UNIVERSITY OF CALIFORNIA,
IRVINE

Understanding and Guaranteeing Security, Privacy, and Safety of Smart Homes

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Computer Engineering

by

Rahmadi Trimananda

Dissertation Committee:
Professor Brian Demsky, Chair
Professor Athina Markopoulou
Professor Guoqing (Harry) Xu

2020

DEDICATION

To God the Trinity, my family, and humanity

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First of all, I would like to express my gratitude to the Triune God: God the Father, God the Son, and God the Holy Spirit, the author of my life who brought me into existence, created me with creativity and talents, and gave me breath and every other blessing that have enabled me to pursue research to discover His glory and beauty in an astoundingly most profound way through computer science and, in particular, software engineering and programming languages. Truly, He reigns in every corner of our universe, and His traces of beauty are evident in every bit we have and will ever discover in research. It is exactly as what Abraham Kuijper (pronounced as "Kuyper"), a great theologian and former Prime Minister of the Netherlands, said: **"There is not a square inch in the whole domain of our human existence over which Christ, who is Sovereign over all, does not cry, Mine!"**

Although I can never say it enough, I sincerely thank Professor Brian Demsky for he has played a pivotal role in shaping and forming my skills in the past 5 years, in particular, in software engineering and programming languages. I sincerely thank him for his patience in teaching, and his passion for research. This world would not be the same without him!

I would like to also thank professors, fellow researchers, and friends who have helped me along the way. I thank Professor Guoqing Harry Xu and Professor Athina Markopoulou, who have tirelessly co-advised me in my research. I thank Professor Rainer Dömer and Professor Mohammad Al Faruque who have served on my candidacy exam committee. I also thank Janus Varmarken, Seyed Amir Hossein Aqajari, Ali Younis, Bin Xu, Bojun Wang, Jason Chuang, and Thomas Kwa who have been my student co-authors—this list includes Anastasia Shuba who gave us a lot of insights for our research on Packet-Level Signatures for Smart Home Devices although she did not become a co-author. Finally, I thank Peizhao Ou, Marwen Zorgui, Laleh Aghababaie Beni, Rohit Zambre, Byron Hawkins, Hamed Gorjiara, Zachary Snyder, Wei Yu Luo, Ahmed Al Nahian, and many others who have been colleagues and friends.

I would like to thank Kumar Deepak, Jason Villarreal, and Pramod Chandraiah who have given me the internship opportunity at Xilinx and become my mentors there. I thank them for their invaluable help, support, and guidance during my internship that has given me a perfect opportunity to work on an industrial compiler.

I would like to thank the ACM SIGPLAN and IEEE for granting permission to include content that has been previously published in conference proceedings into this dissertation.

I would like to thank the creators, developers, and community of Java, Python, and C++ as the 3 main languages I have learned a lot about during PhD research. I used them in almost every project I worked on. I also thank the creators, developers, and community of Java Pathfinder, and Pcap4J and DBSCAN libraries for Java as they have been pivotal for the projects I worked on.

Finally, my family has been the greatest supporter of my PhD journey. In particular, I am grateful for my wife, Adeline Aninda, who has been my greatest fan and supporter from the very beginning; my son, Alden Oliver Yeh, who came along recently into our little family; my father, Yap Man On; my sisters, Ainon and Aina Manan; my father-in-law, Teguh Adie Santosa; my mother-in-law, Chriswanti Kusmanto; my brothers-in-law, Adityo and Adrian Nugroho; my sister-in-law, Felia Indah Kusuma; and my extended family. I especially dedicate my PhD to my mother, Tjiu Kon Nyian, who would never be able to celebrate physically with me at the finish line—God has called her back to be with Him in heaven on April 23, 2020.

My doctoral research and the work presented in this dissertation would not have been possible without the support from the UCI EECS Department Fellowship Award, Broadcom Foundation Fellowship Award, and a UCI Seed Funding Award at UCI. It is also partly supported by the National Science Foundation under grants CCF-1319786, CCF-1837120, CCF-2006948, CNS-1613023, CNS-1649372, CNS-1703598, CNS-1763172, CNS-1815666, CNS-1900654, CNS-2006437, CNS-200 7737, OAC-1740210 and the Office of Naval Research under grants N00014-16-1-2913 and N00014-18-1-2037.

