIA attempts to verify that apps and devices interact in a way that is consistent with what they claim to be. Generally speaking, authentication can be achieved in one of two ways: *identification* or *verification*.

Identification

A system that performs **identification** attempts to recognize a specific entity and distinguish it from other entities in a known population. In this way, the task of identification is a one-to-many matching problem; i.e., the output of an identification system is a single identity or class contained within the population.

To perform identification: First, the system collects (or otherwise obtains) many samples from the population (e.g., models for app-device communications within a population of apps and devices). Later, when an unknown entity presents itself to the system, it is the system's responsibility to determine which member of the population matches that entity.

Verification

A system that performs **verification** either accepts or rejects an entity after examining some input information about the entity's claimed identity. In this way, the task of verification is a one-to-one matching problem; i.e., the output of verification is binary: accept or reject.

To perform verification: First, the system collects (or otherwise obtains) an identity and model for a particular entity, and stores it. Later, when an entity presents itself to the system and asserts its identity, it is the system's responsibility to verify that the presentation matches the model for that identity.

4.2.2 Traffic Analysis & Intrusion Detection Systems (IDS)

Wireless traffic analysis systems and Intrusion Detection Systems have a long and rich history [104, 124, 188, 112, 81, 32, 11, 17, 46, 131, 33, 95, 75, 197, 4, 166]. Generally speaking, wireless traffic analysis systems and Intrusion Detection Systems (IDS) continually monitor computers or networks, collecting data (e.g., system

calls, network communication), extracting quantifiable features from this data, and applying a variety of techniques to analyze the data in search of signs of anomalies or compromise.

With respect to IDSs, there are traditionally two broad categories of IDS with respect to *where* they are located: Host-based IDS (HIDS) and Network-based IDS (NIDS) [75]. HIDS are generally software located on the system being monitored, and typically monitor only that system. NIDS are often physically separate devices located somewhere "upstream" in the network of the system(s) being monitored; NIDS generally monitor many separate systems on a common network.

NIDS are advantageous in the context of our work because their presence is generally transparent to the systems being monitored, which means we could deploy a NIDS to monitor peripheral devices, for example, without needing to modify them in any way. (This assumes that peripheral network traffic is routed through a network device – such as a smartphone or other hub device – where a NIDS is deployed, or that the NIDS has access to all network traffic, e.g., by sniffing an area of interest.) Because of this advantage, we focus our attention on Network-based Intrusion Detection Systems.

There is an extensive collection of work on both categories of IDS. (In fact, there are even additional categories such as hybrid and distributed IDS that exist as well, which combine both host-based and network-based IDS data – from multiple sources – into one system.) A more thorough discussion of IDS, however, is beyond the scope of this dissertation; we refer the interested reader to two recent surveys on this topic [95, 75]. We can also recommend a slightly dated, but nonetheless insightful, brief history and overview of intrusion detection [104].

There are also two broad categories of IDS with respect to *how* intrusions are detected [178]: "systems relying on *misuse-detection* monitor activity with precise descriptions of known malicious behavior, while *anomaly-detection* systems have a

notion of normal activity and flag deviations from that profile." Both approaches have their strengths and weaknesses.

Misuse detection defines patterns ("signatures") for malicious behavior, and scans a data stream for matches, looking for evidence of known attacks [104]. Misuse detectors generate few false positives[4] (i.e., a flag is only raised if the *exact* signature is matched), which is desirable, but they are only capable of detecting known attacks that have a well-defined signature; worse, introducing slight variations into known malware can often cause signature-based detection mechanisms to miss even "known" attacks (i.e., unknown devices or networks would lead to higher rates of false negatives).

Anomaly detection uses models of expected behavior (generally obtained after some training phase), and interprets deviations from this "normal" behavior as a problem [104]. It is worth noting that an inherent assumption of anomaly detection is that attack behavior differs from normal behavior. One advantage of anomaly detectors is that they are capable of detecting unknown attacks. Also, it may be possible to represent the models in an efficient way, making real-time anomaly detection possible. Unfortunately, anomaly detectors are also known for having high false-positive rates (i.e., not all anomalies are an attack).

Both misuse detection and anomaly detection have their own strengths and weaknesses. In this dissertation we opt to explore techniques commonly used to realize anomaly-based detectors for one key reason: Authentic interactions may take varied forms. Rather than attempt to craft precise signatures for every possible interaction – as misuse detectors attempt to do – we prefer techniques

[4] According to some experts, this is a common statement in the literature, but it is not generally true. In practice, signature-based detection works well within the context of the specific networks and devices used to collect data and develop these signatures. When new devices (devices not observed during the creation of signatures) are introduced into the mix, the false-positive rates of misuse detectors may increase. Similarly, signatures generated based on data from one network may give rise to increased false-positive rates when applied to a different network. This only serves to strengthen our interest in anomaly detectors over misuse detectors.

that are capable of modeling normal behaviors, and deviations from those normal behaviors.

Due to the simplicity of peripheral devices, which generally serve a well-defined purpose and perform well-defined and often repetitive tasks (i.e., low variability), we believe applying anomaly-based monitoring to networks of these devices (such as Bluetooth devices and networks) may work well. Furthermore, given the scale of personal networks (dozens of devices, as opposed to hundreds or thousands of devices), it may be possible to at least help a user identify a specific problematic device that has raised a flag in the anomaly detection system, enabling them to respond in some way (e.g., search online for known problems, report the device to the manufacturer, stop using the device, or perhaps even throw the device away).

4.2.3 Side-Channel Measurement & Analysis to Detect Malware

Side-channel analysis has also been widely explored [107, 44, 27, 25, 55, 175, 194, 72, 187]. In particular, we have obtained invaluable insights from work that has used side-channel analysis as an approach to detect malware. Indeed, recent work that inspired some of our thinking proposed measuring unintentional side-channels of medical devices, such as power consumption, to infer the state of the device [55]. Their theory is that the power consumption of computing devices scales closely with system workload. By measuring power consumption at a (specialized) power outlet, they can use these measurements as a proxy for computing activity (often referred to as *behavior*) and detect problems, including malware. As in our situation, this approach works well when devices perform well-defined, repetitive tasks that should exhibit little variation from one run to another. Furthermore, this approach is non-invasive: it can work without modifying the target device since the side-channel measurement and analysis can be done at the power outlet. However,

this solution requires the devices to be continuously powered via their specialized power outlet to detect malware. In our context, peripheral devices are not always (or ever) line-powered. If these devices are line-powered, it would be quite easy for malware to monitor the power state of the device, conceal itself while line-powered, and continue operation when disconnected from line power.

In light of this work, we believe that it may be possible to accurately model the well-defined (and often repetitive) network communications that occur in WPANs, such as Bluetooth networks. In fact, given devices that perform well-defined and often repetitive tasks, we believe that a device's communications may serve as a proxy for its computing activity. Under this assumption, it may be possible for a classifier to accurately detect a divergence from models for authentic communication, which may be indicative of a device whose firmware has changed, or of the presence of an inauthentic device masquerading as a legitimate device.

A core assumption of this approach is that such "problems" lead to network traffic that does not fit the normal profile. Given an understanding of common malware (such as those enumerated in Figure 4.2), our intuition leads us to believe that an infected peripheral device will produce abnormal network activity (downloading files, creating new channels, opening ports, probing the mobile device, etc.) that can be detected at the hub device (to which it connects).

4.3 System, Network, and Security Model

In this dissertation we consider WPANs that consist of multiple peripheral devices that connect with a central hub device (recall Figure 1.1). As discussed in Chapter 1, we focus on personal *hub* devices and *peripheral* devices (*IoT* devices)[5] in this work,

[5]Technically both peripheral devices and hub devices are IoT devices, but "IoT device" is most often used to refer to devices that fall under the umbrella of peripheral devices; the new, smart, network-capable "things" that connect with applications or services over the Internet via a hub

Malware	Activity
almanahe	Download and execute files
autorun	Open port and IE instances
bamital	Disable system programs
bredolab	Download other malware, crash system, unhook API
delf	Allow remote access
dorkbot	Download other malware, Initiate DOS attacks, Steal personal info
fakeav	Download other malware, Ransomware
kolab	Spam IRC channels
mabezat	Infect host files
ramnit	Run malicious routines
sality	Infect and delete host files
sillyfdc	Disable services
virut	Infect host files, create backdoor
zbot	Steal personal info, create bot
cryptic	Download other malware

Figure 4.2: Examples of malware samples representing unusual device behavior [55].

but note that these concepts extend to other devices under a similar network model. (We elaborate on our system and network model, and discuss how these concepts generalize, below.) This approach – having all IoT devices connect with a personal hub – has many advantages with respect to security and privacy; for instance, data stored on such a hub (e.g., within the Databox [67]) can be tightly controlled, enabling people to determine with whom (or what) their personal data is shared.

Hub devices in WPANs are often mobile devices (e.g., smartphones) but need not be mobile; rather, the role of this device is a system that runs end-user applications and serves as a gateway for IoT devices to access the Internet. In fact, this central gateway device might be a popular home "hub" device[6], such as Amazon Echo [6], Apple HomePod [15], or Google Home [79]; or a dedicated *health* hub device, such as the HealthGo Mini [66]. A more traditional home gateway device, such as a home Wi-Fi access point (AP), could also be a hub. Few Wi-Fi APs today, however, run end-user applications or include WPAN technologies such as

device. For this reason, when we refer to IoT devices in the text, we specifically refer to peripheral devices that are distinct from hub devices such as smartphones.

[6]Indeed, these popular consumer devices are the canonical "IoT gateway" devices of today.

Bluetooth, which makes APs somewhat different as compared to the hubs that we generally consider throughout this dissertation.

IoT devices may interact with more than one hub device. For example, a home's IoT devices may be controllable by all of the members of the family that occupy the home; each member may have their own personal hub device. For the sake of simplicity in the following text, we restrict our discussion to the scenario where multiple IoT devices connect with a single hub device.

IoT devices rely on the hub device as means to access the Internet (to upload data to services, or to interact with other IoT devices, for example) and to interact with the user (by presenting data to the owner/operator of the hub device). The hub devices rely on IoT devices as a means to collect data (health data from a worn device, for example) and to act on the environment (administer medication via an implanted insulin pump, for example).

In light of this context, our work explores the viability of deploying anomaly-detection and intrusion-detection techniques within a hub device at the center of a WPAN. We develop VIA models, which are deployed *within* the hub device, that use access to network traffic to and from its connected IoT devices to verify the authenticity of interactions within the WPAN. VIA uses network-traffic analysis and anomaly-detection techniques (such as those in Section 4.4) to infer deviations from normal interactions between (apps running on) hub devices and nearby IoT devices.

In the remainder of this section we describe our security model, which consists of our assumptions and trust model, the anticipated threats and adversaries, and the goals of our solution.

4.3.1 Assumptions & Trust Model

We assume that any personal hub and IoT devices – when initially deployed – are deployed in a secure environment without any malware. This provides a window of time in which to train VIA models.

Many IoT devices are deployed in, and generally exist in, a secure environment (such as a home), which offers reasonable protection from physical threats; network-based threats may still be an issue, however. Furthermore, depending on the mobility of these devices, they may not always reside in the secure environment. For example, wearable devices such as a fitness band or heart-rate monitor may be worn out of the home. Likewise, hub devices such as smartphones, which are frequently carried by people wherever they go, often leave the confines of the home for extended periods of time. These realities may create an opportunity for an adversary to compromise a hub or peripheral device.

We assume that IoT devices do not support the addition of third-party security mechanisms, such as antivirus, anti-malware, or any other security-related software agents. Furthermore, IoT devices have limited resources (per our definitions of peripheral devices stated in Section 1.1), such as energy, processing power, and memory. Hub devices are generally less constrained in terms of these resources, but any software deployed on a hub device should still be conscious of its impact on energy consumption, memory footprint, network usage, and so forth.

We assume VIA will be deployed within a Trusted Execution Environment (TEE). Some hub platforms come equipped with a TEE, such as SGX [100] or TrustZone [19] (recall Chapter 2). On such a platform, VIA may be safely deployed within a TEE (e.g., [111]), ensuring that VIA can reliably operate even if other system components (e.g., the OS) are compromised. For clarity, we depict potentially untrusted components (red) within the hub and peripheral devices in Figure 1.4.

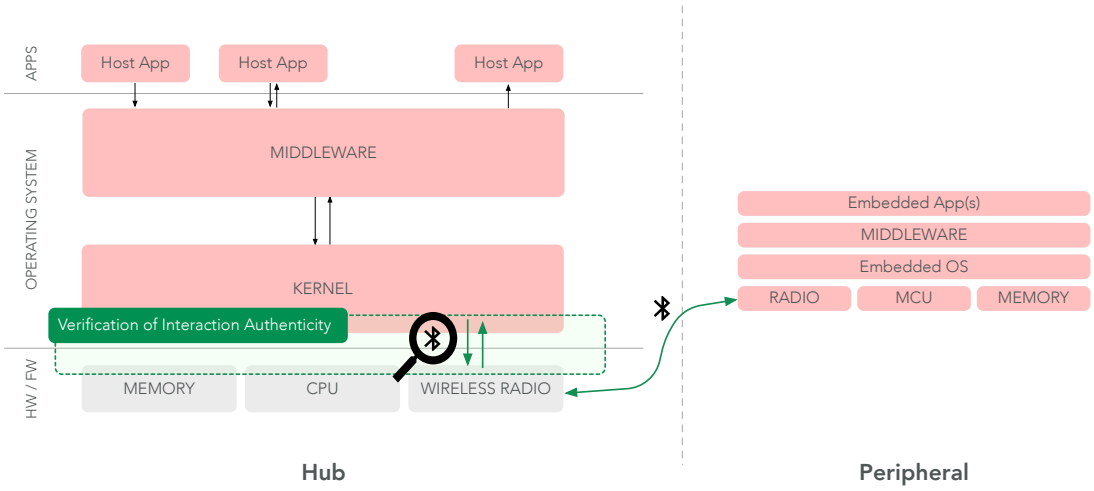

Figure 4.3: Illustration of the VIA threat model and our solution.

Last, we assume that an adversary knows how VIA works (i.e., they may understand the models used to distinguish anomalous behavior from normal behavior). Even with this knowledge, however, our theory is that malicious activity on the part of the adversary *will* result in abnormal activity – and network traffic – that can be observed by VIA. (See Section 4.3.2 for more discussion on this topic).

4.3.2 Threats & Adversary Model

The ultimate goal of the adversary is to compromise IoT and hub devices to steal sensitive data (e.g., personally identifiable information (PII), medical information, financial credentials), commandeer resources (e.g., device's computing resources), tamper with data (e.g., inject false data into applications or services), and to propagate itself (i.e., to leverage the compromised devices to carry out attacks against other devices). To clarify our threat model further, we describe two types of threats that our work specifically aims to address: *inbound threats* (peripheral-to-hub) and *outbound threats* (hub-to-peripheral). We then clarify the scope of our work and the types of adversaries we hope to address with this work.

Inbound Threats

Consider a malicious device that comes into close proximity (i.e., within Bluetooth's wireless radio range) of a target hub device. Here, a malicious device may be a *cloned device*, which is an attacker-controlled device that has cloned the identifiers of another device (MAC address, UUID(s), device name) with the objective of establishing a connection with the target hub device in an attempt to attack it. Or it may be a *compromised device*, which is a legitimate user-owned device that has been compromised by the attacker, and is now under the attacker's control. Again, the objective of the attacker here may be to use this device to attack the target hub device; or more specifically, applications that run on the hub device.

Outbound Threats

Consider a malicious application present on the victim's hub device, which has the ability to access the Bluetooth interface. For example, on Android, this is as simple as an application having the `Bluetooth` and `Bluetooth_ADMIN` permission — two permissions that are claimed by almost all Bluetooth-capable applications [144]. Such a malicious application may, for example, use its access to the Bluetooth interface to attack nearby IoT devices.

Scope & Limitations

As has been ceded in past work [55], our work does not aim to address targeted threats launched by determined, well-funded adversaries. We acknowledge that an adversary with detailed knowledge of any defense mechanisms could in theory design an attack to specifically thwart or evade that defense mechanism. Fortunately, targeted threats appear to be rare [55]. Thus, as others have done (e.g., [55]), we focus our attention on *garden-variety* threats (e.g., generic malware that targets a vulnerability present in a large class of devices), which are a clear and present

danger to hub and IoT devices.

Furthermore, it is important to note that VIA seeks to identify abnormal *behavior* by monitoring network communications (interactions). Because our work concentrates on behavioral features, our system will likely have a hard time detecting any difference between two devices that behave in similar ways, but run on physically different devices/hardware. For example, consider a model which accurately characterizes a heart-rate monitor used in remote patient care. Now consider an imposter heart-rate monitor device under the control of an attacker that "behaves correctly" (e.g., reports heart-rate values in a reasonable range) but is intentionally sending high heart-rate values (perhaps to convey a high resting heart rate to a medical record system, which may result in higher insurance costs due to high-risk health indicators). This imposter is in scope for our work, but we admit that we would not easily (or at all) be able to detect this sort of "attack" since the device is behaving within its normal profile. Thus, regardless of the underlying device/hardware – which may need to be verified by other means, such as platform attestation (e.g., [24]) – we only propose to identify authentic behavior vs. non-authentic behavior based on features related to communication between the devices.

Since our understanding of most of the previously-seen malware is that it does not operate in such a subtle manner (e.g., [55]), we suspect that attacks like the one described above are rare (or even nonexistent). The attacks that we are familiar with – and the attacks that we anticipate are common – are less subtle (recall Figure 4.2), and our models should be successful in detecting attacks that exhibit similar characteristics. Indeed, these are the attacks that lead to more compromised devices, or that enable theft of data; these are the attacks that pose significant risk to the security and privacy of end-users and their data.

4.3.3 Goals

The overall goal of this work is to investigate the viability of applying anomaly-based intrusion-detection techniques to WPANs, such as Bluetooth networks. More specifically, our goals are to:

- **Increase user-awareness of threats in WPANs** by providing a means for hub devices to detect abnormal interactions between devices in the network, and report this to the user.
- **Increase hub device security** by protecting apps on the hub device from malicious IoT devices.
- **Increase IoT device security** by protecting IoT devices from malicious apps on a hub device.

4.4 Verification of Interaction Authenticity

In this section we present *Verification of Interaction Authenticity (VIA)*, our approach to verifying trustworthy interactions (network communications) between devices within WPANs (Figure 4.4). Here, we focus on the task of *verification* to introduce a new mechanism for ensuring that apps and devices continue to interact in a way that is consistent with prior observations (and is, with reasonable confidence, *authentic* and therefore *trustworthy*). Deviation from authentic interactions will result in failed verification, enabling devices to take action and mitigate potential network threats.

At a high level, our larger vision for VIA is that it will be deployed within a hub and work by monitoring all network traffic to and from connected devices. When a hub device and a peripheral device connect, identifying information is exchanged,

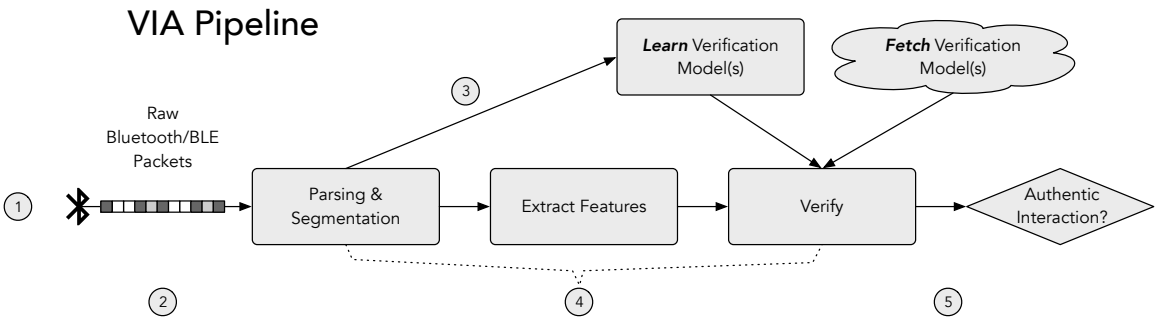

Figure 4.4: Overview of the VIA pipeline: (1) Establish connection between devices. (2) Observe peer device identity claim (e.g., BD_ADDR and LTK) at the time the connection is established. (3) Based on the previous step, load the appropriate verification model. (Verification models can be learned as-needed or fetched from a trusted source, such as a device manufacturer.) (4) Monitor app-device interactions (parse and segment packets, compute n-grams, perform verification). (5) Depending on the verification result: if the interaction is deemed *authentic*, allow interaction to continue and repeat verification; otherwise, take action, such as *alert user* or *disconnect app and device*.

such as unique Bluetooth device addresses (BD_ADDR) and secret information that can be used later for authentication (LTK). Based on this information, an appropriate normality model can be loaded to perform verification. (Verification models can be learned as-needed or fetched from a trusted source, such as a device manufacturer; see Section 4.6.2 for more discussion on bootstrapping initial VIA models.) Going forward, VIA can monitor all app-device communication (i.e., parse and segment packets, compute n-grams, perform verification), and determine whether the interactions are trustworthy. If the interactions are successfully verified as being *authentic*, interactions will be permitted to continue, and verification will be repeated again at some point in the near future; otherwise, VIA will take action, such as alert the user or terminate the connection between the offending app and device. (See Section 4.6.2 for more discussion of response strategies.)

Toward this vision, in this work we study the use of n-grams (Section 4.4.1) and our new methodology for model separation, *hierarchical segmentation* (Section 4.4.2), to build and validate VIA models. In future work we will use the building blocks we present below to fulfill our larger vision. In Section 4.6.2 we discuss some of the

challenges and opportunities surrounding a real-world deployment of VIA in the future.

4.4.1 Using N-grams for Network Traffic Modeling & Analysis

Our approach to model network traffic is to create models based on the contents of packets (headers and/or payloads).[7] Attacks commonly try to exploit vulnerabilities in services or applications by delivering maliciously crafted payloads; or, in the case of packet headers, attacks may try to exploit vulnerabilities in how the headers are parsed and interpreted. By modeling aspects of normal packets, it is possible to detect deviations in packet content that may indicate an attack.

To model a normal packet, some past systems (e.g., PAYL [188], PCkAD [11]), have used n-grams. To describe data in terms of n-grams, we adopt the definition presented by Wressnegger et al. [191] and summarize it below. Each data object x first needs to be represented as a string of symbols from an alphabet, A, where A is often defined as bytes or tokens. For example, in modeling network packets, we simply consider a packet (or part of a packet, such as the packet headers or the packet payload) as a string of bytes. By moving a window of n symbols over the string of bytes in each packet x, we can then extract all substrings of length n. These substrings (n-grams) give rise to a map to a high-dimensional vector space, where each dimension is associated with the occurrences of one n-gram. Formally, this map ϕ can be constructed using the set S of all possible n-grams as,

$$\phi : x \to (\phi_s(x))_{s \in S} \quad \text{with} \quad \phi_s(x) = \text{occ}(s, x)$$

where the function occ(s, x) simply returns the frequencies, the probability, or a binary flag for the occurrences of the n-gram s in the data object x.

[7] This approach assumes that relevant packet contents are accessible to VIA – i.e., non-encrypted.

Past work has shown that even for small n (e.g., $n = 1$), n-grams can be an extremely effective in modeling traffic patterns in a manner that is efficient, accurate, and resilient to mimicry attacks [188]. (In a mimicry attack, the attacker (1) takes control of a network device, and (2) successfully executes a payload while *mimicking* normal behavior. Thus, if an exploit sequence is contained in the normal profile, the attack will go undetected.) Attempts to improve upon simple 1-gram models have shown only slight improvements. For instance, past work has shown that slightly higher detection rates, and slightly lower false-positive rates can be achieved [11], but these improvements come at the expense of increased computational complexity, or the use of additional preprocessing that imposes domain knowledge. Furthermore, Angiulli et al. suggest that there is an inherent trade-off in building n-gram models where $n > 1$: greater values of n lead to higher false-positive rates, whereas 1-gram models have been shown to have lower detection rates (but not by much).

Ultimately, 1-gram models are simple, effective, and have low false-positive rates, which is a good starting point for investigating the viability of applying anomaly-detection techniques for verification of authentic traffic within Bluetooth networks. Thus, in this work, we use 1-grams to build models based on byte-frequency distributions observed over a sequence of one or more Bluetooth packets. The resulting models can then be used to verify future network communications.

4.4.2 Hierarchical Segmentation

A single model trying to characterize *all* packets has been shown to lead to an ineffective, monolithic model. Therefore, it is necessary to learn a variety of models that separate traffic into different groups, where similar types of traffic can be associated and compared. Past work analyzing IP traffic (e.g., [188]) learns separate models using a combination of destination port, packet payload length, and packet

direction (inbound or outbound).[8] Thus, if there were 5 ports and 10 different payload lengths for each port, their approach would learn 50 separate models for inbound traffic and 50 separate models for outbound traffic.

This approach is problematic for Bluetooth-based WPANs. For instance, some of the conventional notions (e.g., ports) are not directly applicable. We observe, however, that model separation based on ports is effectively meant to separate models based on the underlying protocol and semantics of interactions within the context of a specific protocol. Therefore, to capture specific semantics of underlying protocols in Bluetooth, we introduce our approach to model separation based on *hierarchical segmentation* of the various Bluetooth protocol layers (Figure 4.5).

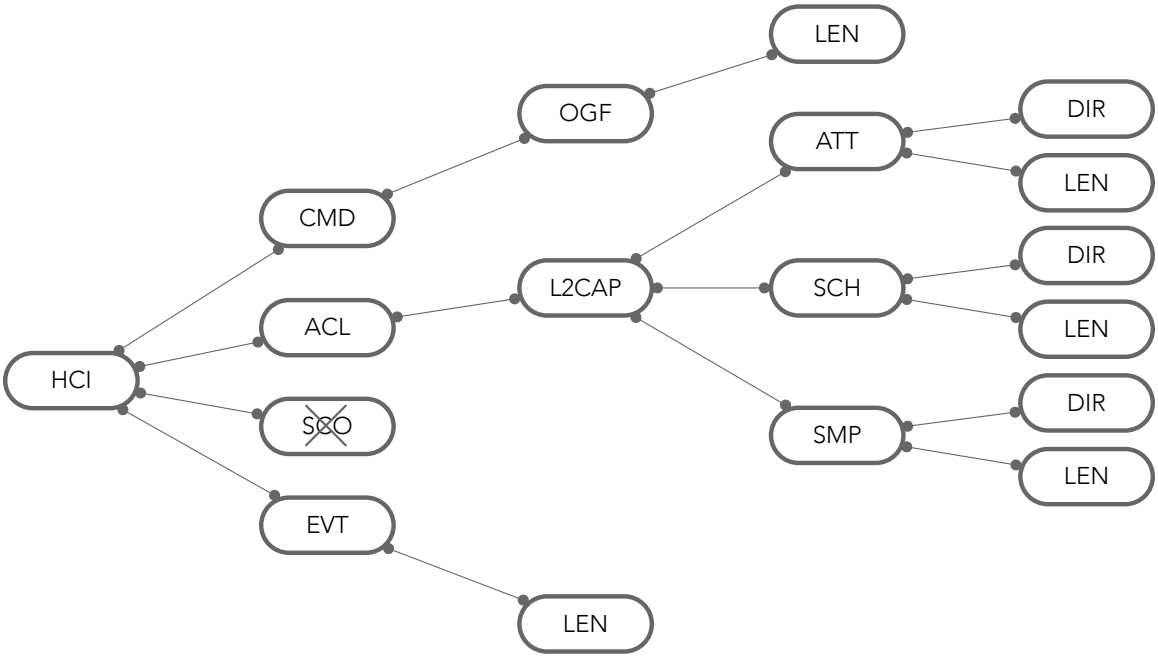

Figure 4.5: Hierarchical segmentation of the Bluetooth protocol stack for VIA models.

As we traverse the hierarchy illustrated in Figure 4.5 from left to right, we move "up the stack" in Bluetooth. The root of the hierarchy corresponds to the Host Controller Interface (HCI) layer – the lowest layer of the Bluetooth protocol in

[8] PAYL and related systems also describe solutions to common edge cases that can be problematic. For example, they describe approaches to merge models in the event of sparse training data.

which we can reliably capture Bluetooth network traffic.

The HCI protocol is made up of four different types of packets: command packets (CMD), which contain directives or requests from Host software; event packets (EVT), which contain responses to CMD requests or notifications of network events (e.g., connection requests); asynchronous data packets (ACL), which are used for higher layers in the Bluetooth protocol to exchange data; and synchronous data packets (SCO), which are primarily used for streaming data, such as audio. In our testbed of more than 20 smart-health and smart-home devices, not one of those devices ever exchanged SCO packets. Thus, VIA models only make use of CMD, EVT, and ACL packets.[9]

Packets in the HCI protocol can flow in one of two directions. Packets that flow along the entry path into the hub are referred to as ingress packets, and packets that flow along the exit path from the hub are referred to as egress packets. Because the direction of a packet is an important characteristic of the packet, shorthand notation is commonly used. Specifically, an ingress packet's direction is also referred to as `d2h` (device-to-host), since the direction of the packet is oriented from a device towards [to] the host. An egress packet's direction is also referred to as `h2d` (host-to-device), since the direction of the packet is oriented from the host toward [to] a connected device.