VITA

Rahmadi Trimananda

EDUCATION

Doctor of Philosophy in Computer Engineering	**2020**
University of California, Irvine	*Irvine, CA, USA*
Master of Science in Computer Engineering	**2009**
Delft University of Technology	*Delft, The Netherlands*
Bachelor of Science in Computer Engineering	**2006**
Pelita Harapan University	*Tangerang, Indonesia*

RESEARCH EXPERIENCE

Graduate Student Researcher	**2015–2020**
University of California, Irvine	*Irvine, CA, USA*
Research Assistant	**2008–2009**
Delft University of Technology	*Delft, The Netherlands*
Research Intern	**June-July 2008**
TNO Delft	*Delft, The Netherlands*

TEACHING EXPERIENCE

Lecturer	**2009–2011**
Pelita Harapan University	*Tangerang, Indonesia*

REFEREED JOURNAL PUBLICATIONS[1]

Packet-Level Signatures for Smart Home Devices **Feb 2020**
ISOC Network and Distributed System Security Symposium

REFEREED CONFERENCE PUBLICATIONS

Understanding and Automatically Detecting Conflicting Interactions between Smart Home IoT Applications **Nov 2020**
ACM SIGSOFT Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

Securing Smart Home Devices against Compromised Cloud Servers (Poster) **Jun 2020**
3rd USENIX Workshop on Hot Topics in Edge Computing (HotEdge)

Packet-Level Signatures for Smart Home Devices **Feb 2020**
ISOC Network and Distributed System Security Symposium (NDSS)

Vigilia: Securing Smart Home Edge Computing **Nov 2018**
ACM/IEEE Symposium on Edge Computing (SEC)

SOFTWARE

IoTCheck `http://plrg.ics.uci.edu/iotcheck/`
An automatic conflict detection tool that uses model checking to automatically detect smart home app conflicts.

Fidelius `http://plrg.ics.uci.edu/fidelius/`
A runtime system for secure cloud-based storage and communication even in the presence of compromised servers.

PingPong `http://plrg.ics.uci.edu/pingpong/`
A tool that can automatically extract packet-level signatures for device events from network traffic.

Vigilia `http://plrg.ics.uci.edu/vigilia/`
A system that shrinks the attack surface of smart home IoT systems by restricting network access of devices.

[1]These are only publications made during PhD. Please check `https://rtrimana.github.io/` for the complete list.

ABSTRACT OF THE DISSERTATION

Understanding and Guaranteeing Security, Privacy, and Safety of Smart Homes

By

Rahmadi Trimananda

Doctor of Philosophy in Computer Engineering

University of California, Irvine, 2020

Professor Brian Demsky, Chair

In this work, we view smart home from 3 different sides: devices, platforms, and apps. We first attempt to better understand the problems on each side and later explore new methods and techniques to better guarantee security, privacy, and safety of smart homes.

On the devices side, we discovered that smart home devices are vulnerable to passive inference attacks based on network traffic, even in the presence of encryption. We first present this passive inference attack and our techniques that we developed to exploit this vulnerability on smart home devices. We created PingPong, a tool that can automatically extract packet-level signatures for device events (*e.g.*, light bulb turning ON/OFF) from network traffic. We evaluated PingPong on popular smart home devices ranging from smart plugs and thermostats to cameras, voice-activated devices, and smart TVs. We were able to: (1) automatically extract previously unknown signatures that consist of simple sequences of packet lengths and directions; (2) use those signatures to detect the devices or specific events with an average recall of more than 97%; (3) show that the signatures are unique among hundreds of millions of packets of real world network traffic; (4) show that our methodology is also applicable to publicly available datasets; and (5) demonstrate its robustness in different settings: events triggered by local and remote smartphones, as well as by home-automation systems. Furthermore, we also present our discussion and evaluation on existing techniques (*e.g.*, packet padding) as possible defenses against passive inference attacks and their analyses.

On the platforms side, smart home platforms such as SmartThings enable homeowners to manage devices in sophisticated ways to save energy, improve security, and provide conveniences. Unfortunately, we discovered that smart home platforms contain vulnerabilities, potentially impacting home security and privacy. Aside from the traditional defense techniques to enhance the security and privacy of smart home devices, we also created Vigilia, a system that shrinks the attack surface of smart home IoT systems by restricting the network access of devices. As existing smart home systems are closed, we have created an open implementation of a similar programming and configuration model in Vigilia and extended the execution environment to maximally restrict communications by instantiating device-based network permissions. We have implemented and compared Vigilia with forefront IoT-defense systems; our results demonstrate that Vigilia outperforms these systems and incurs negligible overhead.

On the apps side, smart home platforms allow developers to write apps to make smart home devices work together to accomplish tasks, *e.g.*, home security and energy conservation—smart home devices provide the convenience of remotely controlling and automating home appliances. A smart home app typically implements narrow functionality and thus to fully implement desired functionality homeowners may need to install multiple apps. These different apps can conflict with each other and these conflicts can result in undesired actions such as locking the door during a fire. We study conflicts between apps on Samsung SmartThings, the most popular platform for developing and deploying smart home IoT devices. By collecting and studying 198 official and 69 third-party apps, we found significant app conflicts in 3 categories: (1) close to 60% of app pairs that access the same device, (2) more than 90% of app pairs with physical interactions, and (3) around 11% of app pairs that access the same global variable. Our results suggest that the problem of conflicts between smart home apps is serious and can create potential safety risks. We then developed an automatic conflict detection tool that uses model checking to automatically detect up to 96% of the conflicts.

Chapter 1

Introduction

This dissertation presents our work in understanding and guaranteeing security, privacy, and safety of smart homes. In this work, we look at the three aspects of smart homes: devices, platforms, and apps[1].

Devices. Various sources have reported that smart home devices have vulnerabilities and are prone to attacks [81, 173, 75, 218, 164, 178, 104, 211, 48, 111, 15]. We specifically discovered that these devices are prone to a new passive inference attack: packet-level signatures. We developed a tool called PingPong to further evaluate this vulnerability. We also propose and evaluate a number of techniques to defend against it.

Platforms. Smart home platforms allow users to develop apps that integrate their smart home devices to perform certain functionalities. These systems also allow users to install multiple apps for their homes. Unfortunately, these integration systems have vulnerabilities that make them prone to attacks. We devised and implemented smart home hardening techniques in a tool called Vigilia. We evaluated the tool and we found that it significantly reduced the attack surface of smart home systems.

[1]The results presented here are based on [195, 197, 193]