Each of the CMD, EVT, and ACL packets have substantially different characteristics in terms of directionality and packet length, examples of which can be seen in Figure 4.6 and Figure 4.7. CMD packets are unidirectional and flow exclusively from the Host to the Controller (`h2d`); the lengths of these packets are generally quite small, likely because the HCI protocol defines a a limited set of valid commands, most with few parameters. EVT packets flow exclusively from the

[9]Our approach to modeling should be applicable to SCO packets as well, but we have not had an opportunity to study SCO traffic to date. It is likely that devices with microphones or speakers will make use of SCO traffic, but none of the devices in our testbed had either.

Controller to the Host (`d2h`); the lengths of these packets are highly variable. ACL packets are bi-directional and can flow in either direction between the Host and Controller (`h2d` or `d2h`); these packets primarily transport application-layer data communicated between devices. We note that there tends to be more variability in ingress ACL packets. We see these differences between CMD, EVT, and ACL packets as meaningful features for creating separation between VIA models.

To illustrate the characteristics described above as seen in our dataset, we compiled all of the packets from all of the parsed trace files for a particular device (in this case, the Omron upper-arm blood-pressure monitor) and plot (Figure 4.6, Figure 4.7) the distribution of observed packet types, lengths, and directions. Note that while these plots are for a specific device, other devices show similar trends: most or all packets are 50 bytes or less; there are a few packet lengths that are frequently observed; the lengths of HCI EVT packets are highly variable while the lengths of HCI CMD packets are relatively small; and, there is more variability in the lengths of HCI ACL packets sent from a peripheral device to an app on the hub device as compared to the lengths of packets sent from the app to the peripheral device.

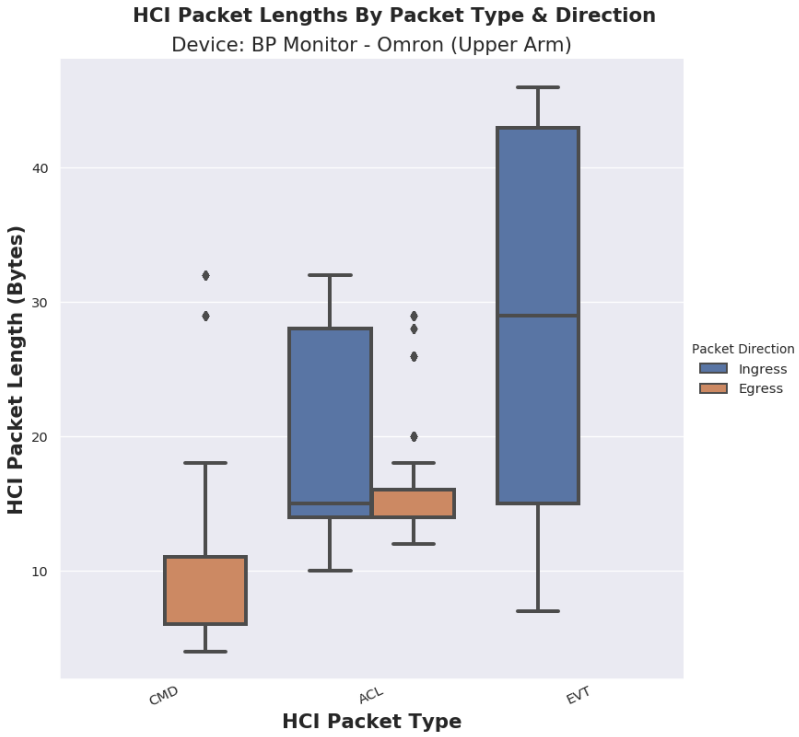

Figure 4.6: Distribution of HCI packet lengths by packet type and direction. HCI command packets (CMD) are generally small, and only egress from the perspective of our network traces. HCI event packets (EVT) are highly-variable in length, and only ingress from the perspective of our network traces. HCI data packets (ACL) are bi-directional and exhibit more variance on the ingress path.

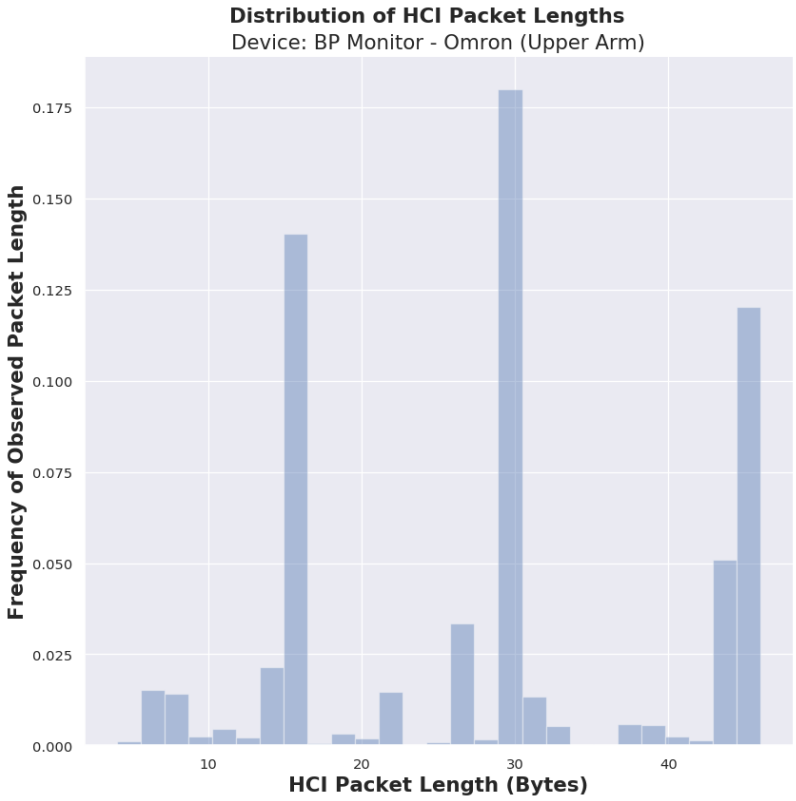

Figure 4.7: Distribution of HCI packet lengths and their observed frequencies. The sizes of packets observed in our dataset are small for the most part, and many packets are represented by a small number of packet lengths. This supports our approach of hierarchical segmentation that takes packet length into account as well.

Additional differences can be seen within each of the HCI packet types, especially ACL packets. Notably, ACL packets transport Logical Link Control and Adaptation Protocol (L2CAP) packets, which is a layer that provides features such as packet segmentation and retransmission. The primary function of the L2CAP protocol is to provide a reliable transport for Bluetooth applications. Above the L2CAP layer, we identify three critical protocols that can be used for further model separation: the Attribute Protocol (ATT), the Signaling Protocol (SCH), and the Security Management Protocol (SMP). It is through these protocols that devices can perform authentication, establish connections and logical channels, and exchange user data.

Commands (CMD) and events (EVT) are specified by an opcode and event

code, respectively. While there are nearly 300 HCI commands and events, they are grouped into several categories based on their function (e.g., configuring the Controller, requesting information about the Controller or nearby devices, discovering devices, establishing connections). Different *groups* of opcodes (OGF in Figure 4.5) lead to one form of separation for models composed of CMD packets. Event packets are not so easily separated; in our current work, packet length appears to be the best attribute for separation among event packets.

To summarize, VIA uses several features to learn models: a combination of n-grams, packet type, packet length, and packet directionality. These features enable VIA to create models that are highly effective in recognizing network traffic that is consistent with previously-learned authentic communications for interactions between a particular app/device pair, and to distinguish among the interactions for different app/device pairs. To give the reader some intuition for the features used in VIA, Figure 4.8 illustrates an overlay of the n-grams generated from more than 30 different traces[10] over time between a Kinsa thermometer and its companion application. Figure 4.8 provides compelling evidence for the viability of VIA as a solution for verifying trustworthy behavior within WPANs. We provide more quantitative evidence in Section 4.5.

[10] A *trace* is a packet capture consisting of all packets that are observed between the time that a hub and peripheral device establish and terminate a connection. For more information on the collection of traces, see Section 4.5.1.

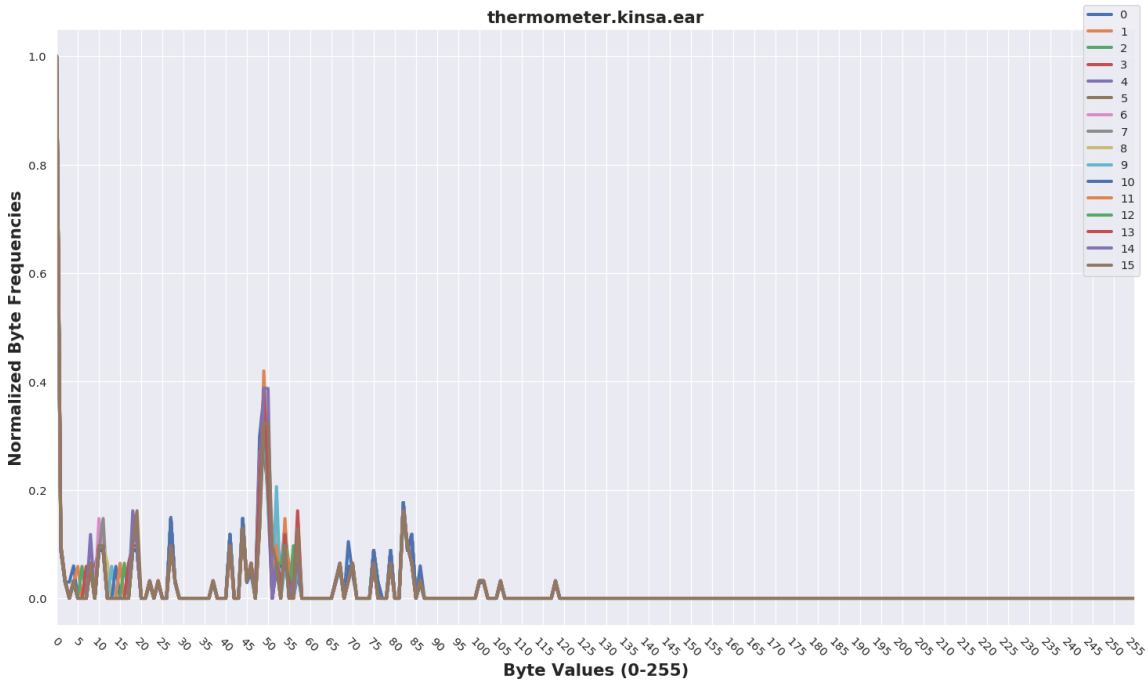

Figure 4.8: Example of normalized byte frequencies for HCI traces.

4.4.3 Model Selection & Verification

After models have been created, VIA can perform verification. To perform verification, VIA scans each packet according to the hierarchical segmentation described above, and selects a representative model for verification. Next, VIA computes the byte-value distribution of the relevant segment of the new packet and compares the result against the chosen verification model. If a new packet (or sequence of packets) is consistent with the verification model, VIA deems the interaction to be *authentic*. In Section 4.5, we discuss the use of off-the-shelf classifiers that can be used to determine whether a new packet is consistent with the verification model.

The description above is a slight simplification of how verification is performed. In reality, packets exchanged in Bluetooth communications are usually quite small. For example, in Bluetooth version 4.0 and 4.1, the maximum Attribute Protocol (ATT) payload is 20 bytes.

From preliminary experiments, we found that learning and verification works

best using a sequence of packets. For example, Figure 4.9 illustrates n-grams produced from examining the first b bytes of a concatenated sequence, where b equals 10, 50, 100, and nearly 300, respectively. In this example, the HCI trace captures all of the packets that were exchanged between the companion app and device (thermometer) connected and exchanged data about a measurement taken from the device. Clearly, using more packets will provide more information for model learning and verification. In our work we chose to use all of the available bytes in a network trace[11] to learn and evaluate our models, which equates to a few minutes of network activity per trace.

[11]The decision to use all of the bytes in a network trace is appropriate here because of how we constructed the traces, which we describe in detail in Section 4.5.1. In short, each trace contains bytes from only a short period of time (3-10 minutes) in which an app-device pair connect, exchange data, and disconnect. In future work, we will perform a more in-depth examination of the data to determine an optimal duration that should be used for building a model.

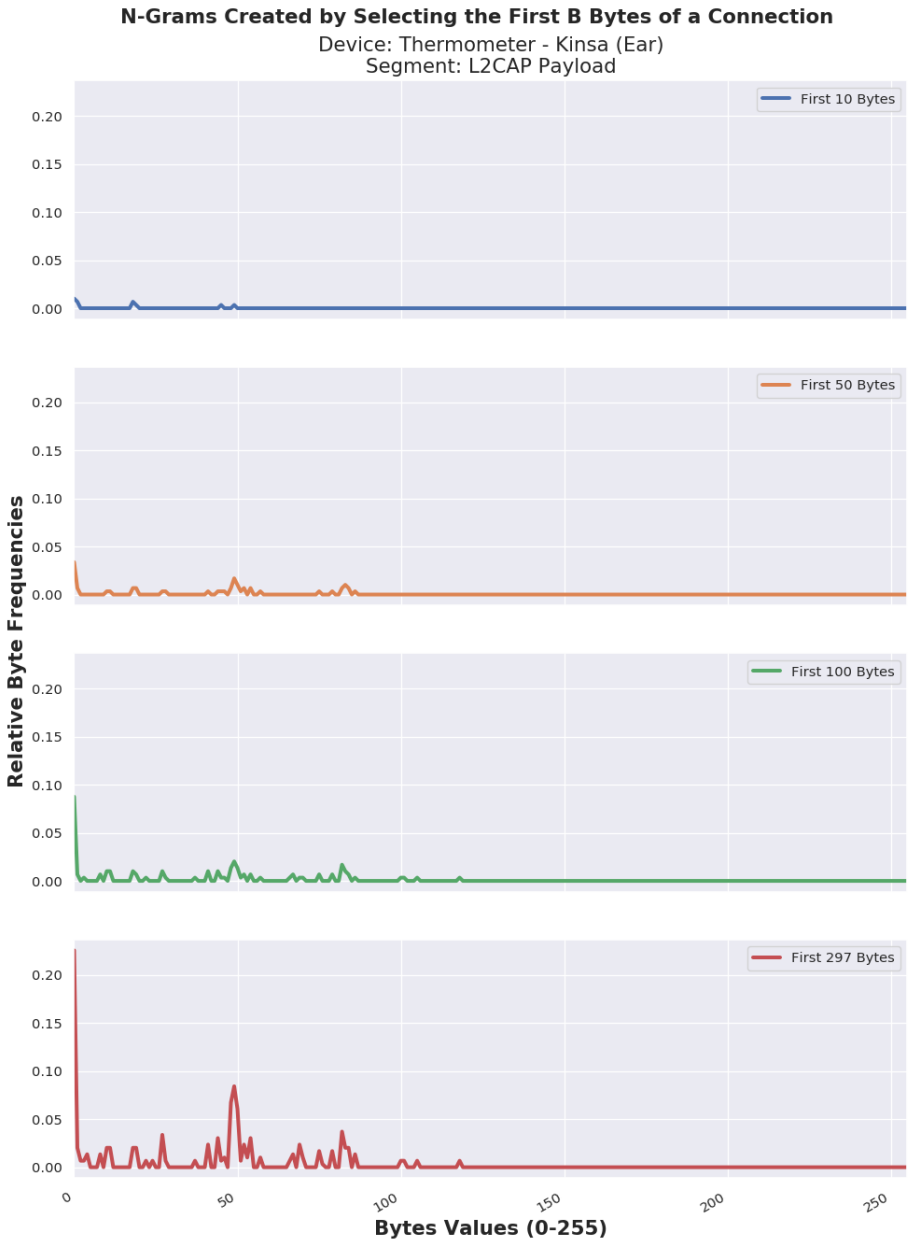

Figure 4.9: Example of the information gained from aggregating varying amounts of bytes from a trace before learning models or performing verification. Specifically, these subplots illustrate the n-grams that are produced from examining the first b bytes of a concatenated sequence, where b equals 10, 50, 100, and nearly 300, respectively. (In the last subplot, only 297 bytes are shown because only that many bytes were exchanged at the ACL/L2CAP layer during this particular network trace.)

4.5 Evaluation

In this section we study the use of *n*-grams and our new methodology for model separation in Bluetooth (hierarchical segmentation) as a promising approach for verifying app-device interactions over time. To evaluate the viability of our approach, we assembled a testbed (Figure 4.10) of two distinct device categories (smart-health and smart-home devices) consisting of 9 different device types, and 20 devices in total (Table 4.1). From this testbed, we produced a novel dataset of more than 300 Bluetooth network traces (Section 4.5.1), which will be made public following the publication of this dissertation.[12]

The remainder of this section describes our experiments that use the traces collected from our smart-device testbed. The primary thrust of our experiments is to show that our approach can produce models capable of differentiating between sufficiently dissimilar devices. (We examine various interpretations of *dissimilar*, such as devices that differ by type, manufacturer, and instance.) Separation between devices is one way of verifying app-device interactions; in other words, interactions for similar devices are classified to a specific class, whereas different interactions are classified into a different class (or classes).

Throughout our experiments, we use multi-class classifiers (e.g., Random Forest) to classify a sequence of packets into an appropriate class, and evaluate the performance of the classifiers by conducting a stratified 10-fold cross validation.

4.5.1 Experimental Setup

In this section we discuss the details of our testbed and dataset, how we triggered and captured authentic interactions between apps and devices, and details about our data pre-processing and analysis.

[12] https://crawdad.org/dartmouth/bluetooth-hci

A New Smart-Health and Smart-Home Testbed & Dataset

To evaluate our approach it was first necessary to assemble a testbed of devices. We focus on two broad categories of smart devices that are common for WPAN scenarios: smart-health devices and smart-home devices. We carefully selected devices to ensure that our testbed was composed of diverse devices, in terms of their functions; yet, we also wanted to study similar devices to evaluate the limitations of our approach to modeling, and the abilities of our models to differentiate between similar devices. In the text below, we describe devices by their *type* (which refers to a device's functionality and purpose), *make* (which refers to the manufacturer of the device), and *model* (which refers to an identifier, such as a name or number, that is used to distinguish among devices made by the same manufacturer).

At the time this work was conducted, our testbed consisted of 20 Bluetooth-enabled, smart-health and smart-home devices: two weight and body composition scales, each from different manufacturers; five blood-pressure monitors, from three different manufacturers;[13] three heart-rate monitors (two of which have identical make and model), from two different manufacturers; one pulse oximeter; one TENS unit (a unit is technically made up of two distinct Bluetooth devices); two glucose monitors, from two different manufacturers; two thermometers (same make, different models); two smart locks, each from different manufacturers; and two identical smart environment sensors (same make and model). A summary of the smart devices in our testbed is presented in Table 4.1; a photo of some of these devices is shown in Figure 4.10. Also, a summary of the mobile apps used to interact with the smart devices in our testbed is shown in Table 4.2.

[13]The blood-pressure monitors present any interesting case study. Not only does our dataset have three devices of the same type from three different manufacturers, but also, the monitors measure blood pressure at two different locations on the body: upper arm and wrist.

Figure 4.10: A photo of our smart-health and smart-home devices.

Data Collection: Capturing HCI Traces

Using our testbed, we collected a large number of network traces (more than 300 in total) that captured interactions between 20 distinct devices with 13 different smartphone apps. Here, a *trace* refers to a packet capture consisting of all packets that are observed between the time that a hub and peripheral device establish and terminate a connection; an *interaction* refers to a semantically meaningful exchange of packets between a hub and peripheral. As a result of these definitions, an interaction, in general, can be thought of as a subsequence of a trace. Based on how we constructed our traces (recall Section 4.4.3), however, a trace and an interaction are effectively equivalent in our experiments.

The above-mentioned smartphone apps were installed on a Nexus 5 smartphone running Android 6.0.1 ("Marshmallow"), API level 23, kernel version 3.4.0. Along with executing the apps, the smartphone also served as our primary device for data collection. To capture HCI traces, we enabled the Bluetooth *HCI snoop log* developer option. (This feature is a common developer option introduced in

Table 4.1: List of Bluetooth-enabled smart devices that make up our testbed.

Identifier	Device Model
Smart Health	
Scale-RENPHO	RENPHO Smart Bluetooth Body Fat Scale
Scale-Gurus	Bluetooth Smart Body Fat Scale by Weight Gurus
BPCuff-WR-iHealth	iHealth View Bluetooth Wrist Blood Pressure Monitor
BPCuff-UA-iHealth	iHealth Feel Bluetooth Upper Arm Blood Pressure Monitor
BPCuff-UA-OMRON	OMRON Evolv Wireless Upper Arm Blood Pressure Monitor
BPCuff-WR-OMRON	OMRON 10 Series Wireless Wrist Blood Pressure Monitor
BPCuff-UA-Choice	Choice Wireless Blood Pressure Monitor, Upper Arm
HRMonitor-CH-Polar-1	Polar H7 Wearable Heart Rate Monitor (Chest)
HRMonitor-CH-Polar-2	Polar H7 Wearable Heart Rate Monitor (Chest)
HRMonitor-CH-Zephyr	Zephyr Wearable Heart Rate Monitor (Chest)
PulseOx-iHealth	iHealth Air Wireless Fingertip Pulse Oximeter
TENS-OMRON	OMRON Avail Dual Channel TENS unit
Gluco-iHealth	iHealth Wireless Smart Blood Sugar Test Kit
Gluco-Choice	Choice Wireless Blood Glucose Monitor
Therm-Oral-KINSA	KINSA QuickCare (oral smart thermometer)
Therm-Ear-KINSA	KINSA Smart Ear (in-ear smart thermometer)
Smart Home	
SmartLock-August	August Smart Lock Pro + Connect (3^{rd} Gen.)
SmartLock-Schlage	Schlage Sense Smart Deadbolt
EnvSensor-Inkbird-1	Inkbird combo mini Bluetooth (temp/hum) sensor
EnvSensor-Inkbird-2	Inkbird combo mini Bluetooth (temp/hum) sensor

NOTE: Numbers are used in the identifier to distinguish between multiple instances of devices. Two letters are used in the identifier (between the device type and manufacturer) to denote the placement of a device where there are multiple options: WR = Wrist, UA = Upper Arm, CH = Chest.

Android 4.4. It is interesting to note that using this feature does not even require rooting the phone.) The HCI snoop log captures all Bluetooth HCI packets to a binary-encoded file, which it writes to an SD card; the log format resembles the Snoop Version 2 Packet Capture File Format described in RFC 1761 [45].

Each trace captured interactions between one app-device pair. Specifically, each trace captured *all* communications observed at the HCI layer (and therefore all protocol layers above the HCI layer). For each app-device pair we collected at least 10 traces, each of which included 3-10 minutes of network activity.

We observed that our traces also contain ambient network activity, such as advertisements from nearby devices not in our testbed. We filtered out this ambient activity by simply "ignoring" certain HCI EVT packets; as a result, ambient network activity does not interfere with our ability to learn verification models.

Table 4.2: List of smart device apps.

App	Corresponding Device(s)
Smart Health	
RENPHO	RENPHO scale
Weight Gurus	Weight Gurus scale
iHealth MyVitals	iHealth blood-pressure monitors, pulse oximeter
OMRON Connect	OMRON blood-pressure monitors
Choice Blood Pressure	Choice blood-pressure monitor
Polar Beat	Polar and Zephyr heart-rate monitors
OMRON TENS	OMRON TENS unit
iHealth Gluco-Smart	iHealth blood-glucose meter
AgaMatrix Diabetes Manager	Choice blood-glucose meter
Kinsa	Kinsa oral and ear thermometers
Smart Home	
Schlage Home	Schlage smart deadbolt
August Home	August smart lock
Engbird	Environment sensors
Bluetooth Dev/Test Apps	
nRF Connect	General BLE exploration
Light Blue Explorer	Schlage smart deadbolt
BLE Peripheral Simulator	August smart lock

Emulating Normal App-Device Interactions

We gathered HCI traces by manually using the apps and devices in our testbed to emulate a wide variety of normal app-device interactions. The actions we performed consisted of: navigating the "official" smartphone app[14] and exercising features that trigger network communication with a corresponding device, as well as acting upon the devices in such a way that triggers communication with its corresponding smartphone app.

It was not our intention to discover and exercise *every* functional feature (and thus every BLE service or characteristic) of a particular app/device. Rather, it was our intention to observe *typical* features and interactions between devices and their official app, which could be used to construct normality models suitable for performing verification in future app-device interactions.

Our current stance is that any network activity (authentic or inauthentic,

[14]With the exception of the heart-rate monitors, each device had a single "official" smartphone app to which it would connect.

benign or harmful) that deviates significantly from our normality models should fail verification, which provides an opportunity for further investigation or some other response to mitigate potential harm. In future work we will explore differences between benign and authentic anomalies (e.g., genuine app-device communication that occurs in response to a rare event, such as a heart attack), and anomalies that are potentially harmful or inauthentic.

Data Pre-processing

Prior to analysis we applied pre-processing to each raw trace file. At the time of our experiments, the pre-processing code was executed on a VM running Kali Linux, kernel version 4.14.

Data pre-processing is necessary to generate an intermediate file with per-packet features and labels. More specifically, after collecting each HCI trace (an *HCI snoop file*), we moved the raw file from the smartphone to a local VM using the Android Debugger command line tool (`adb`). We reset the HCI snoop file on the smartphone between each trace so that each HCI snoop file contained only packets belonging to interactions between a particular app-device combination (recall Section 4.5.1); we refer to this as an *app-device session*.

To parse HCI traces, we extended two open-source projects: `btsnoop` [15] and `bluepy`. [16] Our extensions extend the parsing of the HCI protocol and other protocols that the HCI protocol encapsulates; namely, we extract features for each packet within the HCI traces, such as packet types, lengths, endpoint identifiers, protocol semantics, and segmented headers and payloads belonging to higher-level Bluetooth protocols (e.g., the Attribute Protocol (ATT), the Security Management Protocol (SMP), the Signaling Protocol); and, because each trace captured a single app-device session, we labeled each packet according to the device it was sent

[15] https://github.com/traviswpeters/btsnoop
[16] https://github.com/traviswpeters/bluepy

to/from. Finally, these features and labels were written to CSV-formatted files for subsequent (offline) analysis.

Data Analysis & Model Validation

We conducted our analysis using a collection of custom-written Jupyter Notebooks [103]. To enable others to easily reproduce our findings, we wrote a custom Dockerfile that automates the process of building a consistent Docker [63] container environment that is suitable for loading our dataset and running our Notebooks. At the time of this work, our Jupyter Notebooks were executed on a Docker container running atop Linux, kernel version 4.9.125.

In our analysis, we loaded the pre-processed files into $m \times n$ matrices, where the rows are the observations (packets) in the trace, and the columns are the features (e.g., packet types, lengths, raw packet bytes) parsed from the raw trace files. Upon loading a processed trace file, we segmented each packet according to the hierarchical segmentation methodology described in Section 4.4.2 (specifically, recall Figure 4.5), and computed n-grams over the packet bytes in the respective segments. To be clear, for each leaf of hierarchical segmentation, our approach produces a 256-dimensional vector.

To learn models, we experimented with various multi-class classifiers (e.g., Random Forest), and different "branches" within hierarchical segmentation, to classify a sequence of packets into one of multiple classes. (In our experiments we examined various definitions for classes, such as separate classes for each device type, type *and* make, and instance.) In the subsections below, we present experimental results for the multi-class Random Forest classifiers, which were trained and validated with n-grams computed from the HCI$\rightarrow$ACL$\rightarrow$L2CAP$\rightarrow \cdots$ branch of hierarchical segmentation.

To evaluate the models produced by the Random Forest classifier examined in

this work, we conducted a stratified 10-fold cross validation that repeatedly split our dataset into training and testing subsets, and measured the classifier's performance. The classifier was trained with n-grams and class labels from a training set. After training, we used the trained models to predict a class label for each sample in a corresponding test set (i.e., the n-grams computed from a new trace) and compared the predicted label to the actual label.

Due to limitations of our dataset, such as imbalances in the class sizes, we employed model validation techniques (n-fold cross validation and stratification) to ensure that our analysis was accurate and reliable. N-fold cross validation averages the performance of a classifier over multiple partitions (folds) of the data, which mitigates the effects of an (un)favorable partition (and potentially unreliable conclusions about the performance of the classifier). Stratification ensures that the various folds maintain a balanced distribution of the represented classes when partitioning the data.

In our evaluation, we used off-the-shelf classifiers (e.g., Random Forest) built into scikit-learn [151], a popular python package for machine learning. In our evaluation, we use classifiers that can be applied both as binary classifiers (two classes) and multi-class classifiers (more than two classes). While we evaluated VIA using a few such classifiers, we focused our attention on the Random Forest classifier because they are generally easier to interpret and explain, they are non-parametric (so there is no need to worry about outliers or whether the data is linearly separable), and they are fast and scalable (so there is generally less concern with parameter tuning). Because this work is more concerned with validating the approach, and not so much fine-tuning the underlying machine learning classifiers, these attributes make the Random Forest classifier a reasonable starting point.

In the binary classification experiments, classifiers labeled samples as the

target class ("positive" class) or the *other* class ("negative" class).[17] In the multi-class classification experiments, classifiers labeled samples according to a *target* class that the sample most closely resembled.

To report our results, we use the following performance measures, which are useful for evaluating our classifiers in performing both binary classification and multi-class classification tasks:

- **Precision** is the ratio of correctly predicted positive observations to the total predicted positive observations. A perfect classifier performs at 100%.

$$\text{Precision} = \frac{tp}{tp + fp}$$

- **Recall** is the ratio of correctly predicted positive observations to the total number of observations that are actually in the positive class. A perfect classifier performs at 100%.

$$\text{Recall} = \frac{tp}{tp + fn}$$

- **F1-score** is the weighted average of precision and recall. A perfect classifier performs at 100%.

$$\text{F1-score} = \frac{2 * (precision * recall)}{precision + recall}$$

In the above definitions, *tp* refers to true positives, *tn* refers to true negatives, *fp* refers to false positives, and *fn* refers to false negatives. The precision and recall provide insights into the types of errors that the classifiers make, which is preferable over simply reporting the number of misclassified samples. Accuracy is another

[17] We use "positive" here because this intuitively maps to a successful identification/verification, whereas "negative" (or "other") intuitively maps to an unsuccessful identification/verification.

common performance measure, but accuracy alone can yield misleading results (e.g., when classes are imbalanced). The F1-score generally provides a more robust measure of the performance of a classifier; for this reason, we opt to report the F1-score and not the accuracy.

Reproducibility

To promote open access research and reproducibility of this scientific work, and to equip future researchers with better tools for investigation into Bluetooth, we will release the Vagrantfile and Dockerfile for the respective VirtualBox VM and Docker container, as well as our dataset, after this dissertation is released. Our extensions to the two open-source projects mentioned above are already publicly available (see links in footnotes 15 and 16).

4.5.2 Overview of Experiments & Results

We evaluated VIA in terms of two fundamental approaches to authentication (recall Section 4.2.1); specifically, we evaluated VIA's ability to perform identification (one-to-many matching) and verification (one-to-one matching) tasks. Moreover, we evaluated VIA's ability to perform these tasks at different granularities. In our evaluation we considered three granularities of separation between devices: by device type, by device type *and* make, and by specific device instance.[18] The specific classification granularities and the corresponding labels are summarized in Table 4.3.

Table 4.3: A list of class labels by classification granularity.

Granularity (# of Labels)	Labels
Device Type (9)	*Scale, BP-Monitor, HR-Monitor, Pulse-Ox, TENS, Glucose-Meter, Thermometer, Smart-Lock, ENV-Sensor*
Device Type+Make (15)	*Scale-RENPHO, Scale-Gurus, BP-Monitor-iHealth, BP-Monitor-OMRON, BP-Monitor-Choice, HR-Monitor-Polar, HR-Monitor-Zephyr, Pulse-Ox-iHealth, TENS-OMRON, Glucose-Meter-iHealth, Glucose-Meter-Choice, Thermometer-KINSA, Smart-Lock-August, Smart-Lock-Schlage, ENV-Sensor-Inkbird*
Device Instance (20)	*Scale-RENPHO, Scale-Gurus, BP-Monitor-iHealth-1, BP-Monitor-iHealth-2, BP-Monitor-OMRON-1, BP-Monitor-OMRON-2, BP-Monitor-Choice, HR-Monitor-Polar-1, HR-Monitor-Polar-2, HR-Monitor-Zephyr, Pulse-Ox-iHealth, TENS-OMRON, Glucose-Meter-iHealth, Glucose-Meter-Choice, Thermometer-KINSA-1, Thermometer-KINSA-2, Smart-Lock-August, Smart-Lock-Schlage, ENV-Sensor-Inkbird-1, ENV-Sensor-Inkbird-2*

NOTE: Numbers are used in the identifier to distinguish between multiple instances of devices.

[18] We do not evaluate the granularity of device type, make, and model (*device-type-make-model*) here. While it may seem a natural next step, our testbed did not contain enough examples to provide a meaningful evaluation at this class granularity. We refer the reader to Section 4.5.5, where we evaluate VIA using subsets of devices from our testbed that do have variance in device model.

Our results can be summarized as follows:

- **Identification Tasks:** VIA successfully identified devices by *device-type* with an F1-score of 94% or higher in all cases. VIA successfully identified devices by *device-type-make* with an F1-score of 94% or higher in all cases. VIA successfully identified devices by *device-instance* with an F1-score of 91% or higher in most cases.

- **Verification Tasks:** VIA successfully verified that specific network communications belong to a target *device-type* class with F1-score higher than 91% in all cases. VIA successfully verified that specific network communications belong to a target *device-type-make* class with F1-score higher than 91% in all cases. VIA successfully verified that specific network communications belong to a target *device-instance* class with F1-score higher than 93% in all but one case.

We also conducted experiments to examine the cases where the F1-score is lower (e.g., environment sensors), and concluded that these cases arise because VIA, in some instances, confused distinct instances of otherwise identical devices (i.e., same device type, make, and model) or devices that are nearly identical (i.e., same device type and make). We discuss our findings in more detail below.

4.5.3 Identification Experiments & Results

In this section we present our evaluation of VIA in terms of its ability to perform identification tasks. Specifically, we considered three different granularities of separation between devices (noted above), and evaluated VIA's ability to correctly identify devices at each of the three granularities.

Identification is a multi-class classification task (recall Section 4.2.1): a device's identity is initially unknown, and (depending on the experiment) VIA seeks to determine the *device-type*, *device-type-make*, or *device-instance* of a device by examining the app-device network communications. First, VIA obtains many samples from a population of devices. Each device has an identity in the form of a `BD_ADDR` (though other forms of identity are possible, such as `LTK`), as well as a set of typical network interactions. While other strategies are possible, and other features may provide useful information, in our work network interactions are currently represented with n-grams. Without loss of generality, we refer to some representation of typical network interactions as a *device profile*.

When VIA observes a new network interaction between an app and device with an unknown identity, VIA can use its device profiles to identify the device from the original population. Simply put, VIA monitors network communications to formulate a profile for the observed traffic, and then examines its collection of device profiles to determine the device profile that best matches the unknown device's profile. The unknown device is assigned the identity that corresponds to the device profile it most closely resembles.[19]

In identification tasks, successful identification is defined as a classification

[19] To date, we have not yet evaluated how VIA performs when presented with an unknown device (i.e., a device not seen in training). In VIA's current design, any observation from an unknown device will be classified to the most similar device class from the training set, which is not ideal. One solution to this issue is to implement a confidence threshold. If the threshold is not exceeded, VIA would classify the observation as "unknown," meaning that the observation does not look sufficiently similar to any device in its known population of devices. We plan to add and evaluate this enhancement to VIA's identification capabilities in future work.

into the correct (positive) target class; classification into *any* other class is recorded as an unsuccessful identification (negative). The results for VIA and the multi-class classification tasks are summarized in Table 4.4, Table 4.5, and Table 4.6.

Identification Task #1: Classification by Device Type

In this experiment we investigated whether VIA can identify the specific device type (*device-type*) to which a new sample belongs. To accomplish this goal, we assigned each trace a label according to the type of device (recall Table 4.3). Our testbed consisted of 9 unique classes for device type (i.e., we created distinct classes where devices are separated by type; devices of the same type shared a label). We then conducted a stratified 10-fold cross-validation using models learned from data labeled by device type. The cross-validation results for each of the 9 *device-type* classes are summarized in Table 4.4 and visualized in Figure 4.11. In the *device-type* identification tasks, VIA achieved an F1-score of 94% or better in all *device-type* classes.

Device Type	Precision	Recall	F1-score
BP Monitor	1.00	1.00	1.00
Environment Sensor	0.89	1.00	0.94
Glucose Monitor	1.00	0.96	0.98
HR Monitor	1.00	1.00	1.00
Pulse Oximeter	1.00	1.00	1.00
Smart Lock	1.00	1.00	1.00
Smart Scale	1.00	1.00	1.00
TENS Unit	1.00	1.00	1.00
Thermometer	1.00	0.97	0.98

Table 4.4: The precision, recall, and F1-score for each device type (*device-type*) class. The corresponding confusion matrix is shown in Figure 4.11.

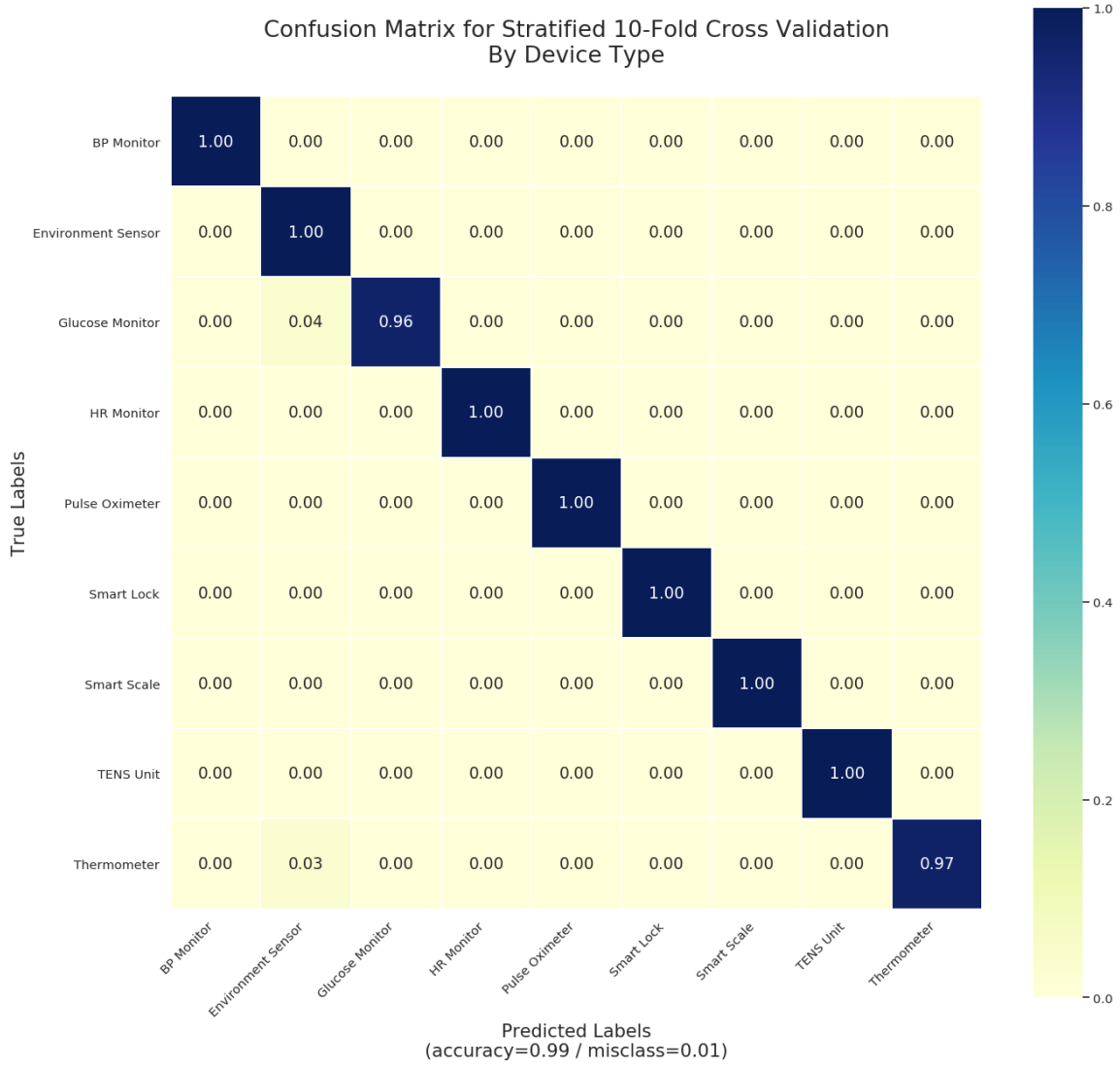

Figure 4.11: Confusion matrix for classification of devices by type (*device-type*). While our smart-device testbed contained 20 devices in total, some devices are functionally similar (same type). In this experiment, our testbed was separated into 9 different classes based on device type. This confusion matrix shows the success of classification when devices were combined into a class if they were of the same type.

Identification Task #2: Classification by Device Type+Make

In this experiment we investigated whether VIA can identify the specific device type and make (*device-type-make*) to which a new sample belongs. To accomplish this goal, we assigned each trace a label according to the specific device type and make to which it belongs (recall Table 4.3). Our testbed consisted of 15 unique classes for device type and make (i.e., we created distinct classes where devices were separated by type *and* make; devices of the same type and make shared a label). We then conducted a stratified 10-fold cross-validation using models learned from data labeled by their device type and make. The cross-validation results for each of the 15 *device-type-make* classes are summarized in Table 4.5 and visualized in Figure 4.12. In the *device-type-make* identification tasks, VIA achieved an F1-score of 94% or better in all *device-type-make* classes.

Device Type & Make	Precision	Recall	F1-score
BP Monitor - Choice	1.00	1.00	1.00
BP Monitor - Omron	1.00	1.00	1.00
BP Monitor - iHealth	1.00	1.00	1.00
Environment Sensor - Inkbird	0.89	1.00	0.94
Glucose Monitor - Choice	1.00	1.00	1.00
Glucose Monitor - iHealth	1.00	0.91	0.95
HR Monitor - Polar	1.00	1.00	1.00
HR Monitor - Zephyr	1.00	1.00	1.00
Pulse Oximeter - iHealth	1.00	1.00	1.00
Smart Lock - August	0.95	1.00	0.98
Smart Lock - Schlage	1.00	0.95	0.98
Smart Scale - Gurus	1.00	1.00	1.00
Smart Scale - Renpho	1.00	1.00	1.00
TENS Unit - Omron	1.00	1.00	1.00
Thermometer - Kinsa	1.00	0.97	0.98

Table 4.5: The precision, recall, and F1-score for each device type and make (*device-type-make*) class. The corresponding confusion matrix is shown in Figure 4.12.

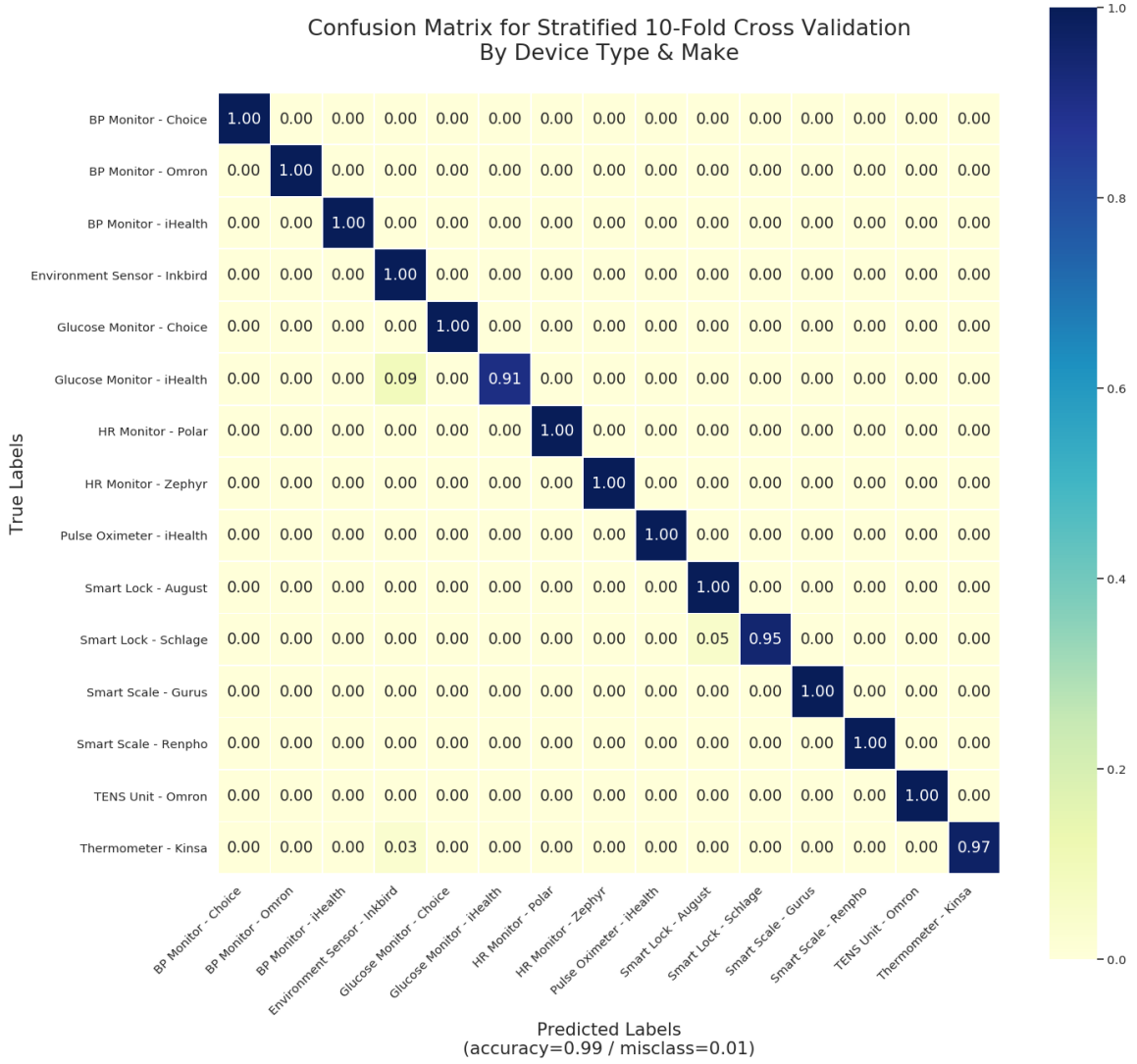

Figure 4.12: Confusion matrix for classification of devices by type and make (*device-type-make*). While our smart-device testbed contained 20 devices in total, some devices were functionally similar (same type) but may or may not have been made by the same manufacturer. In this experiment, our testbed was separated into 15 different classes based on type *and* make. This confusion matrix shows the success of classification when devices were combined into a class if they were of the same type and make.

Identification Task #3: Classification by Device Instance

In this experiment we investigated whether VIA can identify the specific device (*device-instance*) to which a new sample belongs. To accomplish this goal, we assigned each trace a label according to the specific device to which it belongs (recall Table 4.3). Our testbed consisted of 20 unique classes based on the number of distinct device instances (i.e., we created a distinct class for each distinct device instance, even if the devices were the same type, make, and model). We then conducted a stratified 10-fold cross-validation using models learned from data labeled by their device instances. The cross-validation results for each of the 20 *device-instance* classes are summarized in Table 4.6 and visualized in Figure 4.13. In the *device-instance* identification tasks, VIA achieved an F1-score of 91% or better in nearly all *device-instance* classes. Indeed, the only cases where VIA performed worse were tasks where VIA had to distinguish between nearly identical devices (same type, make, and model). We discuss this observation further in the Discussion section below.

Device Instance	Precision	Recall	F1-score
BP Monitor - Choice (Upper Arm)	1.00	1.00	1.00
BP Monitor - Omron (Upper Arm)	1.00	1.00	1.00
BP Monitor - Omron (Wrist)	1.00	1.00	1.00
BP Monitor - iHealth (Upper Arm)	1.00	0.92	0.96
BP Monitor - iHealth (Wrist)	0.94	1.00	0.97
Environment Sensor - Inkbird (1)	0.33	0.14	0.20
Environment Sensor - Inkbird (2)	0.50	0.78	0.61
Glucose Monitor - Choice (1)	1.00	1.00	1.00
Glucose Monitor - iHealth (1)	0.91	0.91	0.91
HR Monitor - Polar (1)	0.92	1.00	0.96
HR Monitor - Polar (2)	1.00	1.00	1.00
HR Monitor - Zephyr (1)	1.00	0.92	0.96
Pulse Oximeter - iHealth (1)	1.00	1.00	1.00
Smart Lock - August (1)	0.91	1.00	0.95
Smart Lock - Schlage (1)	1.00	0.90	0.95
Smart Scale - Gurus (1)	1.00	1.00	1.00
Smart Scale - Renpho (1)	1.00	1.00	1.00
TENS Unit - Omron (1)	1.00	1.00	1.00
Thermometer - Kinsa (Ear)	1.00	1.00	1.00
Thermometer - Kinsa (Oral)	1.00	0.94	0.97

Table 4.6: The precision, recall, and F1-score for each device instance (*device-instance*) class. The corresponding confusion matrix is shown in Figure 4.13.

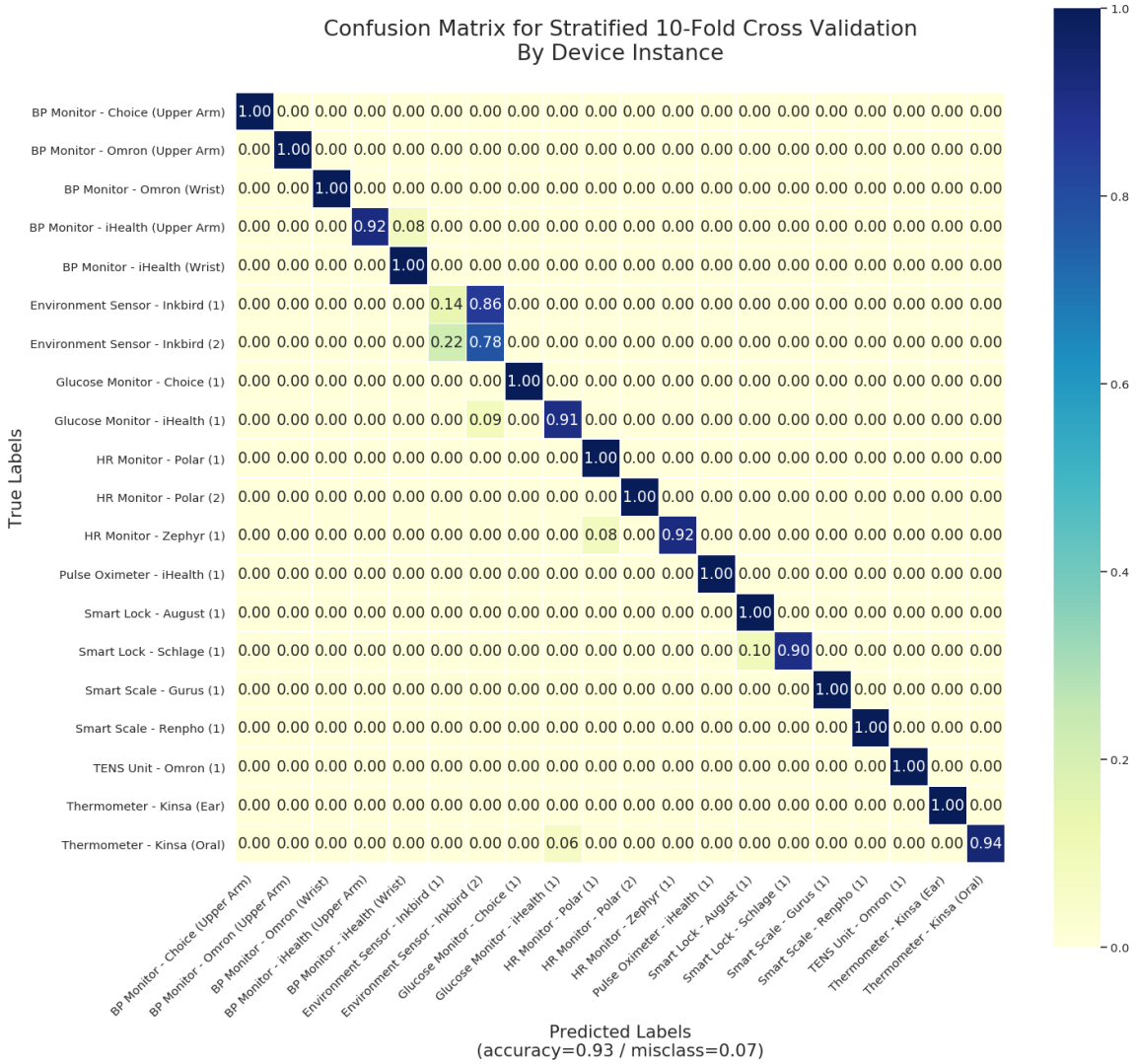

Figure 4.13: Confusion matrix for classification of devices by instance. Our smart-device testbed contained 20 devices (distinct *instances*) in total. In this experiment, our testbed was separated into 20 different classes based on the number of distinct device instances. This confusion matrix shows the success of classification when each device constituted its own class.

Discussion

In our evaluation of VIA's ability to perform identification tasks, it is important to understand that the F1-score is lower in the *device-instance* evaluation because similar devices are in both the positive and negative classes. For example, the *Environment Sensor* devices present the one case where the F1-scores are well below 90%. We chose to include examples of *identical* devices (i.e., same device type, make, and model) in this testbed to study this specific case: *how well does VIA perform correct identification when some devices are highly similar (or identical, as in this case)?* Unsurprisingly, these identical devices run the same software, on different (but similar) hardware, which yields profiles for app-device communications that are nearly identical. As a result, VIA confused these devices when performing identification.

One could conclude that VIA's ability to perform identification is best done at the granularity of *device-type* or *device-type-make*; however, we urge the reader to consider the consequences of this conclusion. For example, the consequence of identification at this granularity is that VIA may not be able to distinguish between different instances of devices with the same type, make and model. Furthermore, we would be remiss to not point out that VIA's perceived success in performing identification – specifically in distinguishing between different instances of devices – may be exaggerated by the fact that our testbed contained few devices that shared the same type, make *and* model – the cases where VIA is most likely to be confounded.

In light of these observations, we conclude that further analysis is needed to better understand the strengths and limitations of VIA. To this end, we performed additional experiments with subsets of the highly-similar devices that we did have in our testbed. We present our findings in Section 4.5.5.

4.5.4 Verification Experiments & Results

In this section we present our evaluation of VIA in terms of its ability to perform verification tasks. As before, we considered three different granularities of separation between devices (noted above), and evaluated VIA's ability to correctly verify devices at each of the three granularities.

Verification is a binary (two-class) classification task (recall Section 4.2.1): a device's identity is known or observed a priori, which enables VIA to load a target identity for comparison. After examining newly-observed app-device network communications, VIA either accepts or rejects that the observations are sufficiently similar to the target identity. As mentioned above, we considered three different definitions (granularities) for identity: *device-type*, *device-type-make*, or *device-instance*.

To accomplish verification, VIA first obtains many samples from a population of devices and learns a specific classifier for each device. Each device has an identity in the form of a `BD_ADDR` (though other forms of identity are possible, such as `LTK`), as well as a set of typical network interactions, which makes it possible to train a classifier that recognizes that device. While other strategies are possible, and other features may provide useful information, in our work network interactions are currently represented with n-grams. Without loss of generality, we refer to some representation of typical network interactions as a *device profile*.

When VIA observes a new network interaction between an app and device with a known identity, VIA (1) uses the identity to load the appropriate classifier, and (2) uses the classifier to determine whether or not the new communications are sufficiently similar to the target identity. Simply put, VIA monitors network communications, obtains a claimed-identity for app-device interactions, loads the corresponding classifier (which was obtained from training using the device profile for the target identity, henceforth referred to as the *verification profile*), formulates a

profile for newly-observed traffic (the *test profile*), and uses the classifier to conduct a verification procedure that determines whether the test profile is sufficiently similar to the verification profile.[20] In the end, VIA's verification procedure relies on the classifier to classify new observations into one of two classes: the *target* class or the *other* class. If the interactions are actually from a device that matches the identity of the target identity, the interactions should be classified to the *target* class, and VIA accepts the observations as being authentic. If, however, the observations are classified to the *other* class, VIA rejects the observations and and deems them to be inauthentic.

In the experiments conducted in the following sections, we repeatedly fixed a single class to be the *target* class and combine all other classes into a single *other* class; this formulation creates the desired two-class classification problem.[21] We then evaluated the verification results produced by VIA using the following definitions. A successful verification is defined as a classification of observations belonging to the *target* class into the *target* class (true positive), or a classification of observations belonging to the *other* class into the *other* class (true negative). An unsuccessful verification is defined as a classification of observations belonging to the *target* class into the *other* class (false negative), or a classification of observations belonging to the *other* class into the *target* class (false positive). The results for VIA and the binary-class classification tasks are summarized in Table 4.7, Table 4.8, and Table 4.9.

[20] Thus, VIA's ability to determine whether observations are 'sufficiently similar' with past observations is dependent on the underlying classifier that is used, and the classifier's ability to create a distinct decision boundary between the *target* class and the *other* class. (Recall Section 4.5.1.)

[21] To date, we have not yet evaluated how VIA performs when presented with an unknown device (i.e., a device not seen in training). In VIA's current design, any observation from an unknown device will be classified to the class it most closely resembles; ideally this class would be the *other* class, but since the unknown observation is represented in neither the *target* class nor the *other* class, VIA's behavior in this scenario is unknown. One solution to this issue is to implement a confidence threshold before allowing an observation to be classified into the *target* class. If the new observation does not exceed the confidence threshold, VIA would classify the observation into the *other* class, meaning that the observation does not look sufficiently similar to the *target* class for verification. We plan to add and evaluate this enhancement to VIA's verification capabilities in future work.