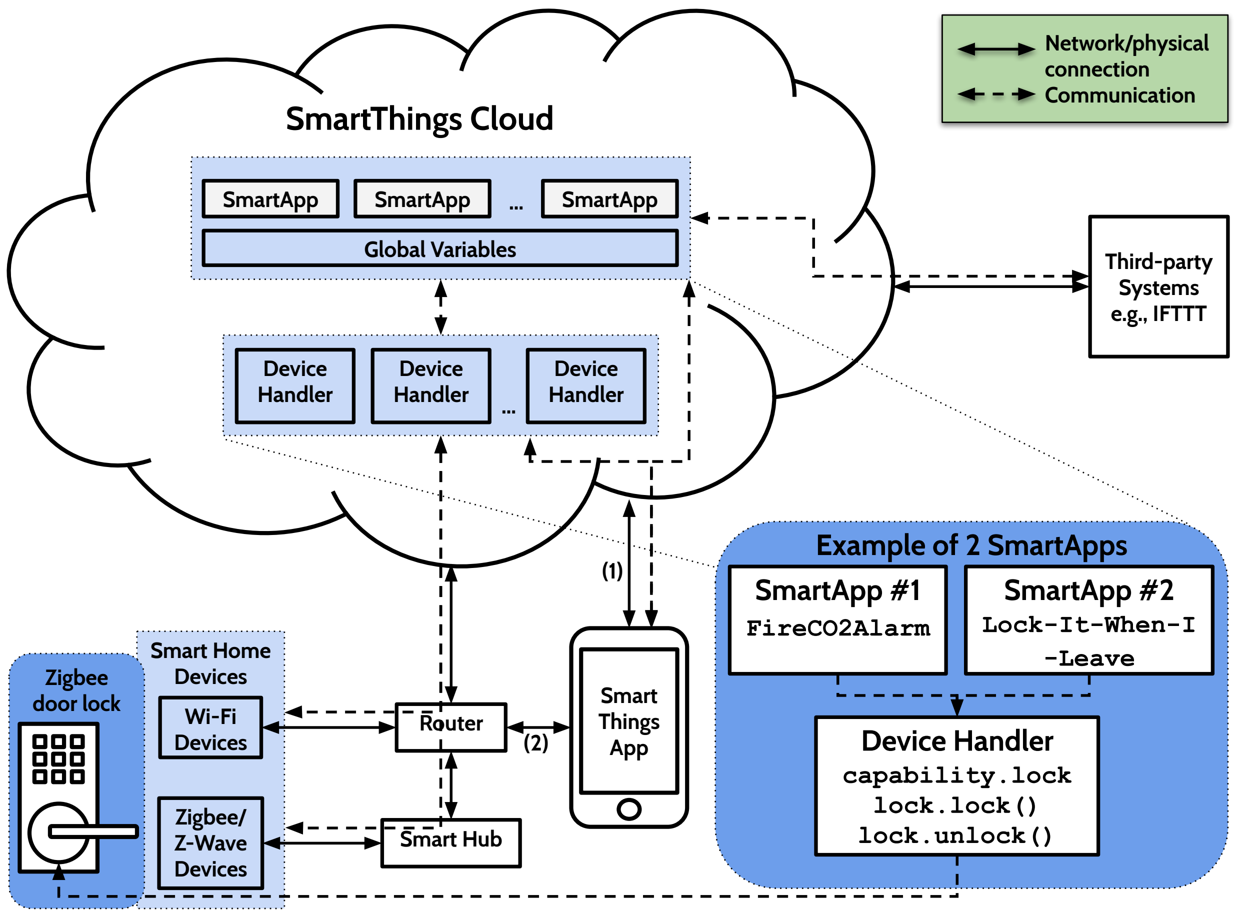


Figure 1.1: SmartThings platform that integrates devices and manages multiple apps.

Apps. Previous work discovered that the interactions between multiple apps in a smart home often create conflicts [120, 210, 163, 143, 215], which could be safety critical. We conducted a thorough study on the interactions and conflicts between smart home apps and developed a tool to automatically model-check these apps and report conflicts.

1.1 Overview of a Smart Home

We first begin by giving an overview of a smart home that consists of three components: devices, platforms, and apps. To illustrate, we use SmartThings [182], the de-facto smart home IoT development platform.

1.1.1 Components

Figure 1.1 shows an overview of a smart home that uses the SmartThings platform. There are three main components, as discussed shortly. The network/physical connections between these components are shown in Figure 1.1 as solid lines, while dashed lines represent communication paths.

(1) Smart Home Devices: SmartThings supports both SmartThings-branded and third-party devices as well as a variety of communication protocols, including Wi-Fi, Zigbee, and Z-Wave. While Wi-Fi devices are connected directly to the home router, Zigbee/Z-Wave devices are connected to a SmartThings smart hub through dedicated radios. The smart hub is connected to the home router and relays the communication between the Zigbee/Z-Wave devices and the SmartThings cloud via the router. Classes of devices that are supported by the SmartThings platform include both actuators (*e.g.*, switches, locks, thermostats, lights, or alarms) and sensors (*e.g.*, illuminance, motion, water, or sound sensors).

(2) SmartThings Platform: The SmartThings platform runtime and software components mainly run on the SmartThings cloud. The SmartThings cloud hosts smart home apps (*i.e.*, SmartApps) and device handlers (*i.e.*, drivers that directly control devices) developed using an event-based programming model in Groovy [95], a managed language running on top of the Java Virtual Machine (JVM). SmartApps implement desired functionalities on smart home devices by accessing global variables and device features through *capabilities* exposed by *device handlers*. For instance, a door lock can be accessed by SmartApps through its device handler that declares lock-related capabilities using `capability.lock`. These capabilities provide access to features such as *door-lock* and *door-unlock* via APIs such as `lock()` and `unlock()`. Third-party systems, *e.g.*, IFTTT (If-This-Then-That) [23], can also connect to the SmartThings cloud and control smart home devices through SmartApps that expose HTTP endpoints as a control interface.

(3) SmartThings Smartphone App: Homeowners can use the SmartThings smartphone app to install devices and SmartApps. To communicate with home devices, the smartphone first connects and sends control information to the SmartThings cloud either over the Internet or via the home router, illustrated by arrows (1) and (2) in Figure 1.1, and then the SmartThings cloud forwards the information to smart home devices via the home router and the smart hub.