Verification Task #1: Classification by Device Type

In this experiment we investigated whether VIA can verify that a new sample belongs to a specific device type (*device-type*). To evaluate verification by device type, we re-labeled data (recall Table 4.3) according to a *one-vs-all* strategy: We fixed one device type (e.g., BP Monitor) to be the *target* class for verification, and grouped the remaining device types into an *other* class; traces associated with the fixed-type were labeled as members of the *target* class, and all other traces (associated with all of the other types) were labeled as members of the *other* class. After re-labeling the data using the one-vs-all strategy, we conducted a stratified 10-fold cross-validation using models learned from the re-labeled data. We repeated this process 9 times to produce results where each device type was fixed as the *target* class. The cross-validation results for each of the 9 *device-type* classes are summarized in Table 4.7 and visualized in Figure 4.14. In the *device-type* verification tasks, VIA achieved an F1-score of 91% or better in all *device-type* classes.

Device Type	Precision	Recall	F1-score
BP Monitor	1.00	0.96	0.98
Environment Sensor	1.00	0.94	0.97
Glucose Monitor	1.00	0.96	0.98
HR Monitor	1.00	0.91	0.95
Pulse Oximeter	1.00	1.00	1.00
Smart Scale	1.00	0.90	0.95
Smart Lock	1.00	1.00	1.00
TENS Unit	1.00	0.83	0.91
Thermometer	1.00	0.97	0.98

Table 4.7: The precision, recall, and F1-score for one-vs-all verification by device type (*device-type*). The corresponding confusion matrices are shown in Figure 4.14.

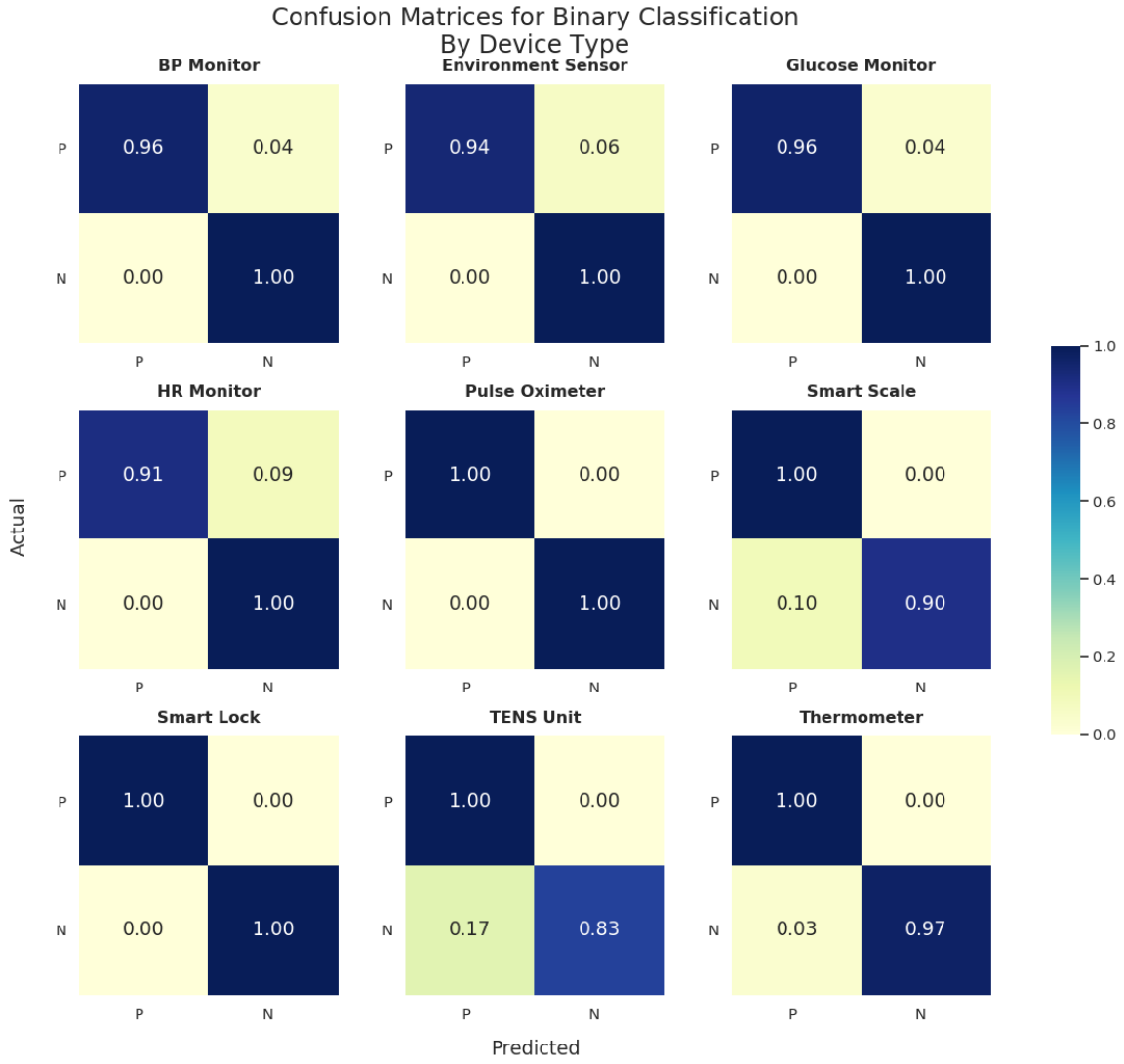

Figure 4.14: Confusion matrices for one-vs-all verification by device type (*device-type*). While our smart-device testbed contained 20 devices in total, some devices were functionally similar (same type). In this experiment, our testbed was separated into 9 different classes based on device type. The confusion matrices show the success of verification when one type was fixed and all other types are combined into a single class.

Verification Task #2: Classification by Device Type+Make

In this experiment we investigated whether VIA can verify that a new sample belongs to a specific device type and make (*device-type-make*). To evaluate verification by device type and make, we re-labeled data (recall Table 4.3) according to a *one-vs-all* strategy: We fixed one device type and make (e.g., BP Monitor - iHealth) to be the *target* class for verification, and grouped the remaining device types and makes into an *other* class; traces associated with the fixed type and make were labeled as members of the *target* class, and all other traces (associated with all of the other types and makes) were labeled as members of the *other* class. After re-labeling the data using the one-vs-all strategy, we conducted a stratified 10-fold cross-validation using models learned from the re-labeled data. We repeated this process 15 times to produce results where each device type and make was fixed as the *target* class. The cross-validation results for each of the 15 *device-type-make* classes are summarized in Table 4.8 and visualized in Figure 4.15. In the *device-type-make* verification tasks, VIA achieved an F1-score of 91% or better in nearly all *device-type-make* classes. Indeed, the only cases where VIA performed worse were tasks where VIA had to perform verification when similar devices (e.g., blood-pressure monitors) were in both the *target* class *and* the *other* class. We discuss this observation further in the Discussion section below.

Device Type & Make	Precision	Recall	F1-score
BP Monitor - Choice	1.00	1.00	1.00
BP Monitor - iHealth	0.75	0.93	0.83
BP Monitor - Omron	1.00	1.00	1.00
Environment Sensor - Inkbird	1.00	0.94	0.97
Glucose Monitor - Choice	1.00	0.88	0.93
Glucose Monitor - iHealth	1.00	0.91	0.95
HR Monitor - Polar	1.00	0.91	0.95
HR Monitor - Zephyr	1.00	0.83	0.91
Pulse Oximeter - iHealth	1.00	1.00	1.00
Smart Scale - Gurus	1.00	0.90	0.95
Smart Scale - Renpho	1.00	0.90	0.95
Smart Lock - August	0.87	1.00	0.93
Smart Lock - Schlage	1.00	0.86	0.92
TENS Unit - Omron	1.00	0.92	0.96
Thermometer - Kinsa	1.00	0.97	0.98

Table 4.8: The precision, recall, and F1-score for one-vs-all verification by device type and make (*device-type-make*). The corresponding confusion matrices are shown in Figure 4.15.

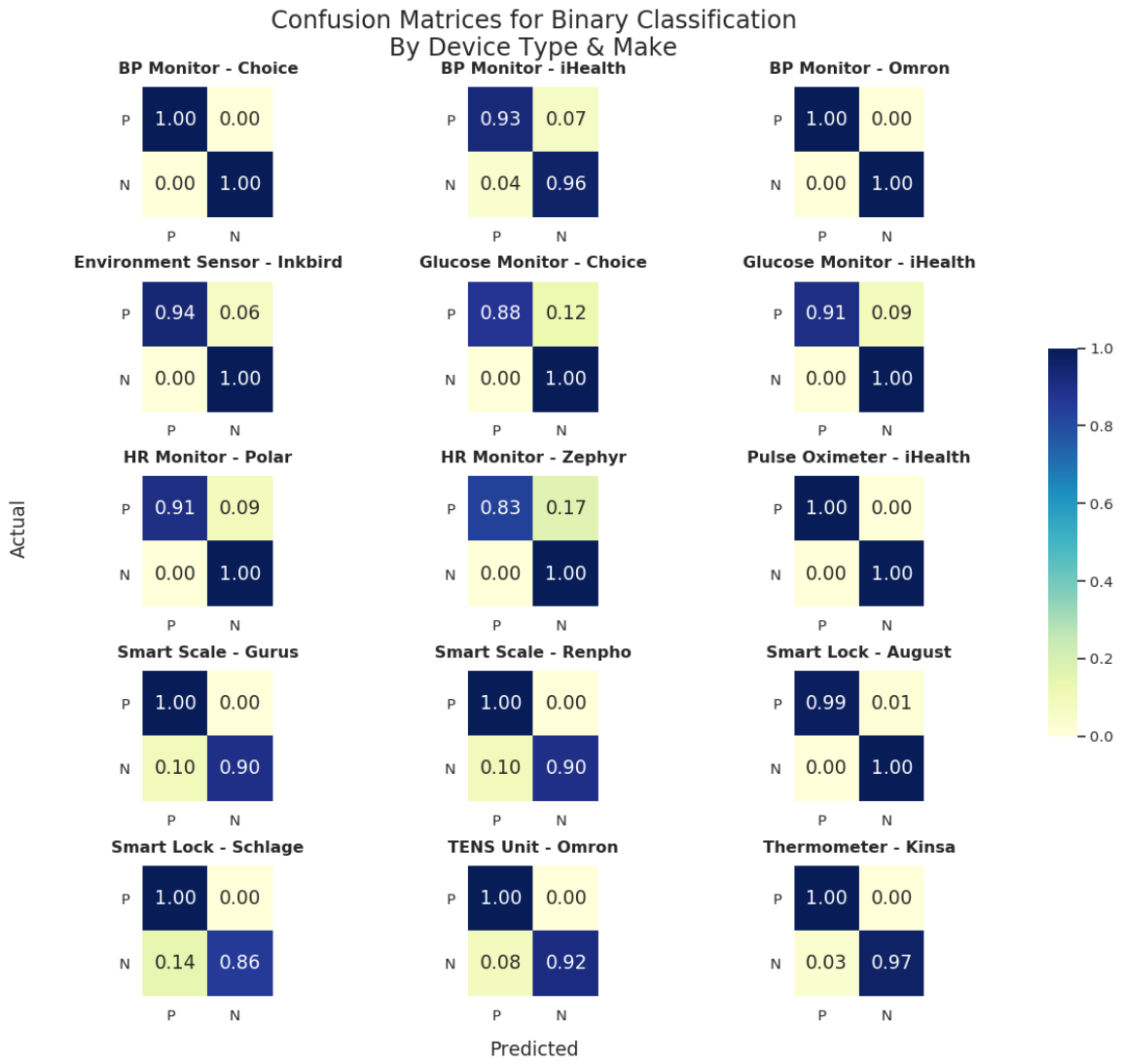

Figure 4.15: Confusion matrices for one-vs-all verification by device type and make (*device-type-make*). While our smart-device testbed contained 20 devices in total, some devices were functionally similar (same type) but may or may not have been made by the same manufacturer. In this experiment, our testbed was separated into 15 different classes based on type *and* make. The confusion matrices show the success of verification when a type and make was fixed and all other types and makes were combined into a single class.

Verification Task #3: Classification by Device Instance

In this experiment we investigated whether VIA can verify that a new sample belongs to a specific device (*device-instance*). To evaluate verification by device instance, we re-labeled data (recall Table 4.3) according to a *one-vs-all* strategy: We fixed one instance (e.g., Environment Sensor - Inkbird (1)) to be the *target* class for verification, and grouped the remaining device instances into an *other* class; traces associated with the fixed instance were labeled as members of the *target* class, and all other traces (associated with all of the other instances) were labeled as members of the *other* class. After re-labeling the data using the one-vs-all strategy, we conducted a stratified 10-fold cross-validation using models learned from the re-labeled data. We repeated this process 20 times to produce results where each device instance is fixed as the *target* class. The cross-validation results for each of the 20 *device-instance* classes are summarized in Table 4.9 and visualized in Figure 4.16. In the *device-instance* verification tasks, VIA achieved an F1-score of 93% or better in nearly all *device-instance* classes. Indeed, the only cases where VIA performed worse were tasks where VIA had to perform verification when nearly identical devices (e.g., environment sensors, heart-rate monitors) were in both the *target* class *and* the *other* class. We discuss this observation further in the Discussion section below.

Device Instance	Precision	Recall	F1-score
BP Monitor - Choice (Upper Arm)	1.00	1.00	1.00
BP Monitor - iHealth (Upper Arm)	1.00	0.92	0.96
BP Monitor - iHealth (Wrist)	1.00	0.88	0.94
BP Monitor - Omron (Upper Arm)	1.00	1.00	1.00
BP Monitor - Omron (Wrist)	1.00	1.00	1.00
Environment Sensor - Inkbird (1)	0.20	0.14	0.17
Environment Sensor - Inkbird (2)	0.40	0.22	0.29
Glucose Monitor - Choice (1)	1.00	0.94	0.97
Glucose Monitor - iHealth (1)	1.00	0.91	0.95
HR Monitor - Polar (1)	1.00	0.45	0.62
HR Monitor - Polar (2)	1.00	0.09	0.17
HR Monitor - Zephyr (1)	1.00	0.83	0.91
Pulse Oximeter - iHealth (1)	1.00	1.00	1.00
Smart Scale - Gurus (1)	1.00	0.90	0.95
Smart Scale - Renpho (1)	1.00	0.90	0.95
Smart Lock - August (1)	0.87	1.00	0.93
Smart Lock - Schlage (1)	1.00	0.90	0.95
TENS Unit - Omron (1)	1.00	0.92	0.96
Thermometer - Kinsa (Ear)	1.00	1.00	1.00
Thermometer - Kinsa (Oral)	1.00	0.94	0.97

Table 4.9: The precision, recall, and F1-score for one-vs-all verification by device instance (*device-instance*). The corresponding confusion matrices are shown in Figure 4.16.

Figure 4.16: Confusion matrices for one-vs-all verification by device instance (*device-instance*). Our smart-device testbed contained 20 devices (distinct *instances*) in total. In this experiment, our testbed was separated into 20 different classes based on the number of distinct device instances. The confusion matrices show the success of verification when a specific device instance was fixed and all other instances were combined into a single class.

Discussion

In our evaluation of VIA's ability to perform verification tasks, it is important to understand that the F1-score is lower in the *device-type-make* evaluation and the *device-instance* evaluation because similar (sometimes nearly identical) devices are in both the *target* and *other* classes. For instance, when attempting to verify an iHealth blood-pressure monitor in the *device-type-make* evaluation, there are blood-pressure monitors in both the *target* class (all iHealth blood-pressure monitors) and *other* class (Choice and Omron blood-pressure monitors). As another example, when attempting to verify an Inkbird environment sensor in the *device-instance* evaluation, there is one instance of an environment sensor in both the *target* class (Environment Sensor - Inkbird (1)) and *other* class (Environment Sensor - Inkbird (2)). As we noted above in the discussion of identification tasks, these particular observations are not surprising. The similar devices – and indeed the nearly identical devices – operate in similar ways, and in some cases even run the same software, on different (but similar) hardware. It is therefore unsurprising to observe that VIA confused these devices when performing certain verification tasks.

One could conclude that VIA's verification is best done at the granularity of *device-type*; however, we urge the reader to consider the consequences of this conclusion. For example, the consequence of verification at the granularity of *device-type* is that VIA may not take into account important distinctive features that arise from devices with different make, model, or instance. Furthermore, we would be remiss to not point out that VIA's perceived success in performing verification may be exaggerated by the fact that our testbed contained few devices that shared the same type, make *and* model – the cases where VIA is most likely to be confounded.

In light of these observations, we conclude that further analysis is needed to better understand the strengths and limitations of VIA. To this end, we performed additional experiments with subsets of the highly-similar devices that we did have

in our testbed. We present our findings in the next section.

4.5.5 Similarity Experiments & Results

Our previous experiments show that our models are exceptionally accurate in many identification and verification tasks, but they also illustrate interesting cases where our models may not perform well. In the experiments we discuss next, we examine cases where devices were highly similar (e.g., same device type, same device make, same device model).

Blood-Pressure Monitors

For this experiment, we used five different blood-pressure monitors from three different manufacturers (Omron, iHealth, and Choice) and conducted a comparative analysis among the five devices. We trained and evaluated models based on these five classes, similar to the identification tasks above. The cross-validation results are summarized in Table 4.10 and visualized in Figure 4.17.

It is clear from the confusion matrix in Figure 4.17 that our models result in highly-accurate classification. In other words, our approach shows coherent differentiation between the models learned for devices from different manufacturers. It is also evident, however, that there was confusion between similar devices made by the same manufacturer. In other words, similar types of devices made by the same manufacturer may result in highly similar device profiles. For instance, in this experiment, we used two blood-pressure monitors from iHealth; one takes measurements from the wrist (WR) while the other takes measurements from the upper arm (UA). From the observed network traffic between the iHealth app and the iHealth blood-pressure monitors, these devices appeared to be nearly identical. (See Figure A.2 and Figure A.3 in Appendix A for detailed depictions of their respective device profiles.) A similar discussion applies for the Omron WR

and UA devices. This result suggests that VIA will have difficulty distinguishing between devices of the same type, made by the same manufacturer. This outcome is not unexpected, however. It makes sense that device makers would use similar hardware and software (perhaps even *the same* hardware and software) to build their various devices. Thus, our takeaway is that VIA is able to verify that traces belonging to similar devices from the same manufacturer are consistent with a previously learned model. Indeed, if we were to collapse the confusion matrix in Figure 4.17 down to separation by manufacturer, we would see perfect separation.

We conclude that even though the devices themselves are similar in terms of the function they provide to the end-user (blood-pressure measurements), this experiment demonstrates that VIA can reliably distinguish between functionally-similar devices that are made by different manufacturers. This strongly suggests that it would be difficult for adversaries to successfully masquerade themselves as authentic devices *at the same time* that they carry out any other nefarious actions (since even highly similar devices are rarely misclassified).

Blood-Pressure Monitor Instance	Precision	Recall	F1-score
BP Monitor - Choice (Upper Arm)	1.00	1.00	1.00
BP Monitor - Omron (Upper Arm)	1.00	1.00	1.00
BP Monitor - Omron (Wrist)	1.00	1.00	1.00
BP Monitor - iHealth (Upper Arm)	1.00	0.92	0.96
BP Monitor - iHealth (Wrist)	0.94	1.00	0.97

Table 4.10: The corresponding precision, recall, and F1-score for the confusion matrix shown in Figure 4.17.

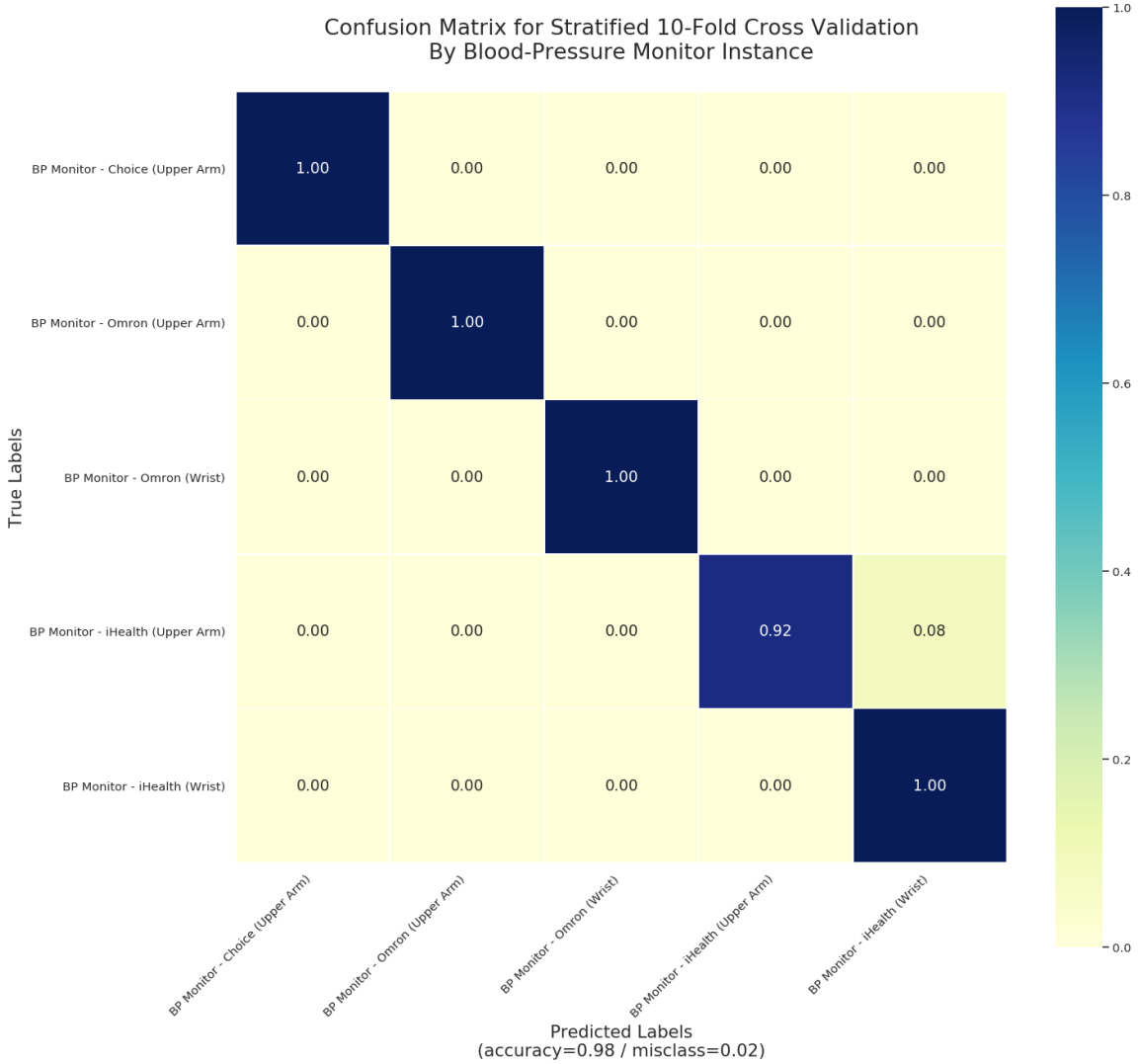

Figure 4.17: Confusion matrix for classification of blood-pressure monitors. Our smart-device testbed contains five different blood-pressure monitors from three different manufacturers, with a variety of measurement locations (i.e., wrist vs. upper arm monitors).

Environment Sensors

For this experiment, we used two Inkbird environment sensors to investigate VIA's ability to distinguish among distinct instances of the same device (i.e., devices that are identical in their hardware, firmware, and software). We trained and evaluated models based on these two classes, similar to the identification tasks above. The cross-validation results are summarized in Table 4.11 and visualized in Figure 4.18.

It is clear from the confusion matrix in Figure 4.18 that VIA struggled to distinguish between these highly similar devices. Once again, this result is not surprising. These devices are identical in terms of their software and hardware; it therefore makes sense that these devices would exhibit similar communication patterns and packet contents. While this does point to a limitation of VIA's ability to distinguish between distinct devices, we argue that this result does not undermine the value of VIA. Because our objective is to verify the authenticity of interactions between apps and devices, the results presented in Table 4.5 and Table 4.8 are much more representative of the value of VIA in practice; i.e., confusion between profiles for an app and two devices that appear to function and communicate in nearly identical ways does not pose an immediately obvious threat to a WPAN.

Heart-Rate Monitors

The Polar heart-rate monitors in our testbed presented another opportunity to investigate VIA's ability to distinguish among distinct instances of the same device. Further experiments with these devices, however, produced no new insights from what was discussed above. For this reason, we omit the table that summarizes the precision, recall, and F1-score, as well as the corresponding confusion matrix.

Environment Sensor Instance	Precision	Recall	F1-score
Environment Sensor - Inkbird (1)	0.43	0.43	0.43
Environment Sensor - Inkbird (2)	0.56	0.56	0.56

Table 4.11: The corresponding precision, recall, and F1-score for the confusion matrix shown in Figure 4.18.

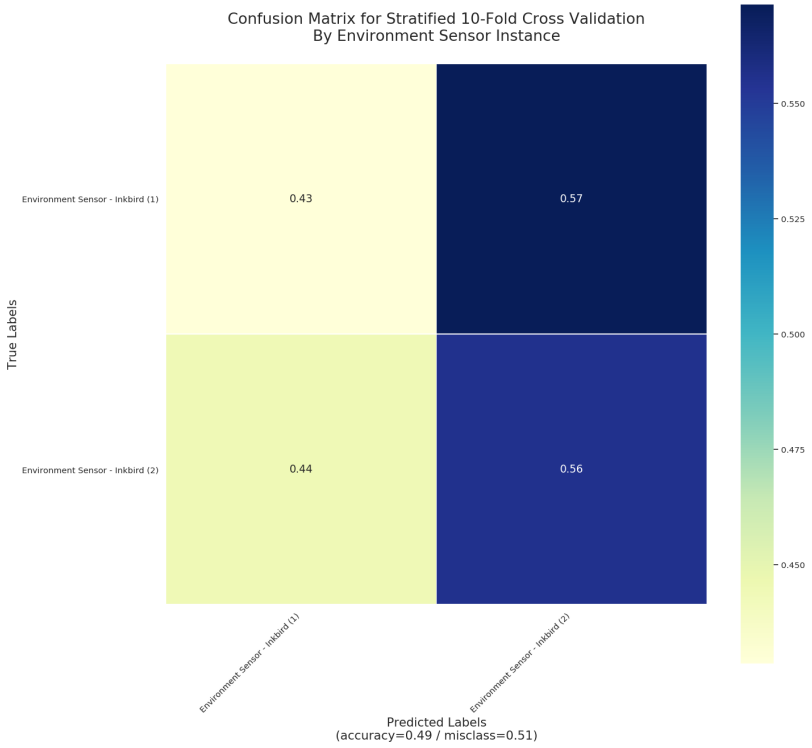

Figure 4.18: Confusion matrix for classification of environment sensors. Our smart-device testbed contains two identical environment sensors (same make and model) from the same manufacturer.

4.5.6 Future Exploration

In summary, we believe these results demonstrate that our approach to modeling authentic app-device interactions can provide a reliable means for verifying the authenticity of future interactions between devices once an initial (authentic) model is learned/obtained. There are a myriad of ways that we can further explore the dataset we presented in this section. Similarly, there is still much that we can do to evaluate VIA and its viability as a mechanism to ensure more trustworthy interactions between devices in WPANs over time. This work is a starting point for future research, and would benefit from continued exploration to provide more insight into the extent to which VIA can improve the trustworthiness of interactions within WPANs.

4.6 Discussion

In this section we add interesting insights from this work, discuss some of the challenges and opportunities surrounding real-world deployments of VIA, as well as limitations of our present work.

4.6.1 Observations of Channel Security "In The Wild"

By collecting traces from more than 20 Bluetooth devices we were able to make interesting observations about the type of security used in devices that were manufactured relatively recently (all devices in our testbed were manufactured in 2016 or later).

Using Encryption in Higher-Level Protocols for Data Security

Wang et al. observed that uniform byte-frequency distributions are observed in n-grams computed from packet payloads where the underlying channel is en-

crypted [188]. Through manual observation of the many HCI traces we collected, we were able to identify interesting patterns consistent with the use of encrypted channels (or lack thereof). Namely, from our analysis we found that only two devices (the smartlocks) in our entire testbed use encryption for communications above the link layer. This result is especially surprising considering the fact that the majority of the devices in our testbed communicate personal health data, which can be stolen or tampered with within the Host while in transit between the apps and devices. Indeed, this finding reinforces the importance of the work presented in Chapter 2.

While our observations were the result of manual inspection of HCI traces, in the future we can employ common entropy tests[22] to analyze whether Bluetooth-based WPAN devices make use of encryption at higher protocol layers. In the same vein of our work (making WPANs more trustworthy), it may be useful (e.g., enforcing security policies) to know whether connected devices are using encryption at higher protocol layers.

Using Encryption in Lower-Level Protocols for Data Security

Since we have access to all traffic across the HCI, we can observe (from software) which devices and connections make use of over-the-air security features. This is done by observing the HCI command opcodes that are used to negotiate and establish channel security at the lower layers of the Bluetooth protocol. As stated above, this information may prove useful for evaluating or improving the trustworthiness of WPANs.

[22] High entropy is indicative of less information in data; less information in data is generally achieved through explicit obfuscation of the data, for example using encryption.

4.6.2 Challenges & Opportunities for Real-World Deployment

This section discusses some of the challenges and opportunities surrounding a real-world deployment of VIA. This section also helps to shed light on how we believe our current work should be extended to meet our larger vision of using VIA as an approach to improve the integrity and security of WPANs.

Bootstrapping: Obtaining Initial Verification Models

In practice, a deployment of VIA must address how initial verification models are obtained. We envision two likely approaches (recall Figure 4.4): One approach is to learn verification models on-the-fly. For example, when a new, previously-unseen peripheral device connects with a hub running VIA, some number of initial interactions can be used to learn the models for authentic interactions. After this initial phase, the learned models can be used for performing verification of subsequent interactions.

Another approach is to obtain verification models from a trusted source, such as a device manufacturer or software/firmware vendor. These models could be fetched based on device identifiers that are exchanged when two devices initiate a connection establishment procedure. Similar to on-the-fly learning, after the appropriate model(s) have been retrieved, they can be used for performing verification of subsequent interactions. We note that this approach is gaining traction today: for example, Manufacturer Usage Descriptions (MUDs) [115] is a proposed IETF standard for formally specifying the expected network behavior of an IoT device. MUD exploits a similar observation that we do: IoT devices (generally) perform a limited set of functions, and therefore have a recognizable communication pattern that can be represented by a model (referred to as a MUD profile in the context of MUD).

Both approaches have trade-offs. Models that are learned on-the-fly may only capture a small subset of all the possible normal interactions, which may lead to false positives later on; this approach, however, ensures that VIA can always create and have access to verification models when needed. Models that are fetched from a trusted source may be better models for subsequent verification since domain experts are able to build the models; this approach, however, requires participation from the original device and software vendors (i.e., they must be willing to exercise all of the normal interactions to create models, and provide an infrastructure to share the models), as well as high-availability access to the trusted source(s) (since new devices could be encountered at any time). A robust deployment of VIA will likely support both approaches, relying on more robust models fetched from trusted sources when they are available, and learning models on-the-fly as needed.

Model Updates

Software and firmware on the various devices within WPANs will be updated at some point, which may change the underlying characteristics for what authentic interactions look like. Given this reality, it will be necessary to have a mechanism to update verification models over time. In Bluetooth-based WPANs, we envision a strategy for model adaptation based on the "Services Changed" feature defined in the Bluetooth specification [35] (Vol. 3, Part F, Section 3.2.6). The Services Changed feature exists so that one device (a BLE attribute server) can indicate to its connected devices (BLE attribute clients) that services have changed (i.e., added, removed, or modified).[23] Furthermore, the Services Changed procedure requires that clients acknowledge that they received the server's indication. In the context of our work,

[23]The real reason this feature exists seems to be for performance. Attribute handles are expected to be stable for each device across subsequent connections, which allows clients to cache information, using less packets (and less energy) to retrieve attribute values after a first discovery. The Services Changed feature is a way for the server to inform a client that server attributes have changed (e.g., after a firmware upgrade), so that a client can repeat the service discovery procedure.

this explicit service-update feature can be used to trigger VIA to fetch an updated model from a trusted source, or re-train models on the fly (e.g., using the next five interactions to learn updated verification models).

Response Strategies

Our work thus far focuses on techniques to verify authentic interactions, but an open challenge is how to best respond when verification fails. Current strategies that we are considering are to: "flag" the device and increase the verification requirements to make them more strict going forward; discard data that is not from a verified source; alert the user (e.g., mobile notification, SMS, email, auto-generated report to technical auditor); disconnect the app/device connection; and temporarily disable the network interface altogether.

One concern with the practical deployment of VIA is that false positives (i.e., failed verification that is not due to malicious activity) could deny a user access to a device or app that is needed at a critical time. To mitigate such usability issues, VIA's responsive actions could initially be more lenient and become more aggressive if there is continued deviation from a verification model over time. For instance, a device may first be flagged and inspected in more detail; if verification continues to fail, the system may then generate an alert, then disconnect an app/device, then temporarily disable the relevant network interface to prevent further exploitation possibility, and so forth.

Diagnosing Issues

Today, detection schemes, such as those deployed in IDSs, are generally only used in settings where dedicated technical staff can examine network logs and diagnose issues in their networks. The staff can, for example, identify the affected devices and patch them or remove them from the network. This approach does not translate

to WPANs where non-technical people are responsible for managing their own devices and networks and diagnosing issues.

One possible solution is to depend on device manufacturers, app developers, or other trusted third-party diagnostics services to provide assistance. This approach could be similar to sharing crash reports with software vendors – a common practice today – allowing an end-user to actively confirm that they want to share information produced by VIA's analysis when some issue is detected. This approach has the added benefit that it incorporates the expertise and domain knowledge of those most familiar with authentic behavior of devices and applications to aid in diagnosis and improving VIA's models.

Complimentary Approaches

There are other ways to provide guarantees over the integrity and trustworthiness of connections between devices. One approach is to use an encryption protocol between two parties to protect the confidentiality and integrity of a channel between them. This approach, however, will fail if the secret key is ever divulged (e.g., if one device is compromised and a key is stolen). Furthermore, this approach requires that all devices support the same encryption protocol, which may require implementation changes to many peripheral devices.

Another approach is to use PHY-layer sensing and techniques for device fingerprinting. This approach is similar to our approach in that its primary aim is to learn specific characteristics (e.g., clock skews) of nearby devices [16, 193]. A previously-learned fingerprint can be used to verify that devices encountered in the future are or are not devices with a known fingerprint. One issue with this approach is that it requires access to PHY-layer sensing, which is not ubiquitously available. Indeed, to support PHY-layer sensing, many devices would need new hardware or firmware that provides access to PHY-layer information. Another issue with

PHY-layer sensing is that it cannot necessarily detect if the same device (and radio) is being used by malware instead of authentic software; in this case, the device hardware and its PHY-layer features have not changed, but the previously-created trust relationships between devices can still be abused.

Our approach develops models for communication between devices and apps based on information obtained at the network layer and above, and uses them to verify that future communications are consistent with the previously-learned models. A significant benefit of our approach is that it does not require invasive changes to devices: it could be implemented tomorrow in a trusted OS, or next week in an Intel controller, and most people would not even be aware that it is happening.

All of the approaches discussed in this section have strengths and weaknesses. In an ideal deployment, one or more of these approaches would be deployed together to provide multi-layer protection.

4.6.3 Limitations

The central limitation of this work is that it would benefit from (1) further data collection with the aim of capturing rare (but authentic) interactions, such as catastrophic events, (2) further evaluation with an adversarial presence in the WPAN, and (3) further exploration of the limitations of VIA models.

Rare Interactions & Events

The new smart-device dataset we present in this work captures short (approximately 3-10 minutes), typical interactions in each network trace. Because our initial data collection effort was concentrated on observing typical app-device interactions, it is possible that our current dataset misses other types of interactions. For example, network traffic that only occurs when a patient has a catastrophic health event, such

as a heart attack. In the future, we will explore ways to introduce examples of rare interactions and events into network traces.

Mimicry Attacks

One especially prevalent issue that must be addressed in intrusion detection systems is *mimicry attacks*. We chose to adopt models based on n-grams in our work to aid in the characterization of network communications between apps and devices. We made this choice largely because of past work that showed that n-grams effectively model traffic patterns in a manner that is resilient to mimicry attacks [188]. Evaluating mimicry attacks requires more examples of Bluetooth-based attacks, but identifying relevant examples is difficult. Recent, known vulnerabilities (e.g., BlueBorne [21]), do not try to mimic authentic behavior – they aim to exploit specific vulnerabilities in implementation of Bluetooth software. For instance, the Android information leak vulnerability (CVE-2017-0785) aims to exploit a buffer underflow that leaks the internal memory of an Android device. The exploit is triggered by repeatedly sending redundant requests, which ultimately results in sequential reads of internal memory. In such an attack, the resulting n-grams should yield a high frequency of a limited set of byte values for ingress traffic, and a large amount of widely variable egress traffic. In theory, this sort of behavior would *immediately* deviate from any authentic verification model we have seen in this work. Nevertheless, future work should explore opportunities to exercise known exploits and evaluate how well our models can recognize these kinds of network interactions.

Intra-Class Similarity

In our evaluation we found evidence that our verification models are effective at differentiating between devices within the same class (e.g., blood-pressure monitors). This is not, however, true in all cases. For example, in our examination of

two environment sensors made by the same manufacturer our models were not as successful. As we explain above, we suspect this outcome results from the similarity of the hardware and software in these nearly identical devices. (To see just how similar their profiles appear, see the *n*-grams in Appendix A.) This result points to a potential limitation of VIA: highly similar devices (e.g., same make and model) may not be distinguishable from one another. Although not a serious limitation, it does warrant further investigation into more cases of similar devices to determine the degree to which VIA can distinguish among makes and models.

4.7 Summary

In this chapter we describe an approach to verify interactions between apps and devices over time. We apply techniques commonly used in anomaly-detection and intrusion-detection systems to characterize normal, authentic interactions between apps and devices within WPANs. We observed that authentic app-device interactions in the form of Bluetooth communications are extremely consistent from one interaction to the next. We exploit this observation to learn models early on (e.g., when two devices first connect and "pair"), that can be used to verify that future communications remain consistent over time. If new interactions are found to be inconsistent with previously-learned models, subsequent action can be taken to preserve the integrity and trustworthiness of the WPAN. In summary, we make the following contributions:

- We contribute a new data set that captures more than 300 Bluetooth HCI traces for interactions between 13 different smartphone apps and 20 smart-health and smart-home devices.
- We contribute extensions to open-source Bluetooth analysis software that improve access to tools for practical exploration of the Bluetooth protocol and

Bluetooth-based apps.

- We show how past work in anomaly-detection and intrusion-detection systems can be adapted to work in Bluetooth-based WPANs and learn models that recognize authentic app-device interactions.
- We provide empirical results to show that our approach is successful in differentiating between devices of different types and devices made by manufacturers.

5

Summary and Future Directions

In this dissertation we explore issues associated with the trustworthiness of WPANs, and develop solutions to address these issues.

In Chapter 2, we present our work on BASTION-SGX [156, 184]. We introduce an architecture and methodology for achieving secure and trustworthy I/O on platforms with SGX-enabled processors. In doing so, we identify and solve several challenges in realizing Trusted I/O for Bluetooth on SGX-enabled platforms; we define and present a new Trusted I/O architecture; we present an analytical evaluation of the performance impact of Trusted I/O; and we present a prototype and a case study that demonstrates how our solution effectively protects sensitive

Bluetooth I/O data from privileged malware. This work demonstrates that it is possible to establish more trustworthy channels for app-device communications in WPANs.

In Chapter 3, we present our work on Amulet [91, 139]. We present a secure, low-power platform and suite of tools for developing and deploying mobile-health applications. My specific contributions to this project include substantial contributions to the design and implementation of Amulet's software stack and runtime system, the design and implementation of Amulet's firmware-production toolchain that guarantees application isolation, the design and implementation of resource models that are deployed in graphical developer tools that aid developers in developing secure and efficient applications, and an experimental evaluation of Amulet. This work demonstrates how to design more trustworthy peripheral devices for WPANs. The entire Amulet platform has been publicly released and is freely available.

In Chapter 4, we present our work on VIA. We introduce a novel approach for ongoing verification of authentic interactions between apps and devices in WPANs. We assembled a testbed made up of two distinct device categories (smart-health and smart-home devices) consisting of 9 different device types, and 20 devices in total. From this testbed, we produced a novel dataset of more than 300 Bluetooth HCI network traces. We contribute extensions to open-source Bluetooth analysis software for practical exploration of the Bluetooth protocol and Bluetooth-based apps. We present a novel modeling technique (*hierarchical segmentation*), coupled with n-gram models to reliably characterize and verify app-device interactions. We present an experimental evaluation of our work using the 20 off-the-shelf devices from our testbed. This work demonstrates that it is possible to verify trustworthy behavior within WPANs.

The evaluation of our solutions shows promise for improving the trustworthi-

ness of WPANs. There are, however, remaining questions and other areas of future investigation. We discuss some of these areas next.

App-Device Binding

In future work I will explore a binding mechanism that binds applications (on a mobile device) with devices (or, if relevant, applications on devices) in a meaningful way. Pierson et al. show how devices in close proximity can easily share a secret [158]. Often, this exchange is needed to share a secret, such as a cryptographic key, that is used to encrypt information between devices (e.g., over a wireless radio channel). Any entity with the key – in this case, each device – has the ability to decrypt the information. Is this really what a user intends when two devices are connected to share information? For example, suppose a user wishes to securely connect a medical device with their smartphone or a smart display in a doctor's office to visualize their medical data. Should the phone or display as a whole have access to any information from the medical device? The answer is surely "no." Generally, when a user connects two (or more) devices in this way, they (implicitly) intend for a specific application or system service to have access to the data from the device. The problem here is that there exists no mechanism for a user (or automated system) to connect a specific application with a specific external device.

When devices "pair," they form a "bond" at the device level (i.e., smartphone to medical device). This problem has previously been studied within the context of Android devices [144] and is referred to as the *threat of external device mis-bonding*. In their work, the authors propose Dabinder: a solution to automatically generate security policies ("app-device relations") that bind an external device to its authorized app, and enforces the policies without otherwise impeding on the normal operation. Their approach, however, automatically generates policies by monitoring which application first tries to create a connection with a newly paired device; the applica-

tion that first tries this connection becomes the "authorized app" (which they hope is also the "official app"). While automatic policy generation in this way makes sense on some level (i.e., no human input is needed), it is vulnerable to malware that also looks for newly paired devices and attempts to create a connection with a device first, enabling the malware to become the "official app" that is allowed to communicate with the external device.

An appropriate app-device binding should extend the exchange of secret information to be between the intended ("official") application and external device, while also enabling a human to confirm or "bless" a security policy. The models we presented in Chapter 4 may be applicable here. For example, a binding between an app and device could be allowed so long as the interactions are consistent with a pre-defined VIA model.

Our recent work shows how binding/relationships between apps-devices can be continuously (re-)verified, which presents an interesting alternative to the DaBinder solution, which is static. If, however, an app or device is compromised at a later time, it can abuse its access to the device/app to carry out nefarious activity.

Exploring Additional Anomaly Detection Techniques

In past work, many approaches to anomaly detection have been pursued, including information theoretic approaches, neural networks, support vector machines, genetic algorithms, and many more [178]. We plan to explore other algorithms and features that may lead to improvements in our models.

Anomalies in WPANs

In the past there have been concerns around the effectiveness of machine learning in network intrusion detection [178]. Much of this past work – which focuses on the detection of intrusions in IP-based network traffic – faces extreme challenges due to

characteristics of the networks in which they were deployed. For example, machine learning models have to overcome enormous variability in benign network traffic. Also, because of the size, scale, and nature of these networks, it is challenging to know how a network operator should interpret something that is flagged as anomalous; *is an anomaly indicative of a cybersecurity threat?* Or in the context of smart-health devices and health monitoring, *is an anomaly indicative of a critical health event?* Or *is an anomaly an indication that the anomaly-detection model simply failed to model this type of network activity?* The criticisms thus far seem to ultimately be rooted in the fact that anomaly *detection* is not coupled with tools to help with *interpretation*. In other words, anomaly detection needs to be supported by a description of the anomaly. While this is not an easy task, past work has proposed tools that can help, such as the Multivariate Exploratory Data Analysis (MEDA) Toolbox [49]; these tools could, for example, be used to help network operators understand what event(s) and what feature(s) led to their anomaly-detection system identifying something as anomalous. In future work, we will explore the use of the MEDA Toolbox and other related tools to better understand the anomalies that our work detects, and to better understand the limitations of our approach.

Extending & Applying VIA

Our larger vision for VIA is that it would be deployed within WPANs to perform active verification of network interactions over time. Our work thus far does not make instantaneous decisions about the trustworthiness of a connection. Rather, it assembles a sequence of interactions (packets) and measures the consistency of those interactions with a previously-learned model. But how can this be further quantified?

One idea is to determine the trustworthiness of active connections, based on computed scores from observations in network traffic and compare this against pre-

computed normative models. Different types of scores can be used to evaluate new network traffic across different dimensions. For instance, for each observation one could compute an *instantaneous confidence score*. Each instantaneous score can then applied to a *global and temporal confidence score* that records changes in the system's overall confidence (global) over time (temporal). Ultimately, this confidence is used to make decision about whether a previously established connection remains trustworthy.

Coarse-grained Modeling & Analysis

One present limitation of VIA is that it primarily makes use of fine-grained features obtained from network traffic. Past work (e.g., [112, 48]) has shown that anomaly-detection models computed from normal network statistics based on network flows can be quite effective in detecting anomalies (e.g., an abnormally high volume of packets – *volume anomalies* – from a device observed in a short period of time), which may indicate anomalous activity that warrants further investigation (e.g., a network attack). The successes of past work lead us to believe that incorporating more coarse-grained network features may improve our current approach.

To model normal network traffic based on network flows, many approaches have been considered. Some are based on Principle Component Analysis (PCA), including PCA-based Multivariate Statistical Process Control (PCA-MSPC) [112] and, more recently, PCA-based Multivariate Statistical Network Monitoring (PCA-MSNM) [48]. One of the major benefits of PCA is its unsupervised nature (i.e., it requires no *a priori* specification of potential anomalies).

At a high level, PCA-based anomaly detectors (specifically those based on Multivariate Statistical Process Control Theory) work by dividing network data (calibration data) into two sets (subspaces): a structured subspace (*model*) and a noisy subspace (*residual*). Anomaly detection is performed in both subspaces

using different statistical tests to detect anomalies.[1] The statistics (Q-statistic and D-statistic) are computed using the calibration data to set thresholds (control limits) that can be used to detect anomalies in their respective subspaces. As incoming data is processed by the detector, it is projected into one of the aforementioned subspaces, and the control limits are used to identify anomalies, such as limits being significantly or consistently exceeded [48].

Generally speaking, PCA is applied to a two-dimensional dataset consisting of a number of observations (rows) of a number of features (columns). Like others [112, 47, 46], we can adopt the use of simple counters as quantitative features ("feature-as-a-counter"). Basically, each observed feature value is defined as the number of times a given event (e.g., the number of incoming/outgoing packets and bytes; the number of requests to a port or group of ports) takes place during a time window w. While this definition is rather general, it has been shown to be capable of representing most of the types of information of interest in monitoring network traffic for security [47]. See Figure 5.1 for an example of features used in recent work demonstrating the effectiveness of a PCA-MSNM system [46].

Modeling and analyzing network traffic in a coarse-grained way stands to provide additional, useful information to how our current models are learned. Furthermore, we note that it is not possible to analyze the packet contents inside individual datagrams when network traffic uses secure connections (e.g., HTTPS); in these cases, fine-grained features extracted from deep packet inspection is impossible, but flow-level analysis can still provide meaningful insights into network traffic.

We have already begun to investigate different types of features that can be

[1] A criticism of some past work is that they divide data into two subspaces: one representing *normal* behavior and one representing *anomalous* behavior. Anomaly detection is performed only in the latter. This is not consistent with Multivariate Statistical Process Control Theory – which recent work [48] points out – and is the cause of some problems of past work.

Variable	#features → values
Source IP	2 → *public, private*
Destination IP	2 → *public, private*
Source port	50 → *specific services, Other*
Destination port	50 → *specific services, Other*
Protocol	5 → *TCP, UDP, ICMP, IGMP, Other*
Flags	6 → *A, S, F, R, P, U*
ToS	3 → *0, 192, Other*
# Packets in	5 → *very low, low, medium, high, very high*
# Packets out	5 → *very low, low, medium, high, very high*
# Bytes in	5 → *very low, low, medium, high, very high*
# Bytes out	5 → *very low, low, medium, high, very high*

Figure 5.1: Variable values considered as features in a recent PCA-based anomaly detector [46].

used to detect anomalies in network traffic. Some of the previous features that we thought would be useful were not as useful as we originally believed they would be. For example, inquiry and connection establishment is considerably different for Bluetooth classic devices when compared to BLE devices. It turns out that some of the features (e.g., COD, PSM) are exchanged infrequently or not at all in BLE network communications. In future work we will continue to explore opportunities to include coarse-grained features into our current approach to model learning.

A

N-Grams From Smart-Device Testbed

This appendix contains examples of *n*-grams (in our current work, 1-grams) that are calculated from devices in our smart-device testbed. In each plot, we overlay the *n*-grams computed from subsequent traces. Plotting *n*-grams in this way shows how subsequent (authentic) interactions are very similar.

The *n*-grams shown below are computed over the payload portion of L2CAP packets. We found that *n*-grams computed over L2CAP payloads provide a high-quality source of information for learning our verification models and performing subsequent verification. We also note that the L2CAP protocol is the layer at which apps and devices exchange data in Bluetooth; because our work is interested in modeling app-device interactions, this protocol layer is a logical choice for modeling.

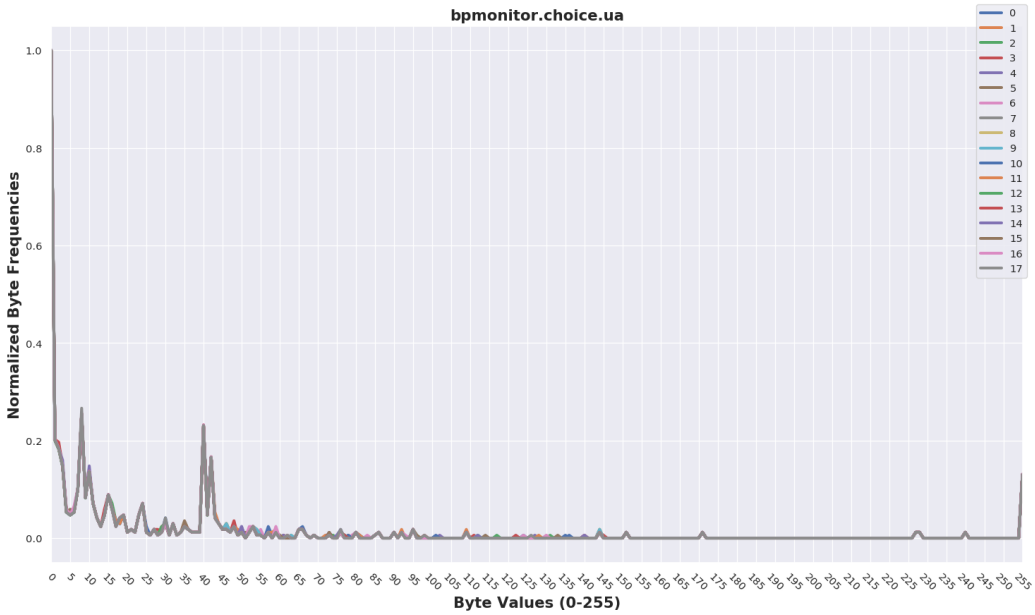

Figure A.1: Choice Blood Pressure Monitor - Upper Arm

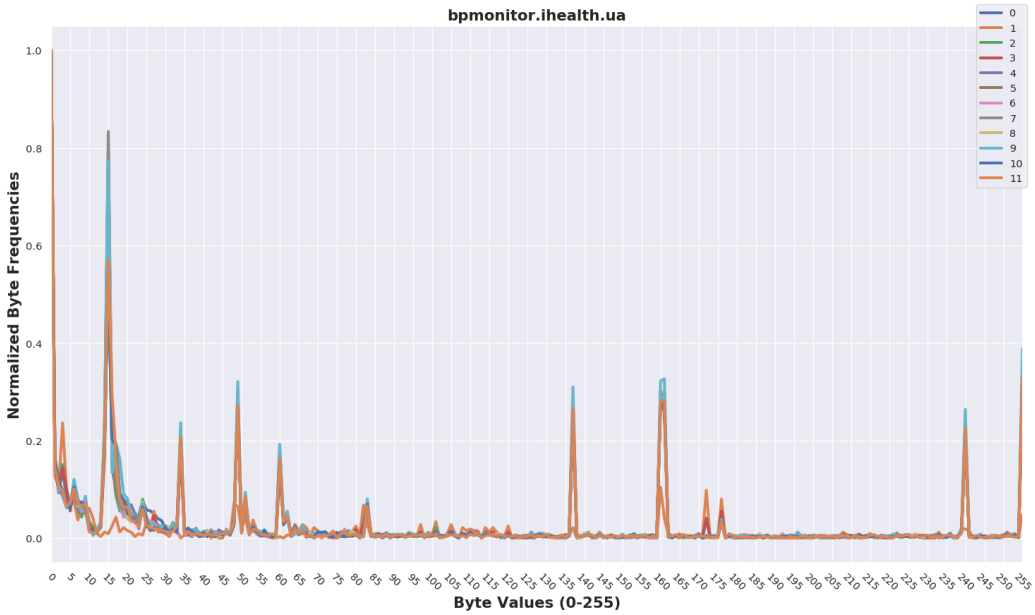

Figure A.2: iHealth Blood Pressure Monitor - Upper Arm

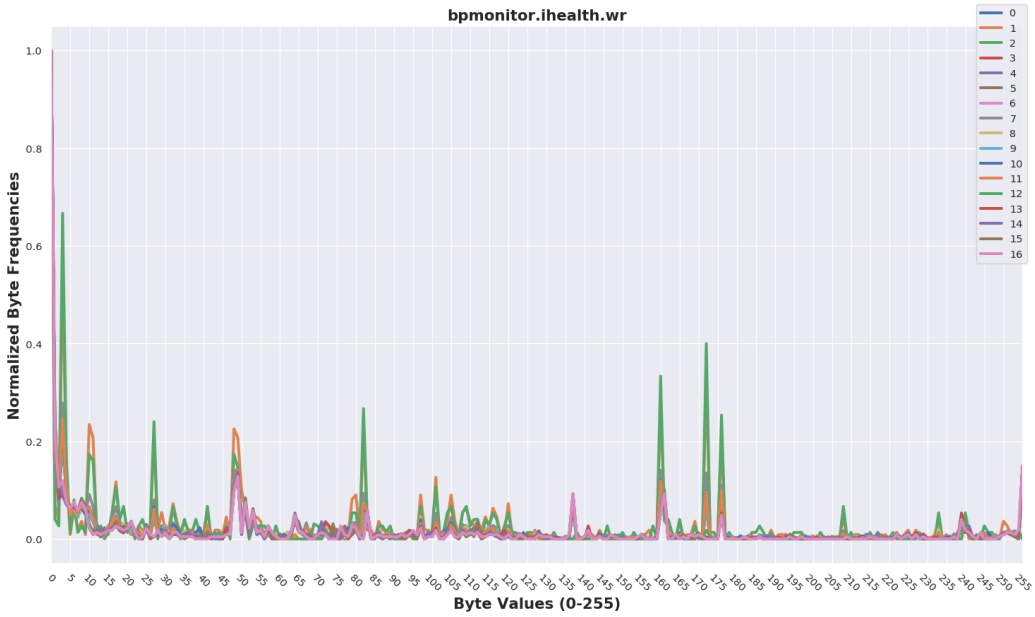

Figure A.3: iHealth Blood Pressure Monitor - Wrist

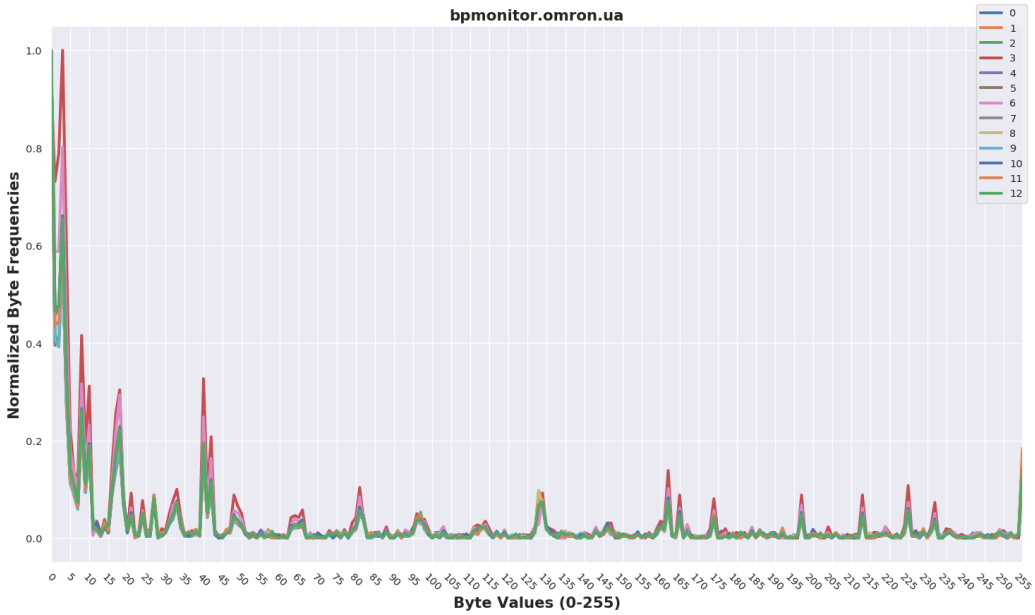

Figure A.4: Omron Blood Pressure Monitor - Upper Arm

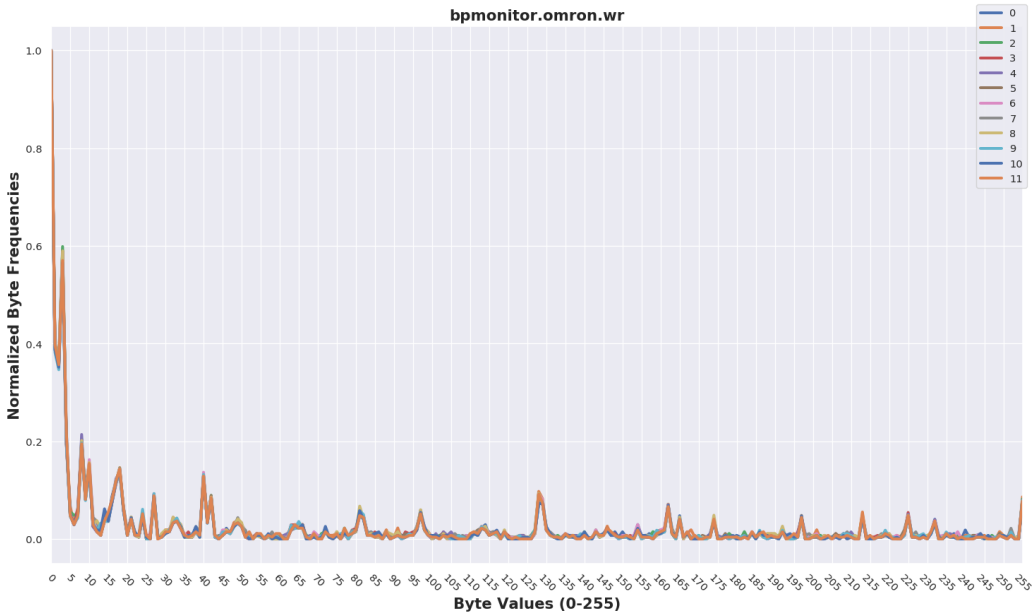

Figure A.5: Omron Blood Pressure Monitor - Wrist

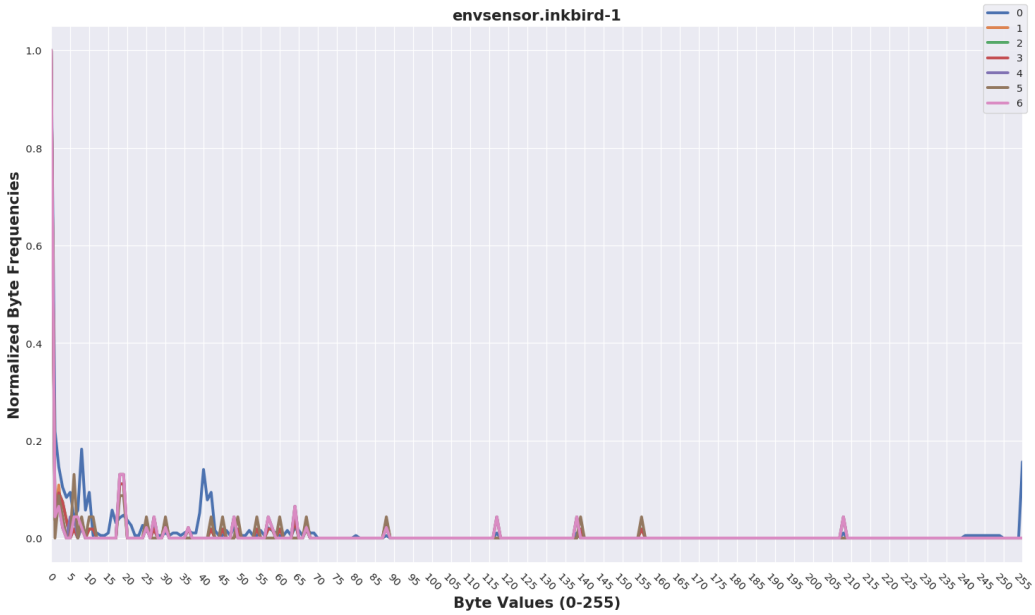

Figure A.6: Inkbird Environment Sensor (1)