1.1.2 Execution Model

SmartThings uses an event-driven execution model and allows multiple SmartApps to run concurrently. Consider for example the `FireCO2Alarm` app [162], which attempts to door-unlock if it detects smoke/fire through a smoke sensor. The app subscribes to the events generated by the sensor's device handler: when the sensor detects smoke/fire, it sends a message to the smart home hub. The smart home hub relays the message to the SmartThings cloud, which in turn runs the sensor's device handler to process the message. The device handler will generate an event and send it to the app's event handler method, which in turn calls another method `takeActions()` to door-unlock. Since multiple apps run concurrently, the two apps `FireCO2Alarm` and `Lock-It-When-I-Leave` share the device handler for the door lock, and thus can both execute `lock()` and `unlock()` at any time on the same device handler. The device handler on the cloud translates each action into device specific commands. The cloud then sends these commands to the local smart hub, which forwards the commands to the door lock.

1.2 Smart Home Devices

In this work, we first looked into smart home devices. Smart homes enable appliances to be controlled locally via the network and typically enable more sophisticated control systems [201]. Companies have launched a wide range of smart-home devices, many of which have serious security issues. A study reported vulnerabilities in 70% of the devices investigated [81]. Bugs have been found in a wide range of devices including routers [207, 209], smartcams [173, 75, 218], baby monitors [164, 178, 104], smart hubs [211], sprinklers [48], smart plugs [111], and smart fridges [15]. The problems in these systems are more basic than missing buffer checks—some of these devices have unsecured embedded web servers that allow anyone to update the firmware, have default passwords, use insecure authentication, or use clear text communications. Hence, these smart home devices have security and privacy problems.

Passive Inference Attacks. Modern smart home devices typically connect to the Internet via the home Wi-Fi router and can be controlled using a smartphone or voice assistant. Although most modern smart home devices encrypt their network traffic, recent work has demonstrated that the smart home is susceptible to passive inference attacks [41, 42, 43, 40, 132, 180, 179, 66, 28]. An eavesdropper may use characteristics of the network traffic generated by smart home devices to infer the device type and activity, and eventually user behavior. However, existing passive inference techniques have limitations. Most can only identify the device type and whether there is device activity (an event), but not the exact type of event or command [41, 42, 43, 40, 132, 180, 179]. Others only apply to a limited number of devices from a specific vendor [66], or need more information from other protocols (*e.g.*, Zigbee/Z-Wave) [28, 222] and the application source code [222]. Inference based on traffic volume analysis can be prevented by traffic shaping [28, 40]. Finally, many of these attacks assume that IP traffic is sniffed upstream from the home router, while the scenario where a local attacker sniffs encrypted Wi-Fi traffic has received less attention [96, 40].

We experiment with a diverse range of smart home devices, namely 19 popular Wi-Fi and Zigbee devices (12 of which are the most popular smart home devices on Amazon) from 16 popular vendors, including smart plugs, light bulbs, thermostats, home security systems, *etc.* During our analysis of the network traffic that these devices generate, we observed that events on smart home devices typically result in communication between the device, the smartphone, and the cloud servers that contains pairs of packets with predictable lengths. A packet pair typically consists of a *request* packet from a device/phone ("PING"), and a *reply* packet back to the device/phone ("PONG"). In most cases, the packet lengths are distinct for different device types and events, thus, can be used to infer the device and the specific type of event that occurred. Building on this observation, we were able to identify new *packet-level signatures* (or *signatures* for short) that consist only of the lengths and directions of a few packets in the smart home device traffic. These signatures: (1) can be extracted in an automated and systematic way without prior knowledge of the device's behavior; (2) can be used to infer fine-grained information, *e.g.*, event types; (3) correspond to a variety of different events (*e.g.*, "toggle ON/OFF" and "Intensity"/"Color"); and (4) have a number of advantages compared to prior (*e.g.*, statistical, volume-based) approaches.

Automated Extraction of Packet-Level Signatures. To further evaluate packet-level signatures, we created PingPong, a methodology and software tool that: (1) automates the extraction of packet-level signatures without prior knowledge about the device, and (2) detects signatures in network traces and real network traffic. For signature extraction, PingPong first generates training data by repeatedly triggering the event, for which a signature is desired, while capturing network traffic. Next, PingPong extracts request-reply packet pairs per flow ("PING-PONG"), clusters these pairs, and post-processes them to concatenate pairs into longer sequences where possible. Finally, PingPong selects sequences with frequencies close to the number of triggered events as the final signatures. The signature detection part of PingPong leverages the simplicity of packet-level signatures and is implemented using simple state machines. PingPong's implementation and datasets are made available at [194].

1.3 Smart Home Platforms

In addition to devices, we have also looked into smart home platforms. Part of the promise of smart home systems is the ability of collections of devices to work together to be smarter and more capable than individual devices. Achieving this requires integration between different devices, which may come from different manufacturers with entirely different software stacks, *e.g.*, Nest Thermostat, WeMo Switch, etc.