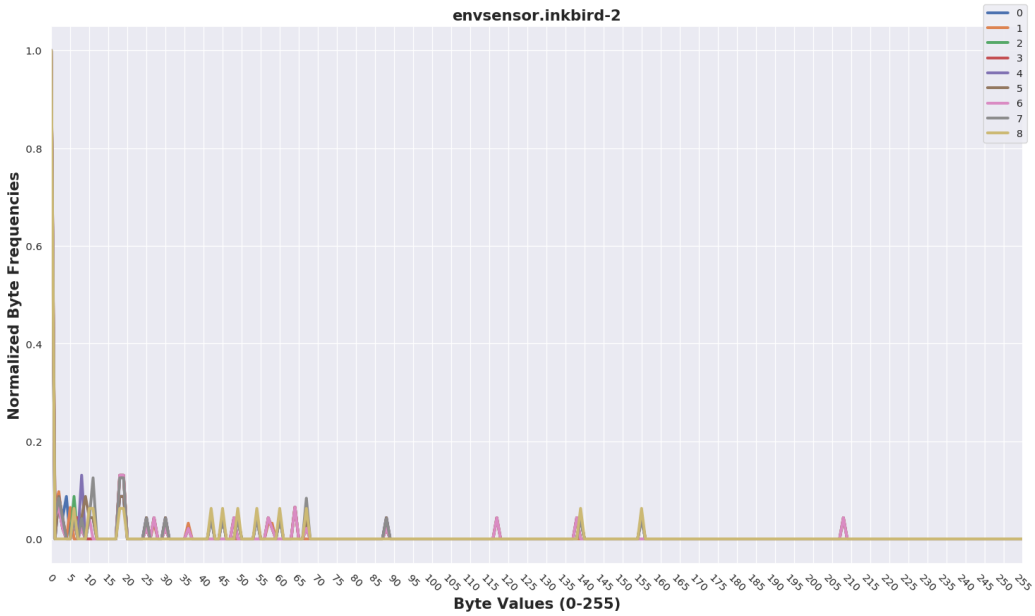

Figure A.7: Inkbird Environment Sensor (2)

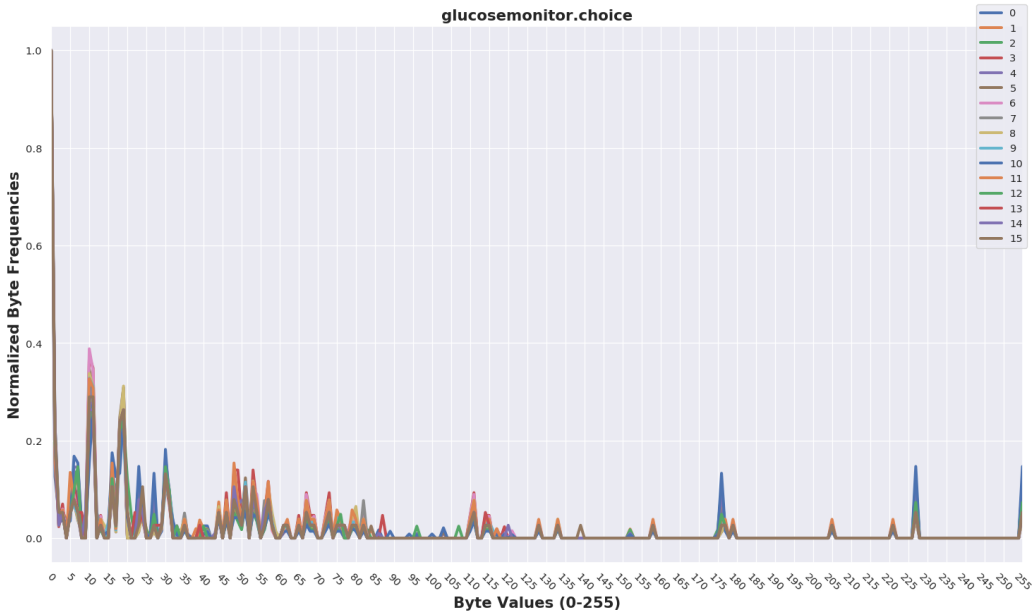

Figure A.8: Choice Glucose Monitor

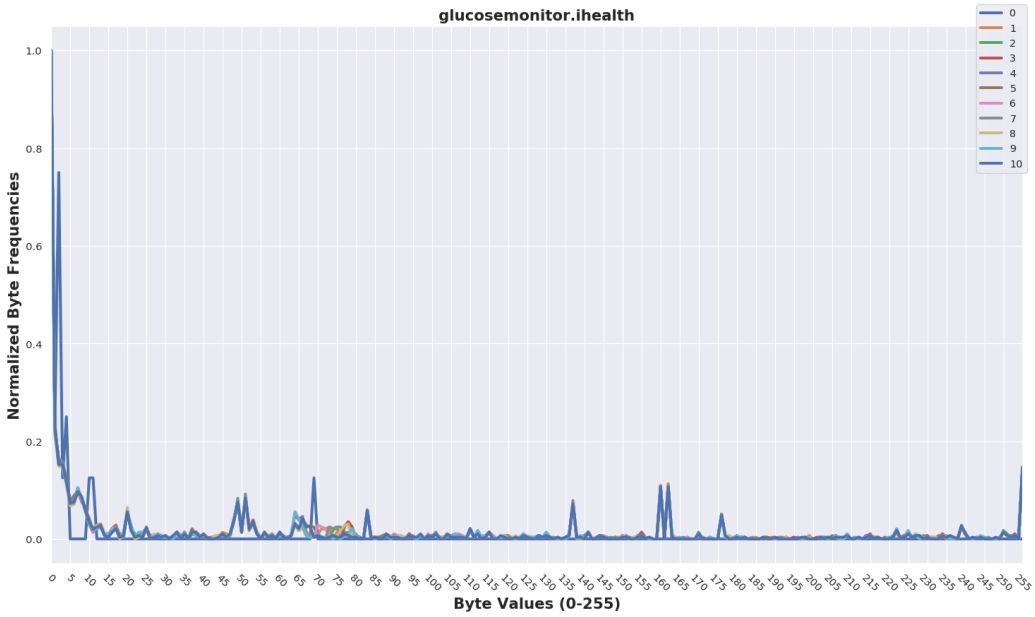

Figure A.9: iHealth Glucose Monitor

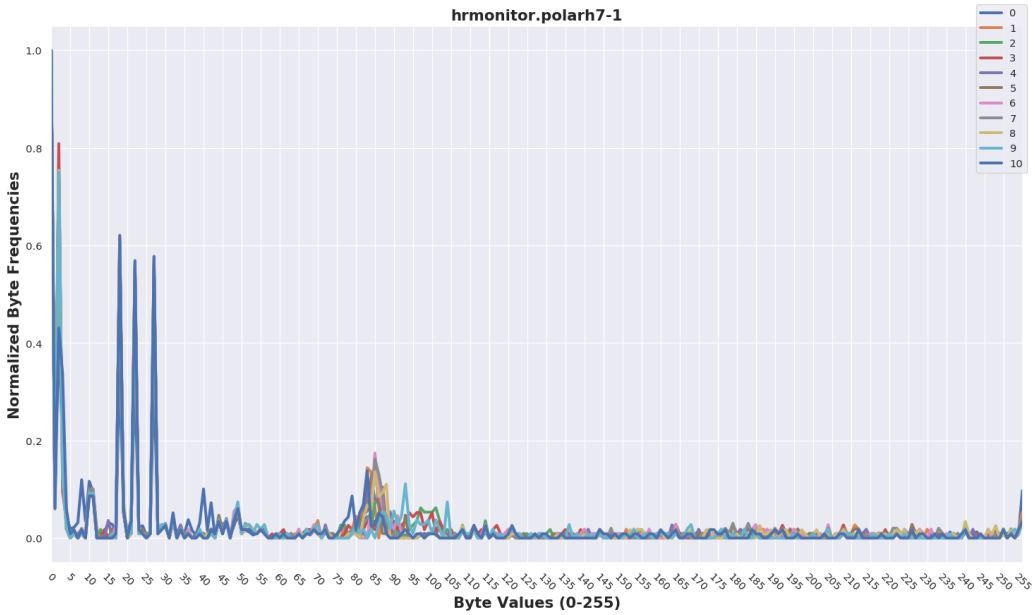

Figure A.10: Polar H7 Heart Rate Monitor (1)

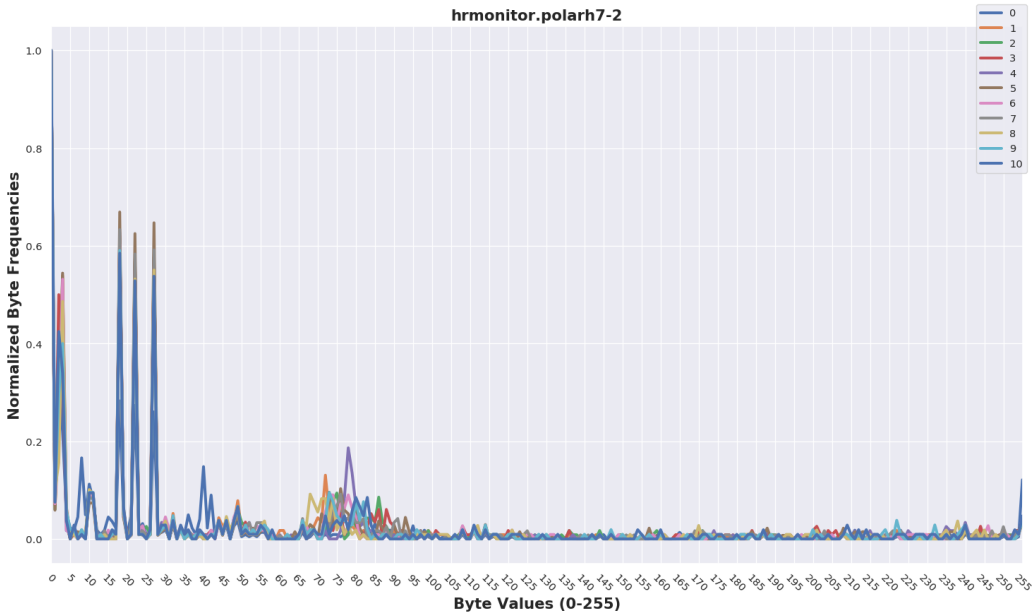

Figure A.11: Polar H7 Heart Rate Monitor (2)

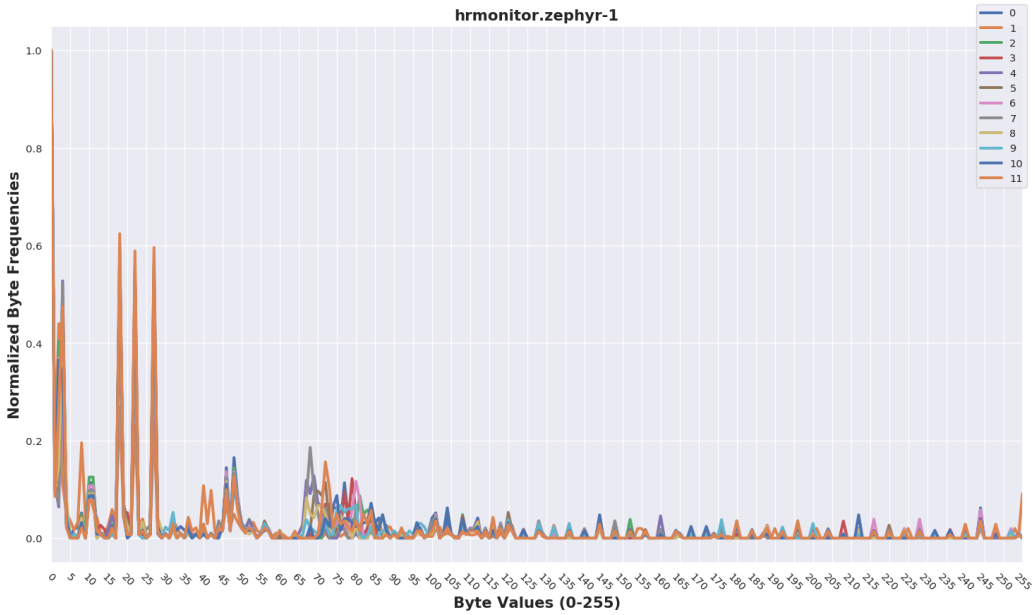

Figure A.12: Zephyr Heart Rate Monitor

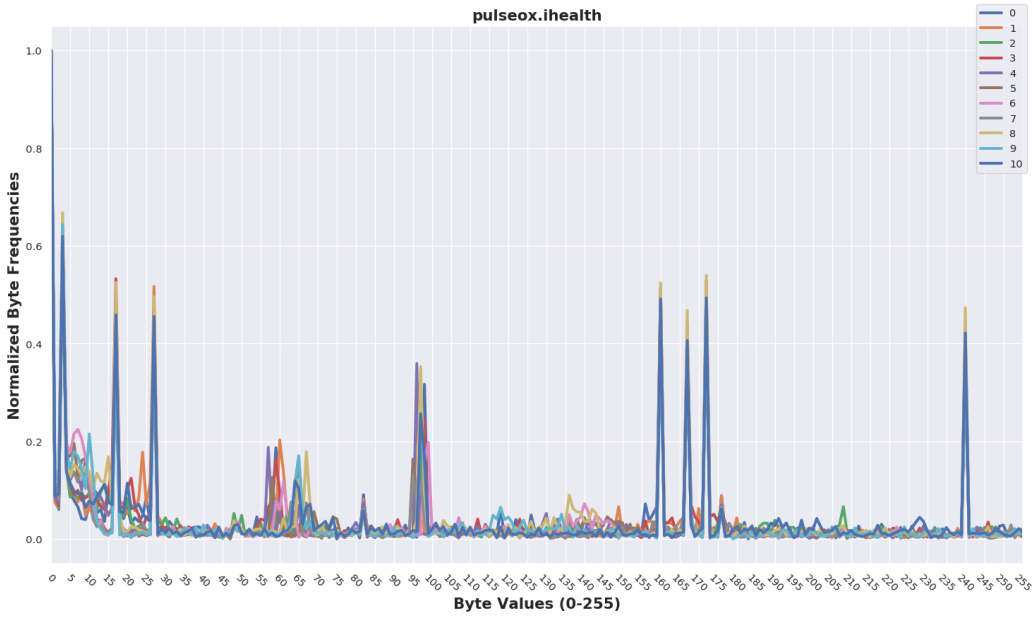

Figure A.13: iHealth Pulse Oximeter

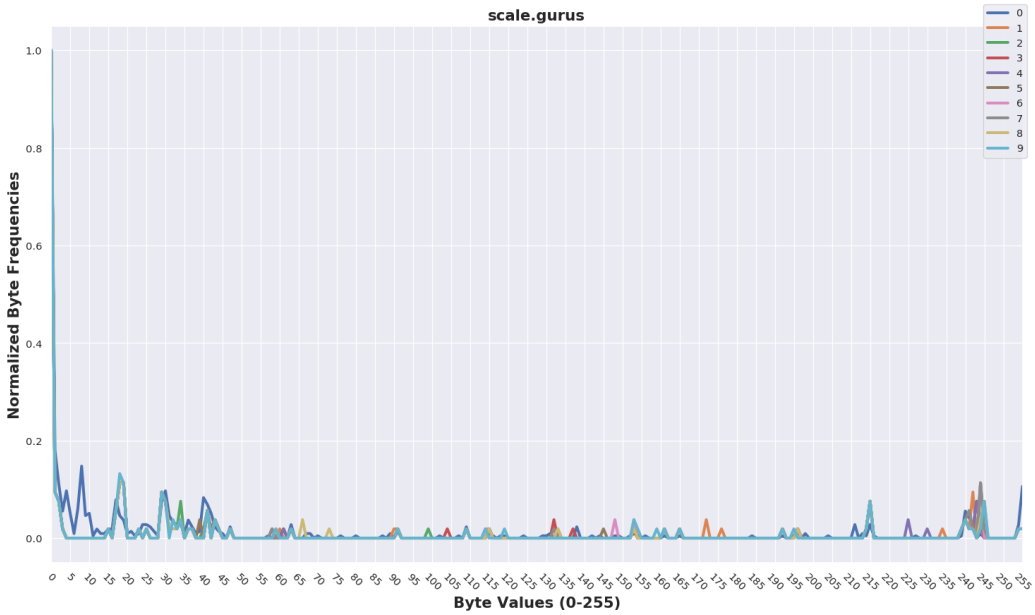

Figure A.14: Gurus Scale

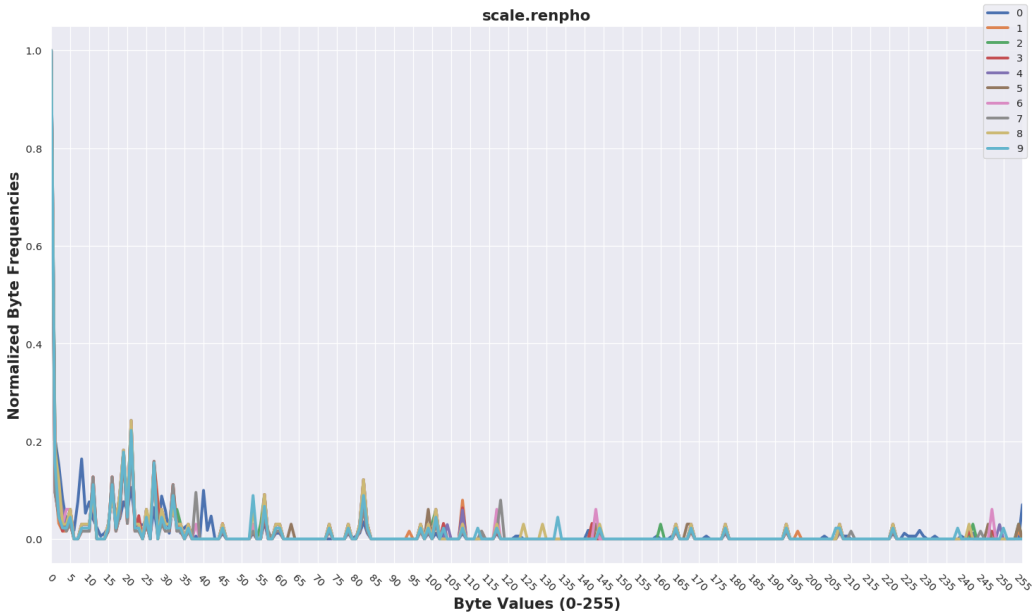

Figure A.15: RENPHO Scale

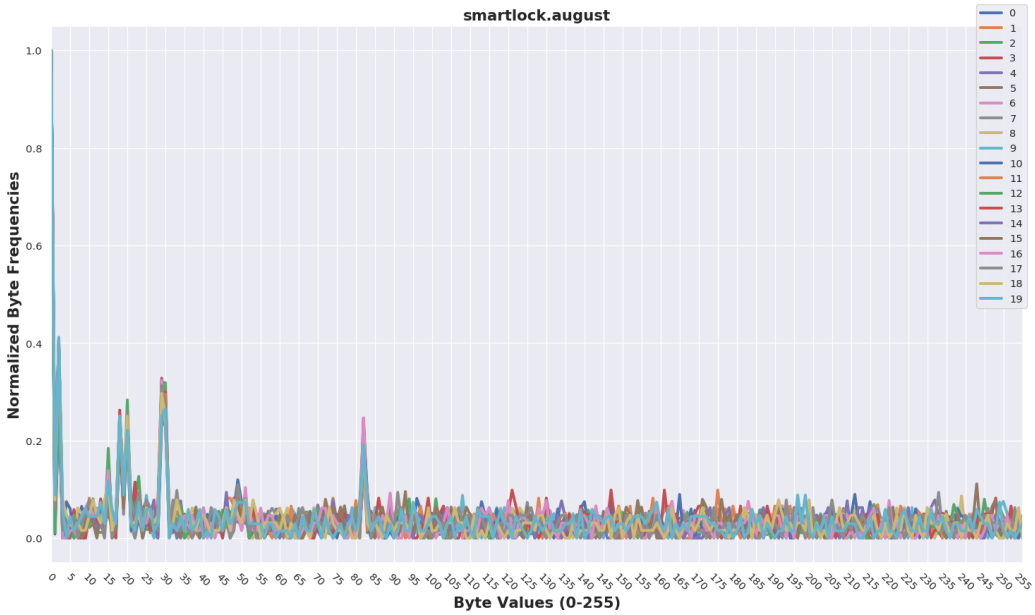

Figure A.16: August Smart Lock

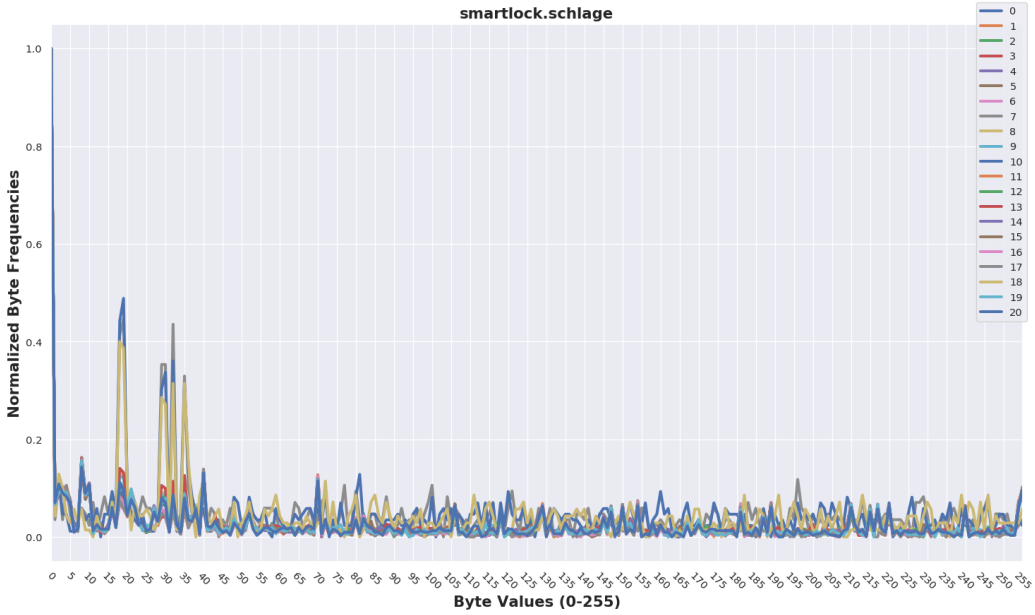

Figure A.17: Schlage Smart Lock

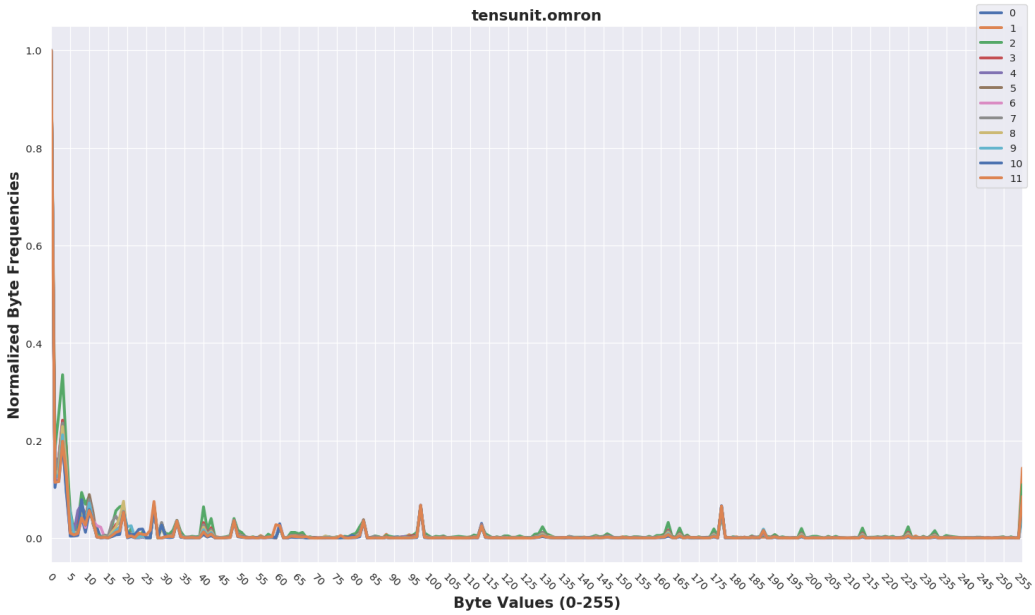

Figure A.18: Omron TENS Unit

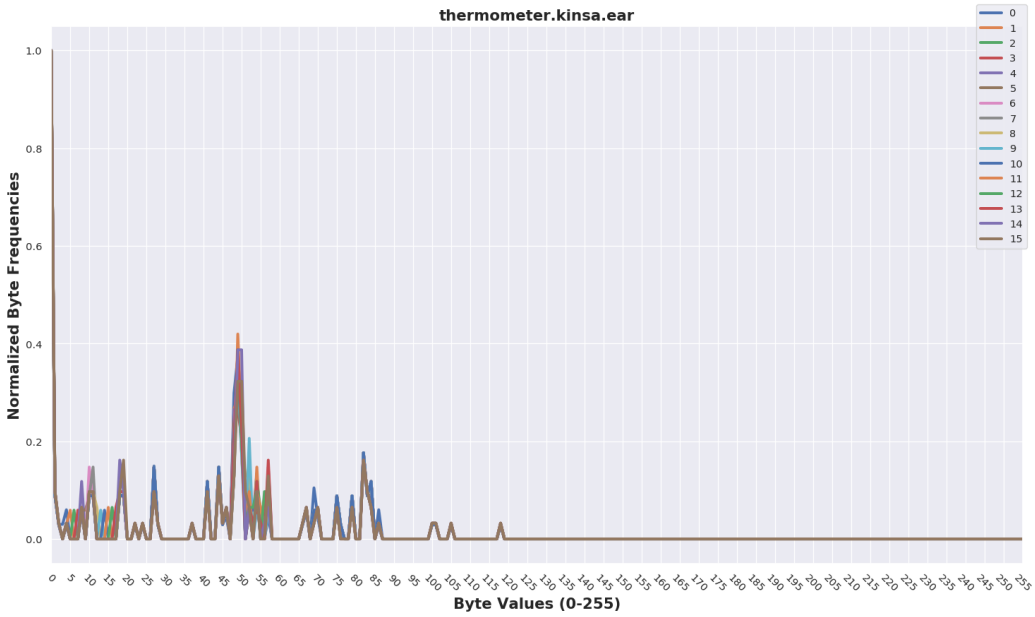

Figure A.19: Kinsa Thermometer - Ear

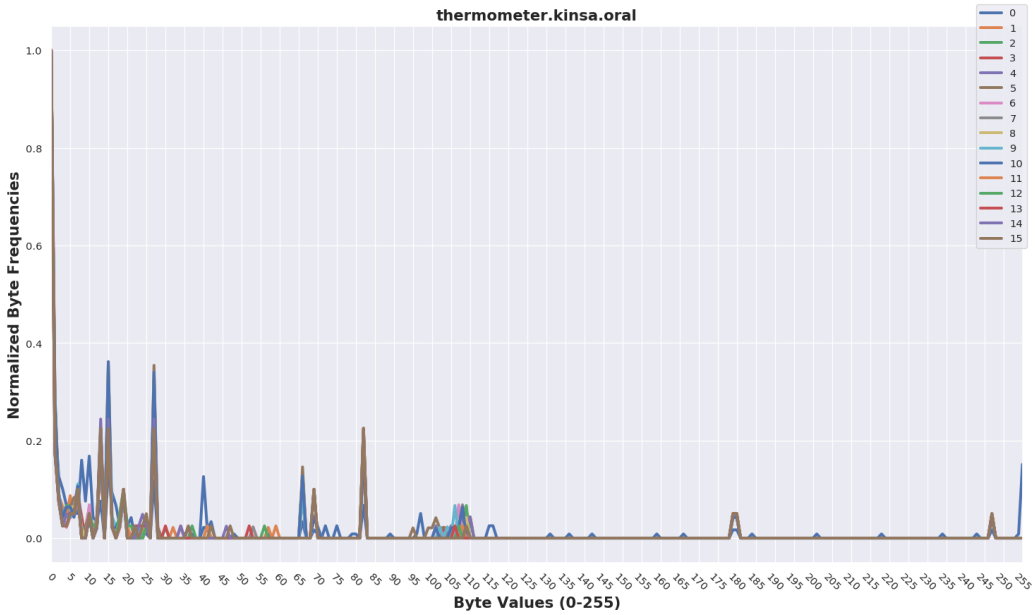

Figure A.20: Kinsa Thermometer - Oral

When I use a word, it means just what I choose it to mean.