Modern smart home platforms support developers writing apps that implement useful functionality on smart devices. Significant efforts have been made to create integration platforms such as Android Things from Google [97], SmartThings from Samsung [182], and the open-source openHAB platform [154]. All of these platforms allow users to create *smart home apps* that integrate multiple devices and perform more complex routines, such as implementing a home security system.

Vulnerabilities. Smart home hubs support integration between these disparate devices, but existing hubs including SmartThings have serious security weaknesses. A lot of smart home devices do not have proper authentication mechanisms—they trust communication from the local area network. In the case of SmartThings, the system is a JVM-based system. It trusts the JVM, despite its bugs, to provide safety. Furthermore, the SmartThings system executes device drivers and applications on their cloud servers. Unfortunately, it gives excessive access to the cloud, and thus device drivers and applications.

Vigilia Approach. We developed Vigilia, a new cross-layer technique to harden smart home systems. Vigilia implements a lightweight approach to securing smart home systems at the network and operating system layers. This work leverages the observation that *most IoT devices are not general-purpose; they do not need to communicate with arbitrary machines and thus do not require full network access.* By enforcing access at the network level, Vigilia shifts the primary burden for security from individual devices to the network. The net effect is that system security no longer relies on every device manufacturer securing their devices and end users keeping devices patched—helping users secure IoT devices when manufacturers do not.

Vigilia makes contributions in the following aspects for smart home systems:

1) **Automatic Extraction of Enforcement of Security Policies:** It presents techniques that automatically extract and enforce fine-grained security policies on applications written using a programming model that is similar to SmartThings.
2) **Secure Enforcement Mechanism:** It uses a set of router-based techniques including modifications to the Wi-Fi stack that ensure that compromised devices cannot subvert the enforcement mechanisms by masquerading as the router or another device.
3) **No Spurious Failures:** It statically checks that programs will respect the policies at runtime and thus will never spuriously fail due to the security enforcement mechanisms.

Vigilia provides an open implementation of a smart home programming model that is similar to mainstream (close) platforms. We have made this implementation available at [196]. We have evaluated Vigilia on four smart home applications that control commercially available IoT devices. Our results demonstrate that Vigilia, among existing commercial and research systems, is the best at protecting these applications from various attacks with only minimal overhead.

1.4 Smart Home Apps

Aside from smart home devices and systems, we have also studied smart home apps. In this work, we focus on Samsung's SmartThings platform because it is the de-facto smart home development environment and has the most extensive collection of smart home apps, including those officially created by SmartThings [181] and those developed by third-party companies and hobbyists. Homeowners that use SmartThings can install any of these SmartApps and run them simultaneously in their home deployment. Many of these apps each implement a specific functionality, *e.g.*, turn off lights in the absence of motion. Thus, homeowners will likely need to install multiple apps that collectively achieve the desired functionality.

The presence of multiple apps that can control the same device creates interactions that can potentially be undesirable: conflicts. For example a homeowner may install the `FireCO2Alarm` [162] app which, upon the detection of smoke, sounds alarms and *door-unlocks*[2]. The same homeowner may also install the `Lock-It-When-I-Leave` [14] app to *door-lock* automatically when the homeowner leaves the house.

While it may appear that these apps can be safely installed together, closer examination reveals that they can interact in surprising ways. Consider the following scenario. If smoke is detected, `FireCO2Alarm` will door-unlock the door. If someone leaves home with the presence tag, this will make the presence sensor change its state from `"present"` to `"not present"`, causing the `Lock-It-When-I-Leave` app to door-lock the door. This defeats the intended purpose of the `FireCO2Alarm` app. Thus, the two apps *conflict*.

Data Races, Atomicity Violations. Interactions of smart home apps may initially appear similar to those of concurrent programs, including data races [64, 80, 127] and atomicity violations [135, 217, 91]. Data races can be resolved by acquiring locks appropriately, while atomicity violations can be resolved by ensuring that locks are held long enough to guarantee that a thread can finish all operations in a batch without interference from other threads.

Unfortunately, these techniques cannot resolve the above-mentioned conflict. Suppose that we use a lock to guarantee the atomicity of the critical region of the code—the `FireCO2Alarm` app needs to acquire the lock before triggering the alarm and holds the lock while the alarm is sounding. Similar actions need to be taken to door-lock and door-unlock for the `Lock-`

[2]We use *door-lock* and *door-unlock* to refer to actions on a physical door, and *lock* and *unlock* to refer to synchronizations in concurrent programming.