—Humpty Dumpty, in Alice in Wonderland's Through the Looking Glass

Glossary

advertiser: A BLE device sending advertising packets.

ATT/GATT client: A Bluetooth device that sends requests to a server and receives responses (and server-initiated updates) from it.

ATT/GATT server: A Bluetooth device that receives requests from a client and sends responses back. It also sends server-initiated updates when configured to do so, and it is the role responsible for storing and making the user data available to the client, organized in attributes. Every BLE device sold must include at least a basic GATT server that can respond to client requests, even if only to return an error response.

authentic: "To be of undisputed origin; genuine; made or done in the original way, or in a way that faithfully resembles an original; based on facts; accurate or reliable" [118].

bandwidth: The amount of data that can be transferred from one point to another per unit time; typically measured in bits per second.

BLE central: A BLE device that "repeatedly scans the preset frequencies for connectable advertising packets and, when suitable, initiates a connection" [62].

BLE peripheral: A BLE device that "sends connectable advertising packets periodically and accepts incoming connections" [62].

Bluetooth: Bluetooth (Classic) is a wireless technology for exchanging data between devices over short distances using short-wavelength UHF radio waves in the ISM radio bands (2.400 to 2.485 GHz). Bluetooth Classic and BLE are ideal technologies for building wireless personal area networks (WPANs).

Bluetooth controller: In Bluetooth, A logical entity made up of all of the layers below the HCI.

Bluetooth Device Address: A unique 48-bit hardware device address used to identify a Bluetooth device, similar to an Ethernet MAC address for a computer.

Bluetooth host: "A logical entity made up of all the layers between Bluetooth's core profiles (i.e., Bluetooth applications and services) and the HCI" [35].

Bluetooth Low Energy: Bluetooth Low Energy (BLE) is a wireless personal area network (WPAN) technology intended to provide simpler Bluetooth connectivity with considerably reduced power consumption and cost.

Bluetooth master: A Bluetooth device that initiates a connection and manages it later.

broacaster: A BLE device that "sends nonconnectable advertising packets periodically to anyone willing to receive them" [62].

Class of Device: In Bluetooth, A 3-byte value that consists of major and minor class numbers that are intended to define a general family of devices; the Bluetooth specification [35] intends for this attribute to be used to indicate the capabilities of a device.

connection interval: In Bluetooth, the period in which each side communicates with the other at least once.

egress: Refers to a packet that flows along the exit path from a hub to a connected peripheral device; also referred to as `h2d` (host-to-device).

enclave: CPU-enforced containers that protect selected code and data from disclosure or modification.

FRAM: Ferroelectric RAM (FRAM) is non-volatile memory technology that offers persistent storage without power, and is 100 times faster than flash memory.

ingress: Refers to a packet that flows along the entry path into a hub from a connected peripheral device; also referred to as `d2h` (device-to-host).

interaction: Any exchange of packets between a hub and peripheral device.

make: Refers to the manufacturer of the device.

Manufacturer Usage Description: An IETF specification and framework for formally specifying the expected network behavior of an IoT device.

model: Refers to an identifier, such as a name or number, that is used to distinguish between devices made by the same manufacturer.

network latency: The amount of time it takes for a packet to cross the network from a device that created the packet to the destination device.

observer: A BLE device that "repeatedly scans preset frequencies to receive any nonconnectable advertising packets currently being broadcasted" [62].

piconet: In Bluetooth, a collection of devices sharing a single radio channel (synchronized to a master device).

scanner: A BLE device scanning for advertising packets.

security policy: A specification of Bluetooth I/O metadata that is used to protect I/O channels between a trusted app and the Bluetooth Controller.

slave: A Bluetooth device that accepts a connection request and follows the master's timing.

SRAM: Static RAM (SRAM) is volatile memory technology that offers storage while powered.

trace: A packet capture consisting of all packets that are observed between the time that a hub and peripheral device establish and terminate a connection.

trusted app: An app that protects select code and data using security features of the platform.

trusted path: A secure path between a user app and a user I/O device, typically realized by a combination of trusted components working together to protect the I/O channel(s).

type: Refers to a device's functionality and purpose.

verification: "The process of establishing the truth, accuracy, or validity of something; the act of making sure or demonstrating that (something) is true, accurate, or justified" [173].

Acronyms

AEAD: authenticated encryption with associated data.

AFT: Amulet Firmware Toolchain.

AOS: Amulet-OS.

ARP: Amulet Resource Profiler.

ARP-View: Amulet Resource Profiler View.

ATT: Attribute Protocol.

BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX.

BDADDR: Bluetooth Device Address.

BER: Bit Error Rate.

BLE: Bluetooth Low Energy.

BT: Bluetooth.

CI: connection interval.

COD: Class of Device.

CRC: Cyclic Redundancy Check.

DLE: Data Length Extension.

fn: False Negative.

fp: False Positive.

GATT: Generic Attribute Protocol.

HCI: Host Controller Interface.

IDS: Intrusion Detection System.

IFS: Inter Frame Space.

IoT: Internet of Things.

L2CAP: Logical Link Control and Adaptation Protocol.

LL: Link Layer.

MIC: Message Integrity Check.

MTU: Maximum Transmission Units.

MUD: Manufacturer Usage Description.

OS: Operating System.

OTA: Over-The-Air.

PAN: Personal Area Network.

PDU: Protocol Data Unit.

PN: Personal Network.

PSM: Protocol/Service Multiplexor.

SGX: Software Guard Extensions.

TCB: Trusted Computing Base.

TEE: Trusted Execution Environment.

TIO: Trusted I/O.

tn: True Negative.

tp: True Positive.

TZ: TrustZone.

VIA: Verification of Interaction Authenticity.

VSDC: Vendor Specific Debug Command.

WPAN: Wireless Personal Area Network.

References

[1] Yuvraj Agarwal, Steve Hodges, Ranveer Chandra, James Scott, Paramvir Bahl, and Rajesh Gupta. Somniloquy: Augmenting Network Interfaces to Reduce PC Energy Usage. In *Proceedings of the USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, pages 365–380. USENIX Association, 2009.

[2] Karan Aggarwal, Chenlei Zhang, Joshua C Campbell, Abram Hindle, and Eleni Stroulia. The Power of System Call Traces: Predicting the Software Energy Consumption Impact of Changes. In *Proceedings of Annual International Conference on Computer Science and Software Engineering*, (CASCON), pages 219–233. IBM Corp., 2014. Online at http://portal.acm.org/citation.cfm?id=2735546.

[3] Mohiuddin Ahmed, Abdun Naser Mahmood, and Jiankun Hu. A survey of network anomaly detection techniques. *Journal of Network and Computer Applications*, 60:19–31, 2016. DOI 10.1016/j.jnca.2015.11.016.

[4] Said Al-Riyami, Frans Coenen, and Alexei Lisitsa. A Re-evaluation of Intrusion Detection Accuracy: Alternative Evaluation Strategy. In *Proceedings of*

the ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 2195–2197. ACM Press, 2018. DOI 10.1145/3243734.3278490.

[5] Wahhab Albazrqaoe, Jun Huang, and Guoliang Xing. Practical Bluetooth Traffic Sniffing. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services - MobiSys '16*, pages 333–345, New York, New York, USA, 2016. ACM Press. DOI 10.1145/2906388.2906403.

[6] Amazon. Amazon Echo, 2018. Online at https://www.amazon.com/echo/.

[7] AMD. AMD Secure Technology, 2018. Online at https://www.amd.com/en/technologies/security.

[8] Android. Optimizing Performance and Battery Life, December 2015. Online at http://developer.android.com/training/wearables/watch-faces/performance.html.

[9] Android. Power Profiles for Android, November 2015. Online at https://source.android.com/devices/tech/power/index.html.

[10] Angel. Angel Sensor, August 2019. Online at https://angel.co/company/angel-sensor.

[11] Fabrizio Angiulli, Luciano Argento, and Angelo Furfaro. Exploiting N-gram location for intrusion detection. *Proceedings of the IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, 2016-Janua:1093—-1098, 2015. DOI 10.1109/ICTAI.2015.155.

[12] ANTLR. ANTLR (ANother Tool for Language Recognition), March 2014. Online at http://www.antlr.org/.

[13] Apple. Apple Watch, December 2015. Online at http://www.apple.com/watch/.

[14] Apple. Energy Efficiency and the User Experience, December 2015. Online at https://developer.apple.com/library/prerelease/watchos/documentation/Performance/Conceptual/EnergyGuide-iOS/.

[15] Apple. Apple HomePod, 2018. Online at https://www.apple.com/homepod/.

[16] Chrisil Arackaparambil, Sergey Bratus, Anna Shubina, and David Kotz. On the Reliability of Wireless Fingerprinting using Clock Skews. In *Proceedings of the ACM Conference on Wireless Network Security (WiSec)*, March 2010. DOI 10.1145/1741866.1741894.

[17] Hiromi Arai, Keita Emura, and Takuya Hayashi. A Framework of Privacy Preserving Anomaly Detection. In *Proceedings of the Workshop on Privacy in the Electronic Society (WPES)*, pages 111–122. ACM Press, 2017. DOI 10.1145/3139550.3139551.

[18] Arm. Mbed OS. ARM mbed IoT Device Platform, 2014. Online at http://mbed.org/technology/os/.

[19] Arm. Arm TrustZone, 2018. Online at https://www.arm.com/products/silicon-ip-security.

[20] Arm. mbed TLS, 2018. Online at https://tls.mbed.org/.

[21] Armis. BlueBorne, 2017. Online at https://www.armis.com/blueborne/.

[22] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'keeffe, Mark L Stillwell, Christof Fetzer, David Goltzsche, David Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, Dan O'Keeffe, Mark L Stillwell, David Goltzsche, David Dave Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer.

SCONE: Secure Linux Containers with Intel SGX. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 689–703, 2016. Online at https://www.usenix.org/conference/osdi16/technical-sessions/presentation/arnautov.

[23] Charles Arthur. Your smartphone's best app? Battery life, say 89% of Britons. *The Guardian*, May 2014. Online at http://www.theguardian.com/technology/2014/may/21/your-smartphones-best-app-battery-life-say-89-of-britons.

[24] N. Asokan, Ferdinand Brasser, Ahmad Ibrahim, Ahmad-Reza Sadeghi, Matthias Schunter, Gene Tsudik, and Christian Wachsmann. SEDA: Scalable Embedded Device Attestation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 964–975. ACM Press, 2015. DOI 10.1145/2810103.2813670.

[25] Adam J Aviv, Benjamin Sapp, Matt Blaze, and Jonathan M Smith. Practicality of Accelerometer Side Channels on Smartphones. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, pages 41–50. ACM, 2012. DOI 10.1145/2420950.2420957.

[26] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, Thomas Schmidt, and Others. RIOT OS: Towards an OS for the Internet of Things (poster). In *Proceedings of the IEEE International Conference on Computer Communications (INFOCOM)*, 2013. DOI 10.1109/INFCOMW.2013.6970748.

[27] D Balzarotti, M Cova, and Giovanni Vigna. ClearShot: Eavesdropping on Keyboard Input from Video. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 170–183, May 2008. DOI 10.1109/SP.2008.28.

[28] Nilanjan Banerjee, Jacob Sorber, Mark D Corner, Sami Rollins, and Deepak Ganesan. Triage: balancing energy and quality of service in a microserver. In *Proceedings of the International Conference on Mobile Systems, Applications and Services (MobiSys)*, pages 152–164. ACM, 2007. DOI 10.1145/1247660.1247680.

[29] Alberto Bartoli, Eric Medvet, Andrea De Lorenzo, and Fabiano Tarlao. Enterprise Wi-Fi: We Need Devices That Are Secure by Default. *Communications of the ACM*, 62(5):33–35, April 2019. DOI 10.1145/3319912.

[30] Patrick Baudin, Francois Bobot, and Richard Bonichon. Frama-C Software Analyzers. \url{http://frama-c.com/}, December 2014. Online at http://frama-c.com/.

[31] Johannes K Becker, David Li, and David Starobinski. Tracking Anonymized Bluetooth Devices. In *Proceedings on Privacy Enhancing Technologies (PETs)*, pages 50–65, 2019. DOI 10.2478/popets-2019-0036.

[32] Dmitri Bekerman, Bracha Shapira, Lior Rokach, and Ariel Bar. Unknown Malware Detection Using Network Traffic Classification. In *Proceedings of the IEEE Conference on Communications and Network Security (CNS)*, pages 134–142, 2015. Online at https://cyber.bgu.ac.il/wp-content/uploads/2017/10/07346821.pdf.

[33] Bruhadeshwar Bezawada, Maalvika Bachani, Jordan Peterson, Hossein Shirazi, Indrakshi Ray, and Indrajit Ray. Behavioral Fingerprinting of IoT Devices. In *Proceedings of the Workshop on Attacks and Solutions in Hardware Security (ASHES)*, pages 41–50. ACM Press, 2018. DOI 10.1145/3266444.3266452.

[34] XiaoXiao Bian. Implement a Virtual Development Platform Based on QEMU. In *Proceedings of the International Conference on Green Informatics (ICGI)*, volume 4, pages 93–97. IEEE, August 2017. DOI 10.1109/ICGI.2017.19.

[35] Bluetooth. Bluetooth Specifications. Online at https://www.bluetooth.com/specifications.

[36] BlueZ. BlueZ, 2018. Online at http://www.bluez.org/.

[37] George Boateng, John A Batsis, Patrick Proctor, Ryan Halter, and David Kotz. GeriActive: Wearable App for Monitoring and Encouraging Physical Activity among Older Adults. In *Proceedings of the IEEE Conference on Body Sensor Networks (BSN)*, pages 46–49, March 2018. DOI 10.1109/BSN.2018.8329655.

[38] George Boateng, Ryan Halter, John A Batsis, and David Kotz. ActivityAware: An App for Real-Time Daily Activity Level Monitoring on the Amulet Wrist-Worn Device. In *Proceedings of the IEEE PerCom Workshop on Pervasive Health Technologies (PerHealth)*, pages 431–435. IEEE, March 2017. DOI 10.1109/PERCOMW.2017.7917601.

[39] George Boateng and David Kotz. StressAware: An App for Real-Time Stress Monitoring on the Amulet Wearable Platform. In *Proceedings of the IEEE MIT Undergraduate Research Technology Conference (URTC)*. IEEE, January 2017. DOI 10.1109/URTC.2016.8284068.

[40] Katie Boeckl, Michael Fagan, William Fisher, Naomi Lefkovitz, Katerina N Megas, Ellen Nadeau, Danna Gabel, O ' Rourke, Ben Piccarreta, and Karen Scarfone. Considerations for Managing Internet of Things (IoT) Cybersecurity and Privacy Risks. Technical report, NIST, June 2019. DOI 10.6028/NIST.IR.8228.

[41] C. Bormann, M. Ersue, and A. Keranen. Terminology for Constrained-Node Networks, 2014.

[42] 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 *Proceedings of the ACM Asia Conference on Computer and Communications Security (Asia CCS)*, pages 479–484. ACM Press, 2019. DOI 10.1145/3321705.3329847.

[43] Helena Brekalo, Raoul Strackx, and Frank Piessens. Mitigating Password Database Breaches with Intel SGX. In *Proceedings of the Workshop on System Software for Trusted Execution (SysTEX)*, pages 1–6. ACM Press, 2016. DOI 10.1145/3007788.3007789.

[44] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In Marc Joye and Jean-Jacques Quisquater, editors, *Cryptographic Hardware and Embedded Systems (CHES)*, volume 3156 of *Lecture Notes in Computer Science*, pages 16–29. Springer Berlin Heidelberg, 2004. DOI 10.1007/978-3-540-28632-5_2.

[45] Brent Callaghan and Robert Gilligan. Snoop Version 2 Packet Capture File Format, 1995. Online at https://tools.ietf.org/html/rfc1761.

[46] José Camacho, Pedro García-Teodoro, and Gabriel Maciá-Fernández. Traffic monitoring and diagnosis with multivariate statistical network monitoring: a case study. In *IEEE Security and Privacy Workshops (SPW)*, number May. IEEE, 2017. DOI 10.1109/SPW.2017.11.

[47] José Camacho, Gabriel Maciá-Fernández, Jesús Díaz-Verdejo, and Pedro García-Teodoro. Tackling the Big Data 4 Vs for Anomaly Detection. In *Proceedings of the IEEE INFOCOM Workshop on Security and Privacy in Big Data*, pages 500–505, 2014. DOI 10.1109/INFCOMW.2014.6849282.

[48] José Camacho, Alejandro Pérez-Villegas, Pedro García-Teodoro, and Gabriel Maciá-Fernández. PCA-based Multivariate Statistical Network Monitoring

for Anomaly Detection. *Computers and Security*, 59(10):118–137, 2016. DOI 10.1016/j.cose.2016.02.008.

[49] José Camacho, Alejandro Pérez-Villegas, Rafael A. Rodríguez-Gómez, and Elena Jiménez-Mañas. Multivariate Exploratory Data Analysis (MEDA) Toolbox for Matlab. *Chemometrics and Intelligent Laboratory Systems*, 143:49–57, April 2015. Online at https://github.com/josecamachop/MEDA-Toolbox.

[50] Goutam Chakraborty, Kshirasagar Naik, Debasish Chakraborty, Norio Shiratori, and David Wei. Analysis of the Bluetooth device discovery protocol. *Wireless Networks*, 16(2):421–436, 2010. DOI 10.1007/s11276-008-0142-1.

[51] Yang Chen, Omprakash Gnawali, Maria Kazandjieva, Philip Levis, and John Regehr. Surviving Sensor Network Software Faults. In *Proceedings of the ACM SIGOPS Symposium on Operating Systems Principles*, (SOSP), pages 235–246. ACM, 2009. DOI 10.1145/1629575.1629598.

[52] Joseph I. Choi, Dave (Jing) Tian, Grant Hernandez, Christopher Patton, Benjamin Mood, Thomas Shrimpton, Kevin R. B. Butler, and Patrick Traynor. A Hybrid Approach to Secure Function Evaluation using SGX. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (Asia CCS)*, pages 100–113. ACM Press, 2019. DOI 10.1145/3321705.3329835.

[53] Cisco. NetFlow, 2018. Online at https://netflow.us/.

[54] Clang. Hardware-assisted AddressSanitizer Design (HWASAN) Documentation. Online at https://clang.llvm.org/docs/HardwareAssistedAddressSanitizerDesign.html.

[55] Shane S Clark, Benjamin Ransford, Amir Rahmati, Shane Guineau, Jacob Sorber, Wenyuan Xu, Kevin Fu, Amir Rahmati, Mastooreh Salajegheh, and Dan

Holcomb. WattsUpDoc: Power Side Channels to Nonintrusively Discover Untargeted Malware on Embedded Medical Devices. In *USENIX Workshop on Health Information Technologies*, pages 221–236, 2013.

[56] Nathan Cooprider, Will Archer, Eric Eide, David Gay, and John Regehr. Efficient Memory Safety for TinyOS. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, (SenSys), pages 205–218. ACM, 2007. DOI 10.1145/1322263.1322283.

[57] Cory Cornelius. *Usable Security for Wireless Body-Area Networks*. PhD thesis, Dartmouth College, 2013.

[58] Intel Corp. PCIe* Device Security Enhancements (Draft) Specification. Online at https://www.intel.com/content/www/us/en/io/pci-express/pcie-device-security-enhancements-spec.

[59] Victor Costan and Srinivas Devadas. Intel SGX Explained. Cryptology ePrint Archive, Report 2016/086, January 2016. Online at http://eprint.iacr.org/2016/086.

[60] Jordan Crook. Security researchers found a way to hack into the Amazon Echo, 2018. Online at https://techcrunch.com/2018/08/13/security-researchers-found-a-way-to-hack-into-the-amazon-echo/.

[61] Debra Curtis and Jonah Kowall. When Is NetFlow 'Good Enough'?, 2012. Online at https://www.gartner.com/id=1971021 https://community.cisco.com/t5/data-center-blogs/netflow-vs-packet-analysis/ba-p/3660051.

[62] Robert Davidson, Kevin Townsend, Chris Wang, and Carles Cufí. *Getting Started with Bluetooth Low Energy: Tools and Techniques for Low-Power Networking*. O'Reilly Media, 2014.

[63] Docker. Docker, 2019. Online at https://www.docker.com.

[64] A Dunkels, B Gronvall, and T Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the Annual IEEE International Conference on Local Computer Networks (LCN)*, pages 455–462, 2004. DOI 10.1109/LCN.2004.38.

[65] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, pages 29–42. ACM, 2006. DOI 10.1145/1182807.1182811.

[66] EDevice. HealthGo Mini, 2018. Online at https://www.edevice.com/products/healthgo-mini.

[67] EPSRC. Databox, 2018. Online at http://www.databoxproject.uk/.

[68] Biyi Fang, Nicholas D Lane, Mi Zhang, Aidan Boran, and Fahim Kawsar. BodyScan: Enabling Radio-based Sensing on Wearable Devices for Contactless Activity and Vital Sign Monitoring. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services*, (MobiSys), pages 97–110. ACM, 2016. DOI 10.1145/2906388.2906411.

[69] Denzil Ferreira, Eija Ferreira, Jorge Goncalves, Vassilis Kostakos, and Anind K Dey. Revisiting Human-battery Interaction with an Interactive Battery Interface. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, (UbiComp), pages 563–572. ACM, 2013. DOI 10.1145/2493432.2493465.

[70] USB Implementers Forum. USB Specification. Online at http://www.usb.org/developers/docs/.

[71] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, 2003. DOI 10.1145/781131.781133.

[72] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Adi Shamir, and Eran Tromer. Physical Key Extraction Attacks on PCs. *Communications of the ACM*, 59(6):70–79, May 2016. DOI 10.1145/2851486.

[73] David Gens. OS-level Software & Hardware Attacks and Defenses. In *MobiSys PhD Forum*, pages 7–8, 2018.

[74] Stylianos Gisdakis, Thanassis Giannetsos, and Panos Papadimitratos. SHIELD: A Data Verification Framework for Participatory Sensing Systems. In *Proceedings of the Conference on Wireless Security (WiSec)*, 2015. DOI 10.1145/2766498.2766503.

[75] Tarrah R. Glass-Vanderlan, Michael D. Iannacone, Maria S. Vincent, Qian, Chen, and Robert A. Bridges. A Survey of Intrusion Detection Systems Leveraging Host Data. *Computing Research Repository (CoRR)*, abs/1805.0:1–40, 2018. Online at https://arxiv.org/abs/1805.06070v2.

[76] Carles Gomez, Joaquim Oller, and Josep Paradells. Overview and evaluation of bluetooth low energy: An emerging low-power wireless technology. *Sensors*, 12(9):11734–11753, September 2012. DOI 10.3390/s120911734.

[77] Carles Gomez and Josep Paradells. Wireless home automation networks: A survey of architectures and technologies. *IEEE Communications Magazine*, 48(6):92–101, June 2010. DOI 10.1109/MCOM.2010.5473869.

[78] Google. Android security white paper. Technical report, Google for Work | Android, May 2015.

[79] Google. Google Home, 2018. Online at https://madeby.google.com/home/.

[80] Emily Greene, Patrick Proctor, and David Kotz. Secure Sharing of mHealth Data Streams through Cryptographically-Enforced Access Control. *Journal of Smart Health*, April 2018. DOI 10.1016/j.smhl.2018.01.003.

[81] Guofei Gu, Roberto Perdisci, Junjie Zhang, and Wenke Lee. BotMiner: Clustering Analysis of Network Traffic for Protocol-and Structure-Independent Botnet Detection. In *USENIX Security*, 2008. Online at http://faculty.cs.tamu.edu/guofei/paper/Gu_Security08_BotMiner.pdf.

[82] Guofei Gu, Phillip Porras, Vinod Yegneswaran, Martin Fong, and Wenke Lee. BotHunter: detecting malware infection through IDS-driven dialog correlation. *USENIX Security '07 Proceedings of the 16th USENIX Security Symposium*, 7:12, 2007. DOI 10.1.1.135.8028.

[83] Zhongshu Gu, Heqing Huang, Bytedance Jialong, Zhang Bytedance, Dong Su, Hani Jamjoom, Ankita Lamba, Dimitrios Pendarakis, and Ian Molloy. YerbaBuena: Securing Deep Learning Inference Data via Enclave-based Ternary Model Partitioning. Technical report, IBM Research, 2019. Online at https://arxiv.org/pdf/1807.00969.pdf https://arxiv.org/abs/1807.00969.

[84] Jose Gutierrez del Arroyo, Jason Bindewald, Scott Graham, and Mason Rice. Enabling Bluetooth Low Energy auditing through synchronized tracking of multiple connections. *International Journal of Critical Infrastructure Protection*, 18:58–70, 2017. DOI 10.1016/j.ijcip.2017.03.006.

[85] Christopher J Hansen. Internetworking with Bluetooth Low Energy. *GetMobile: Mobile Computing and Communications*, 19(2):34–38, April 2015. DOI 10.1145/2817761.2817774.

[86] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application Memory Isolation on Ultra-Low-Power MCUs. In *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[87] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application Memory Isolation on Ultra-Low-Power MCUs. *Proceedings of the USENIX Annual Technical Conference (USENIX ATC)*, 2018.

[88] David B Harmon. Cryptographic transfer of sensor data from the Amulet to a smartphone. Technical Report TR2017-826, Dartmouth College, Computer Science, Hanover, NH, May 2017. Online at http://www.cs.dartmouth.edu/reports/TR2017-826.pdf.

[89] Shaikh Shahriar Hassan, Soumik Das Bibon, Md Shohrab Hossain, and Mohammed Atiquzzaman. Security threats in Bluetooth technology. *Computers and Security*, 2017. DOI 10.1016/j.cose.2017.03.008.