`It-When-I-Leave` app. However, this approach could disable the desirable functionality of the apps. To illustrate, consider a scenario in which the `Lock-It-When-I-Leave` app detects that someone leaves the house. It then acquires the lock before it enters the critical region in which door-lock is performed. It holds the lock to keep the door locked until the person returns. In this period, if the `FireCO2Alarm` app detects smoke/fire and attempts to door-unlock, it will fail because the `Lock-It-When-I-Leave` app holds the lock. We end up in the same situation: *the door is locked during a fire!*

Feature Interaction. Feature interaction considers the problem in which different software features can have negative interactions [56, 37, 112, 156, 39, 38]. Our setting differs from most of the previous work in this area in that smart home apps are developed independently and composed by end users. For example, SmartThings apps are distributed through many different channels (including pay for source). Thus, there does not exist a means to detect/resolve/avoid conflicts during development. Feature interactions have also been studied in research prototypes for home automation [120, 210, 163]. These early systems were prototype systems, and presumed much coarser apps (*e.g.*, a single app for lighting) than current smart home apps implement. HCI researchers have shown that feature interactions in IoT systems make it difficult for users to understand the systems' behavior [216]. In rule-based smart home systems, researchers have developed tools for repairing incorrect rules [147].

Interactions of Mobile Apps. Researchers have also studied interactions between Android apps [52, 198, 57, 101, 122, 45, 119]. However, these techniques focus primarily on cross-app information flow/taint analysis via ICC/IAC mechanisms in Android (*e.g.*, Intents) and thus cannot be used in our setting. In particular, our problem requires checking properties of the *execution trace* that such analyses cannot handle.

The Smart Home App Interaction Problem. The problem we focus on in this work is *conflict of expectations.* The expected result of the `Lock-It-When-I-Leave` app is that the door should be *locked* when the homeowner leaves, while the expected result for the `FireCO2Alarm` app is that the door should be *unlocked* during a fire. These expectations conflict in certain scenarios. Hence, the fundamental question here is *what should be the expected state of the door when these apps interact*: *locked* or *unlocked*? The potential conflict between the `FireCO2Alarm` and `Lock-It-When-I-Leave` apps is *not* correctable using standard mechanisms for *concurrent accesses to program variables or entities*—using locks to restore atomicity still violates *the integrity of the expected result.*

State-of-the-art and Our Work. The research community has been actively looking into smart home apps. There is a body of work that aims to find bugs and issues that could lead to serious security problems [85, 87, 61, 60, 49, 222, 188, 30]. However, none of these techniques focuses on interactions and conflicts between multiple apps. In the cyber-physical systems community, work has been done to identify and resolve conflicts between smart home apps at the system level, viewing apps as *black boxes* [204, 212, 203, 143, 215]. While such techniques are useful in certain simple scenarios, they are still semantics-agnostic and do not work even for the above-mentioned conflict—how can we automatically resolve the conflict without understanding the semantics of the apps, and their priority and timing requirements?

IA-Graph [123, 124] studies smart-home app conflicts and proposes a lightweight approach to check for conflicts. This work extracts an SMT formula that describes the legal transitions for an app and then uses an SMT solver to detect whether a set of apps has conflicting transitions. As acknowledged in the IA-Graph paper, IA-Graph "ignores complicated computations in the app code" and hence the patterns it finds are limited. In addition, not all transitions in an app can be expressed in SMT, further limiting the kinds of conflicts IA-Graph can detect. Another important drawback is IA-Graph does not check whether a conflicting transition is reachable in an execution and hence can produce many false positives.

Unfortunately, without access to their implementation, we could not conduct an empirical evaluation of these issues.

Implications. The implications of this work are two-fold. First, our study opens a new research direction in the area of testing and verification of concurrent programs where the development of different apps are done completely independently. The inability of existing concurrency control mechanisms to resolve smart home apps dictates the need of new techniques (such as IoTCheck) to detect and/or repair these conflicts. Second, for platform vendors such as Google and Samsung, new APIs should be designed and applied to these platforms so that app developers can be directed to make more informed decisions during development even if they are not aware of potential runtime conflicts.