[90] J. Hester, T. Peters, T. Yun, R. Peterson, J. Skinner, B. Golla, K. Storer, S. Hearndon, S. Lord, R. Halter, D. Kotz, and J. Sorber. Demo Abstract: The amulet wearable platform. In *Proceedings of the 14th ACM Conference on Embedded Networked Sensor Systems, SenSys 2016*, 2016. DOI 10.1145/2994551.2996527.

[91] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *Proceedings of the ACM Conference on*

Embedded Networked Sensor Systems (SenSys), SenSys '16, pages 216–229, New York, NY, USA, November 2016. ACM. DOI 10.1145/2994551.2994554.

[92] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Kevin Freeman, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. Amulet: An Energy-Efficient, Multi-Application Wearable Platform. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 216–229, November 2016. DOI 10.1145/2994551.2994554.

[93] Josiah Hester, Travis Peters, Tianlong Yun, Ronald Peterson, Joseph Skinner, Bhargav Golla, Kevin Storer, Steven Hearndon, Sarah Lord, Ryan Halter, David Kotz, and Jacob Sorber. Demo Abstract: The Amulet Wearable Platform. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 290–291, November 2016.

[94] Hexiwear. Hexiwear Open IoT Development SOlution, August 2016. Online at http://www.hexiwear.com/.

[95] Hanan Hindy, David Brosset, Ethan Bayne, Amar Seeam, Christos Tachtatzis, Robert Atkinson, and Xavier Bellekens. A Taxonomy and Survey of Intrusion Detection System Design Techniques, Network Threats and Datasets. *Computing Research Repository (CoRR)*, pages 1–35, 2018. Online at http://arxiv.org/abs/1806.03517.

[96] Matthew Hoekstra, Reshma Lal, Pradeep Pappachan, Vinay Phegade, and Juan Del Cuvillo. Using innovative instructions to create trustworthy software solutions. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM Press, 2013. DOI 10.1145/2487726.2488370.

[97] William Huang, Ye S Kuo, Pat Pannuto, and Prabal Dutta. Opo: A Wearable Sensor for Capturing High-fidelity Face-to-face Interactions. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems*, (SenSys), pages 61–75. ACM, 2014. DOI 10.1145/2668332.2668338.

[98] Galen Hunt, George Letey, and Ed Nightingale. The Seven Properties of Highly Secure Devices. Technical report, Microsoft Research, March 2017. Online at https://www.microsoft.com/en-us/research/publication/seven-properties-highly-secure-devices/.

[99] Ahmad Ibrahim. Securing Embedded Networks through Secure Collective Attestation. In *MobiSys PhD Forum*, pages 1–2, 2018.

[100] Intel. Intel Software Guard Extensions (SGX), 2018. Online at https://software.intel.com/sgx/.

[101] Prerit Jain, Soham Desai, Seongmin Kim, Ming-Wei Shih, JaeHyuk Lee, Changho Choi, Youjung Shin, Taesoo Kim, Brent Byunghoon Kang, and Dongsu Han. OpenSGX: An Open Platform for SGX Research. In *Proceedings of Network and Distributed System Security (NDSS)*, 2016. DOI 10.14722/ndss.2016.23011.

[102] Trevor Jim, J Greg Morrisett, Dan Grossman, Michael W Hicks, James Cheney, and Yanling Wang. Cyclone: A Safe Dialect of C. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC)*, pages 275–288. USENIX Association, 2002.

[103] Project Jupyter. Project Jupyter, 2019. Online at https://jupyter.org.

[104] Richard A. Kemmerer and Giovanni Vigna. Intrusion Detection: A Brief History and Overview. *Computer: Security & Privacy*, 35(4):27–30, 2002. DOI 10.1109/MC.2002.1012428.

[105] Dongwon Kim, Yohan Chon, Wonwoo Jung, Yungeun Kim, and Hojung Cha. Accurate Prediction of Available Battery Time for Mobile Applications. *ACM Transactions on Embedded Computing Systems*, 15(3), May 2016. DOI 10.1145/2875423.

[106] Anna J Knowles. Integrating Bluetooth Low Energy Peripherals with the Amulet. Technical Report TR2016-807, Dartmouth Computer Science, May 2016. Online at http://www.cs.dartmouth.edu/reports/TR2016-807.pdf.

[107] Paul C Kocher. Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems. In Neal Koblitz, editor, *Advances in Cryptology — CRYPTO '96*, volume 1109 of *Lecture Notes in Computer Science*, chapter 9, pages 104–113. Springer Berlin Heidelberg, Berlin, Heidelberg, July 1996. DOI 10.1007/3-540-68697-5_9.

[108] Kari Kostiainen, Elena Reshetova, Jan-Erik Ekberg, and N. Asokan. Old, New, Borrowed, Blue – A Perspective on the Evolution of Mobile Platform Security Architectures. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 13–24. ACM Press, 2011. DOI 10.1145/1943513.1943517.

[109] David Kotz, Ryan Halter, Cory Cornelius, Jacob Sorber, Minho Shin, Ronald Peterson, Shrirang Mare, Aarathi Prasad, Joseph Skinner, and Andrés Molina-Markham. Wearable computing device for secure control of physiological sensors and medical devices, with secure storage of medical records, and bioimpedance biometric. U.S. Patent 9,595,187, March 2017. Online at http://www.cs.dartmouth.edu/~dfk/papers/kotz-patent9595187.pdf.

[110] David Kotz and Travis Peters. Challenges to ensuring human safety throughout the life-cycle of Smart Environments. In *Proceedings of the ACM Workshop*

on the Internet of Safe Things (SafeThings), pages 1–7, November 2017. DOI 10.1145/3137003.3137012.

[111] Dmitrii Kuvaiskii, Somnath Chakrabarti, and Mona Vij. Snort Intrusion Detection System with Intel Software Guard Extension (Intel SGX). In *arXiv preprint arXiv:1802.00508*, 2018. Online at http://arxiv.org/abs/1802.00508.

[112] Anukool Lakhina, Mark Crovella, and Christophe Diot. Diagnosing Network-Wide Traffic Anomalies. In *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM)*, pages 219—-230, 2004. DOI 10.1145/1030194.1015492.

[113] Nicholas D Lane, Petko Georgiev, Cecilia Mascolo, and Ying Gao. ZOE: A Cloud-less Dialog-enabled Continuous Sensing Wearable Exploiting Heterogeneous Computation. In *Proceedings of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 273–286. ACM, May 2015. DOI 10.1145/2742647.2742672.

[114] Quantum Leaps. QP/C Framework, December 2015. Online at http://www.state-machine.com/qpc/index.html.

[115] Eliot Lear, Ralph Droms, and Dan Romascanu. Manufacturer Usage Description Specification, 2019. Online at https://tools.ietf.org/html/rfc8520.

[116] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and Others. TinyOS: An operating system for sensor networks. *Ambient intelligence*, 35:115–148, 2005.

[117] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B Giffin, Pat Pannuto, Dutta Dutta, and Philip Levis. Multiprogramming a 64 kB Computer Safely

and Efficiently. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, SOSP '17, New York, NY, USA, 2017. ACM. DOI 10.1145/3132747.3132786.

[118] Lexico.com. Lexico, 2019. Online at https://www.lexico.com/en/.

[119] Ding Li, Shuai Hao, William G J Halfond, and Ramesh Govindan. Calculating Source Line Level Energy Information for Android Applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, (ISSTA), pages 78–89. ACM, 2013. DOI 10.1145/2483760.2483780.

[120] Fangyu Li, Yang Shi, Aditya Shinde, Jin Ye, and Wenzhan Song. Enhanced Cyber-Physical Security in Internet of Things Through Energy Auditing. *IEEE Internet of Things Journal*, 6(3):5224–5231, June 2019. DOI 10.1109/JIOT.2019.2899492.

[121] Huaqing Lin, Zheng Yan, Yu Chen, and Lifang Zhang. A Survey on Network Security-Related Data Collection Technologies. *IEEE Access*, 6(3):18345–18365, 2018. DOI 10.1109/ACCESS.2018.2817921.

[122] Konrad Lorincz, Bor-rong R Chen, Geoffrey W Challen, Atanu R Chowdhury, Shyamal Patel, Paolo Bonato, and Matt Welsh. Mercury: A Wearable Sensor Network Platform for High-Fidelity Motion Analysis. In *Proceedings of the International Conference on Embedded Networked Sensor Systems (SenSys)*, SenSys '09, pages 183–196, New York, NY, USA, November 2009. ACM. DOI 10.1145/1644038.1644057.

[123] Xiao Ma, Peng Huang, Xinxin Jin, Pei Wang, Soyeon Park, Dongcai Shen, Yuanyuan Zhou, Lawrence K Saul, and Geoffrey M Voelker. eDoctor: Automatically Diagnosing Abnormal Battery Drain Issues on Smartphones. In

Proceedings of the USENIX Conference on Networked Systems Design and Implementation, (NSDI), pages 57–70. USENIX Association, 2013. Online at https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/ma.

[124] Matthew V. Mahoney. Network Traffic Anomaly Detection Based on Packet Bytes. In *Proceedings of the ACM Symposium on Applied Computing (SAC)*, pages 1–5, 2003. DOI 10.1145/952589.952601.

[125] Antonio L Maia Neto, Artur L F Souza, Italo Cunha, Michele Nogueira, Ivan O Nunes, Leonardo Cotta, Nicolas Gentille, Antonio A F Loureiro, Diego F Aranha, Harsh K Patil, and Leonardo B Oliveira. AoT: Authentication and Access Control for the Entire IoT Device Life-Cycle. In *Proceedings of the ACM Conference on Embedded Network Sensor Systems (SenSys)*, pages 1–15. ACM, 2016. DOI 10.1145/2994551.2994555.

[126] Shrirang Mare. *Seamless Authentication For Ubiquitous Devices*. PhD thesis, Dartmouth College, 2016. Online at http://www.cs.dartmouth.edu/reports/abstracts/TR2016-793/.

[127] Shrirang Mare, Reza Rawassizadeh, Ronald Peterson, and David Kotz. Continuous Smartphone Authentication using Wristbands. In *Proceedings of the Workshop on Usable Security and Privacy (USEC)*, 2019. DOI 10.14722/usec.2019.23013.

[128] Shrirang Mare, Jacob Sorber, Minho Shin, Cory Cornelius, and David Kotz. Hide-n-Sense: preserving privacy efficiently in wireless mHealth. *Mobile Networks and Applications (MONET)*, 19(3):331–344, June 2014. DOI 10.1007/s11036-013-0447-x.

[129] Steven B McGowan. Secure Input/Output Device Management. Online at http://www.sumobrain.com/patents/wipo/Secure-device-management/WO2017023434A1.html.

[130] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative instructions and software model for isolated execution. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*. ACM Press, 2013. DOI 10.1145/2487726.2488368.

[131] Yair Meidan, Michael Bohadana, Asaf Shabtai, Juan David Guarnizo, Martín Ochoa, Nils Ole Tippenhauer, and Yuval Elovici. ProfilIoT: A Machine Learning Approach for IoT Device Identification Based on Network Traffic Analysis. In *Proceedings of the Symposium on Applied Computing (SAC'17)*, pages 506–509. ACM Press, 2017. DOI 10.1145/3019612.3019878.

[132] Markus Miettinen, N. Asokan, Thien Duc Nguyen, Ahmad-Reza Sadeghi, and Majid Sobhani. Context-Based Zero-Interaction Pairing and Key Evolution for Advanced Personal Devices. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 880–891. ACM Press, 2014. DOI 10.1145/2660267.2660334.

[133] Markus Miettinen, Samuel Marchal, Ibbad Hafeez, Tommaso Frassetto, N. Asokan, Ahmad Reza Sadeghi, and Sasu Tarkoma. IoT Sentinel: Automated Device-Type Identification for Security Enforcement in IoT. In *Proceedings of the International Conference on Distributed Computing Systems (DCS'17*, pages 2511–2514. IEEE, 2017. DOI 10.1109/ICDCS.2017.284.

[134] Chulhong Min, Youngki Lee, Chungkuk Yoo, Seungwoo Kang, Sangwon Choi, Pillsoon Park, Inseok Hwang, Younghyun Ju, Seungpyo Choi, and

Junehwa Song. PowerForecaster: Predicting Smartphone Power Impact of Continuous Sensing Applications at Pre-installation Time. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (SenSys)*, pages 31–44. ACM, 2015. DOI 10.1145/2809695.2809728.

[135] Chulhong Min, Chungkuk Yoo, Inseok Hwang, Seungwoo Kang, Youngki Lee, Seungchul Lee, Pillsoon Park, Changhun Lee, Seungpyo Choi, and Junehwa Song. Sandra Helps You Learn: The More You Walk, the More Battery Your Phone Drains. In *Proceedings of the ACM International Joint Conference on Pervasive and Ubiquitous Computing*, (UbiComp), pages 421–432. ACM, 2015. DOI 10.1145/2750858.2807553.

[136] Saeed Mirzamohammadi and Ardalan A Sani. Viola: Trustworthy Sensor Notifications for Enhanced Privacy on Mobile Systems. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 263–276. ACM, June 2016. DOI 10.1145/2906388.2906391.

[137] Varun Mishra, Byron Lowens, Sarah Lord, Kelly Caine, and David Kotz. Investigating contextual cues as indicators for EMA delivery. In *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers on - UbiComp '17*, pages 935–940, New York, New York, USA, 2017. ACM Press. DOI 10.1145/3123024.3124571.

[138] Radhika Mittal, Aman Kansal, and Ranveer Chandra. Empowering Developers to Estimate App Energy Consumption. In *Proceedings of the Annual International Conference on Mobile Computing and Networking*, (Mobicom), pages 317–328. ACM, 2012. DOI 10.1145/2348543.2348583.

[139] Andrés Molina-Markham, Ronald Peterson, Joseph Skinner, Tianlong Yun, Bhargav Golla, Kevin Freeman, Travis Peters, Jacob Sorber, Ryan Halter, and David Kotz. Amulet: A secure architecture for mHealth applications for low-power wearable devices. In *Proceedings of the Workshop on Mobile Medical Applications - Design and Development (WMMADD)*, pages 16–21, New York, New York, USA, November 2014. ACM Press. DOI 10.1145/2676431.2676432.

[140] Nour Moustafa, Benjamin Turnbull, and Kim-Kwang Raymond Choo. An Ensemble Intrusion Detection Technique Based on Proposed Statistical Flow Features for Protecting Network Traffic of Internet of Things. *IEEE Internet of Things Journal*, 6(3):4815–4830, June 2019. DOI 10.1109/JIOT.2018.2871719.

[141] Anand Mudgerikar, Puneet Sharma, and Elisa Bertino. E-Spion: A System-Level Intrusion Detection System for IoT Devices. In *Proceedings of the ACM Asia Conference on Computer and Communications Security (Asia CCS)*, pages 493–500. ACM Press, 2019. DOI 10.1145/3321705.3329857.

[142] Patrick Murphy, Ashu Sabharwal, and Behnaam Aazhang. DESIGN OF WARP: A WIRELESS OPEN-ACCESS RESEARCH PLATFORM. In *Proceedings of the European Signal Processing Conference (EUSIPCO)*, 2006. Online at https://www.eurasip.org/Proceedings/Eusipco/Eusipco2006/papers/1568982373.pdf.

[143] MyriadRF. LimeSDR, 2018. Online at https://myriadrf.org/projects/limesdr/.

[144] Muhammad Naveed, Xiaoyong Zhou, Soteris Demetriou, XiaoFeng Wang, and Carl A. Gunter. Inside Job: Understanding and Mitigating the Threat of External Device Mis-Bonding on Android. In *Proceedings of the Network and*

Distributed System Security Symposium (NDSS). Internet Society, February 2014. DOI 10.14722/ndss.2014.23097.

[145] Zhenyu Ning, Fengwei Zhang, Weisong Shi, and Weidong Shi. Position Paper: Challenges Towards Securing Hardware-assisted Execution Environments. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–8. ACM Press, 2017. DOI 10.1145/3092627.3092633.

[146] Adam J Oliner, Anand P Iyer, Ion Stoica, Eemil Lagerspetz, and Sasu Tarkoma. Carat: Collaborative Energy Diagnosis for Mobile Devices. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems*, (SenSys). ACM, 2013. DOI 10.1145/2517351.2517354.

[147] Abhinav Pathak, Y Charlie Hu, and Ming Zhang. Where is the Energy Spent Inside My App?: Fine Grained Energy Accounting on Smartphones with Eprof. In *Proceedings of the ACM European Conference on Computer Systems*, (EuroSys), pages 29–42. ACM, 2012. DOI 10.1145/2168836.2168841.

[148] Vern Paxson. Strategies for Sound Internet Measurement. In *Proceedings of the ACM SIGCOMM Conference on Internet Measurement (IMC)*, pages 263–271, 2004. DOI 10.1145/1028788.1028824.

[149] Pebble. Battery Performance Guide, November 2015. Online at https://developer.getpebble.com/guides/best-practices/battery-perform-guide/.

[150] Pebble. Pebble, November 2015. Online at https://www.pebble.com/.

[151] F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[152] Geovandro C. C. F. Pereira, Renan C. A. Alves, Felipe L. da Silva, Roberto M. Azevedo, Bruno C. Albertini, and Cíntia B. Margi. Performance Evaluation of Cryptographic Algorithms over IoT Platforms and Operating Systems. *Security and Communication Networks*, 2017, August 2017. DOI 10.1155/2017/2046735.

[153] Rui Pereira, Marco Couto, Francisco Ribeiro, Rui Rua, João Paulo Fernandes, Jácome Cunha, and Joao Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? In *Proceedings of the ACM SIGPLAN International Conference on Software Language Engineering (SLE)*, pages 256–267, 2017. DOI 10.1145/3136014.3136031.

[154] Travis Peters. An Assessment of Single-Channel EMG Sensing for Gestural Input. *Technical Report TR2015-767, Dartmouth Computer Science*, September 2014.

[155] Travis Peters. A Survey of Trustworthy Computing on Mobile & Wearable Systems. *Technical Report TR2017-823, Dartmouth Computer Science*, May 2017. Online at http://www.cs.dartmouth.edu/reports/abstracts/TR2017-823/.

[156] Travis Peters, Reshma Lal, Srikanth Varadarajan, Pradeep Pappachan, and David Kotz. BASTION-SGX: Bluetooth and Architectural Support for Trusted I/O on SGX. In *Proceedings of the Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, pages 1–9, June 2018. DOI 10.1145/3214292.3214295.

[157] Timothy Pierson, Travis Peters, Ronald Peterson, and David Kotz. Proximity Detection with Single-Antenna IoT Devices. In *Proceedings of the Conference on Mobile Computing and Networking (MobiCom)*, pages 1—–15, October 2019. DOI 10.1145/3300061.3300120.

[158] Timothy J Pierson, Xiaohui Liang, Ronald Peterson, and David Kotz. Wanda: securely introducing mobile devices - Extended version. Technical Report TR2016-789, Dartmouth Computer Science, February 2016. Online at http://www.cs.dartmouth.edu/reports/abstracts/TR2016-789/.

[159] Gunnar C Pope, Varun Mishra, Stephanie Lewia, Byron Lowens, David Kotz, Sarah Lord, and Ryan Halter. An Ultra-Low Resource Wearable EDA Sensor Using Wavelet Compression. In *Proceedings of the IEEE Conference on Body Sensor Networks (BSN)*, pages 193–196, March 2018. DOI 10.1109/BSN.2018.8329691.

[160] N.R. Potlapally, S. Ravi, A. Raghunathan, and N.K. Jha. A study of the energy consumption characteristics of cryptographic algorithms and security protocols. *IEEE Transactions on Mobile Computing*, 5(2):128–143, February 2006. DOI 10.1109/TMC.2006.16.

[161] Christian Priebe, Kapil Vaswani, and Manuel Costa. EnclaveDB: A Secure Database using SGX. In *IEEE Symposium on Security and Privacy (SP)*, 2018. DOI 10.1109/SP.2018.00025.

[162] Punch Through. Maximizing BLE Throughput on iOS and Android. Online at https://punchthrough.com/maximizing-ble-throughput-on-ios-and-android/.

[163] QEMU. QEMU, 2018. Online at https://www.qemu.org/.

[164] Ashwin Rao, Arash Molavi Kakhki, Abbas Razaghpanah, Amy Tang, Shen Wang, Justine Sherry, Phillipa Gill, Arvind Krishnamurthy, Arnaud Legout, Alan Mislove, and David Choffnes. Using the Middle to Meddle with Mobile. Technical report, Northeastern University, 2013.

[165] Shahid Raza, Linus Wallgren, and Thiemo Voigt. SVELTE: Real-time intrusion detection in the Internet of Things. *Ad Hoc Networks*, 11(8):2661–2674, November 2013. DOI 10.1016/J.ADHOC.2013.04.014.

[166] Jingjing Ren, Daniel J. Dubois, David Choffnes, Anna Maria Mandalari, Roman Kolcun, and Hamed Haddadi. Information Exposure From Consumer IoT Devices: A Multidimensional, Network-Informed Measurement Approach. In *Proceedings of the Internet Measurement Conference (IMC'19)*, pages 267–279, New York, New York, USA, October 2019. ACM Press. DOI 10.1145/3355369.3355577.

[167] Mike Ryan. crackle. Online at https://github.com/mikeryan/crackle/.

[168] Shalaka Chittaranjan Satam. *Bluetooth Anomaly Based Intrusion Detection System (Master's Thesis)*. PhD thesis, Arizona State University, 2017. Online at http://hdl.handle.net/10150/625890.

[169] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *USENIX Security Symposium*, pages 1–12, 2010.

[170] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *USENIX Annual Technical Conference (USENIX ATC'12)*, pages 309–318, 2012. Online at https://www.usenix.org/conference/atc12/technical-sessions/presentation/serebryany.

[171] Kostya Serebryany. OSS-Fuzz - Google's continuous fuzzing service for open source software. In *USENIX Security Symposium (USENIX Security'17)*, 2017.

Online at https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/serebryany.

[172] Kostya Serebryany. ARM Memory Tagging Extension and How It Improves C/C++ Memory Safety. *;login: USENIX Security*, 44(2):12–16, 2019. Online at https://www.usenix.org/publications/login/summer2019/serebryany.

[173] Robert W Shirey. Internet Security Glossary, Version 2. RFC 4949, August 2013. DOI 10.17487/RFC4949.

[174] Victor Shnayder, Mark Hempstead, Bor R Chen, Geoff W Allen, and Matt Welsh. Simulating the Power Consumption of Large-scale Sensor Network Applications. In *Proceedings of the International Conference on Embedded Networked Sensor Systems*, (SenSys), pages 188–200. ACM, 2004. DOI 10.1145/1031495.1031518.

[175] Diksha Shukla, Rajesh Kumar, Abdul Serwadda, and Vir V Phoha. Beware, Your Hands Reveal Your Secrets! In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 904–917. ACM, 2014. DOI 10.1145/2660267.2660360.

[176] Arunan Sivanathan, Hassan Habibi Gharakheili, Franco Loi, Adam Radford, Chamith Wijenayake, Arun Vishwanath, and Vijay Sivaraman. Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics. *IEEE Transactions on Mobile Computing*, 18(8):1745–1759, August 2019. DOI 10.1109/TMC.2018.2866249.

[177] SocialCompare. Comparison of popular Smartwatches, December 2015. Online at http://socialcompare.com/en/comparison/comparison-of-popular-smartwatches.

[178] Robin Sommer and Vern Paxson. Outside the Closed World: On Using Machine Learning for Network Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy (Oakland)*, pages 305–316, 2010. DOI 10.1109/SP.2010.25.

[179] Sony. Open SmartWatch Project, July 2019. Online at https://developer.sony.com/develop/open-devices/more-information/discontinued-projects-initiatives/open-smartwatch-project/.

[180] Jacob Sorber, Nilanjan Banerjee, Mark D Corner, and Sami Rollins. Turducken: Hierarchical Power Management for Mobile Devices. In *Proceedings of the International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 261–274. ACM, 2005. DOI 10.1145/1067170.1067198.

[181] Jacob Sorber, Minho Shin, Ronald Peterson, Cory Cornelius, Shrirang Mare, Aarathi Prasad, Zachary Marois, Emma Smithayer, and David Kotz. An Amulet for trustworthy wearable mHealth. In *Workshop on Mobile Computing Systems and Applications (HotMobile)*, February 2012. DOI 10.1145/2162081.2162092.

[182] TE Connectivity. New Survey from TE Connectivity Uncovers America's Desire for Wearables. Technical report, PR Newswire, May 2015. Online at http://www.prnewswire.com/news-releases/new-survey-from-te-connectivity-uncovers-americas-desire-for-wearables-300089552.html.

[183] The Creators Project. Choose BLOCKS, July 2019. Online at http://www.chooseblocks.com.

[184] Srikanth Varadarajan, Reshma Lal, Steven B. McGowan, Hakan Magnus Eriksson, and Travis W. Peters. System, apparatus and method for providing

trusted input/output communications, August 2019. Online at https://patents.google.com/patent/US10372656B2/en.

[185] Amit Vasudevan, Emmanuel Owusu, Zongwei Zhou, James Newsome, and Jonathan M McCune. Trustworthy Execution on Mobile Devices: What Security Properties Can My Mobile Platform Give Me? In *Trust and Trustworthy Computing*, volume 7344 of *Lecture Notes in Computer Science*, pages 159–178. Springer, 2012. DOI 10.1007/978-3-642-30921-2_10.

[186] Jesse Walker and Jiangtao Li. Key Exchange with Anonymous Authentication Using DAA-SIGMA Protocol. In *Proceedings of the International Conference on Trusted Systems (INTRUST)*. Springer-Verlag, 2010. DOI 10.1007/978-3-642-25283-9_8.

[187] Jack Wampler, Ian Martiny, and Eric Wustrow. ExSpectre: Hiding Malware in Speculative Execution. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*, 2019. DOI 10.14722/ndss.2019.23409.

[188] Ke Wang and Salvatore J. Stolfo. *Anomalous Payload-Based Network Intrusion Detection*. Springer Berlin Heidelberg, 2004. DOI 10.1007/978-3-540-30143-1_11.

[189] Samuel Weiser and Mario Werner. SGXIO: Generic Trusted I/O Path for Intel SGX. In *Proceedings of the ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 261–268. ACM Press, 2017. DOI 10.1145/3029806.3029822.

[190] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the ACM Asia Conference on Computer*

and Communications Security (Asia CCS), pages 73–85. ACM Press, 2019. DOI 10.1145/3321705.3329820.

[191] Christian Wressnegger, Guido Schwenk, Daniel Arp, and Konrad Rieck. A Close Look on N-grams in Intrusion Detection: Anomaly Detection vs. Classification. In *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, pages 67–76, 2013. DOI 10.1145/2517312.2517316.

[192] Haowei Wu, Shengqian Yang, and Atanas Rountev. Static Detection of Energy Defect Patterns in Android Applications. In *Proceedings of the International Conference on Compiler Construction*, (CC), pages 185–195. ACM, 2016. DOI 10.1145/2892208.2892218.

[193] Qiang Xu, Rong Zheng, Walid Saad, and Zhu Han. Device Fingerprinting in Wireless Networks: Challenges and Opportunities. *IEEE Communications Surveys & Tutorials*, 18(1):94–104, January 2016. DOI 10.1109/comst.2015.2476338.

[194] Yuanzhong Xu, Weidong Cui, and M Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P)*, pages 640–656, May 2015. DOI 10.1109/SP.2015.45.

[195] Junjie Yin, Zheng Yang, Hao Cao, Tongtong Liu, Zimu Zhou, and Chenshu Wu. A Survey on Bluetooth 5.0 and Mesh: New Milestones of IoT. *ACM Transactions on Sensor Networks (TOSN)*, 15(3):28, 2019. DOI 10.1145/3317687.

[196] Chanmin Yoon, Dongwon Kim, Wonwoo Jung, Chulkoo Kang, and Hojung Cha. AppScope: Application Energy Metering Framework for Android Smartphones Using Kernel Activity Monitoring. In *Proceedings of the USENIX Conference on Annual Technical Conference*, (USENIX ATC), page 36. USENIX

Association, 2012. Online at https://www.usenix.org/conference/atc12/technical-sessions/presentation/yoon.

[197] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, Haojin Zhu, and Hao-Jin Zhu. HoMonit: Monitoring Smart Home Apps from Encrypted Traffic. In *Proceedings of the Conference on Computer and Communications Security (CCS)*, 2018. DOI 10.1145/3243734.3243820.

[198] Lu Zhao, Guodong Li, B De Sutter, and J Regehr. ARMor: Fully verified software fault isolation. In *Embedded Software (EMSOFT), 2011 Proceedings of the International Conference on*, pages 289–298. Univ. of Utah, Salt Lake City, UT, USA, IEEE, October 2011. Online at http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6064537.

[199] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. ARMlock: Hardware-based Fault Isolation for ARM. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 558–569. ACM, 2014. DOI 10.1145/2660267.2660344.

[200] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. Building Verifiable Trusted Path on Commodity x86 Computers. In *IEEE Symposium on Security and Privacy*, pages 616–630. IEEE, May 2012. DOI 10.1109/SP.2012.42.

[201] ZWear. ZWear - a wearable platform for makers, July 2019. Online at http://zwear.org.