1.5 Organization

The remainder of this dissertation is structured as follows:

- Chapter 2 further presents our findings on packet-level signatures. It also presents the design and evaluation of PingPong, a tool that can automatically extract and detect packet-level signatures in network traffic. Finally, it evaluates existing defenses that we can deploy against packet-level signatures.
- Chapter 3 discusses vulnerabilities in smart home platforms and presents Vigilia, a smart home platform similar to SmartThings, that implements techniques to harden smart home platforms.
- Chapter 4 presents our in-depth study of smart home apps and their interactions that may lead to conflicts. It also presents IoTCheck, a tool developed to automatically detect conflicts between smart home apps through model checking.

- Chapter 5 discusses related work.
- Finally, Chapter 6 concludes, and discusses limitations and future work.

Chapter 2

Packet-Level Signatures for Smart Home Devices

In this chapter, we first present the new packet-level signatures for smart home devices. Next, we present a set of techniques we devised to extract packet-level signatures for smart home devices, the PingPong tool that implements these techniques, and the evaluation of the tool. Finally, we discuss possible defenses against packet-level signatures.

2.1 New Packet-Level Signatures

We begin by presenting packet-level signatures. We discover new IoT device signatures that are simple and intuitive: they consist of short sequences of (typically 2-6) packets of specific lengths, exchanged between the device, the smartphone, and the cloud. The signatures are effective:

1) They detect event occurrences with an average recall of more than 97%, surpassing the state-of-the-art techniques (see Chapter 5 and Section 2.4.2).
2) They are unique: we observe a low false positive rate (FPR), namely 1 false positive (FP) per 40 million packets in network traces with hundreds of millions of packets (see

Section 2.4.3).

3) They characterize a wide range of devices: (i) we extract signatures for 18 out of the 19 devices we experimented with, including the most popular home security devices such as the Ring Alarm Home Security System and Arlo Q Camera (see Section 2.4.1); (ii) we extract signatures for 21 additional devices from a public dataset [166], including more complex devices, *e.g.*, voice-command devices, smart TVs, and even a fridge (see Section 2.4.6).
4) They are robust across a diverse range of settings: (i) we extract signatures both from testbed experiments and publicly available datasets; and (ii) we trigger events in different ways, *i.e.*, using both a local and a remote smartphone, and a home automation system.
5) They can be extracted from both unencrypted and encrypted traffic.
6) They allow quick detection of events as they rely only on a few packet lengths and directions, and do not require any statistical computation.

2.2 Problem Setup

In this chapter, we first present our threat model. Then, we present the smart home environment and the passive inference attacks we consider. We also discuss a key insight we obtained from manually analyzing network traffic from the simplest devices—smart plugs. The packet sequences we observed in smart plugs inspired the PingPong methodology for automatically extracting signatures.

2.2.1 Threat Model

We are concerned with the network traffic of smart home devices leaking private information about smart home devices and users. Although most smart home devices encrypt their communication, information can be leaked by traffic metadata such as the lengths and directions of these encrypted packets.

We consider two different types of adversaries: a *WAN sniffer* and a *Wi-Fi sniffer*. The adversaries differ in terms of the vantage point where traffic is inspected and, thus, what information is available to the adversary. The WAN sniffer monitors network traffic in the communication between the home router and the ISP network (or beyond) [41, 42, 43, 40]. This adversary can inspect the IP headers of all packets, but does not know the device MAC addresses to identify which device has sent the traffic. We assume a standard home network that uses NAT: all traffic from the home is multiplexed onto the router's IP address. Examples of such adversaries include intelligence agencies and ISPs. The Wi-Fi sniffer monitors encrypted IEEE 802.11 traffic, and has not been as widely studied [96, 40]. We assume that the Wi-Fi sniffer does not know the WPA2 key, and thus only has access to the information sent in clear text—the MAC addresses, packet lengths, and timing information. As packets are encrypted, the Wi-Fi sniffer does not have access to network and transport layer information.

For both adversaries, we assume that the adversary knows the type of the smart home device that they wish to target and passively monitor. Thus, they can train the system on another device of the same type offline, extract the signature of the device, and perform the detection of the signature on the traffic coming from the smart home they target. We assume that the devices encrypt their communication and thus neither adversary has access to the clear-text communication.

2.2.2 Smart Home Environment and Experimental Testbed

Experimental Testbed. Figure 2.1 depicts our experimental setup, which resembles a typical smart home environment. We experiment with 19 widely-used smart home devices from 16 different vendors (see Table 2.1). We attempted to select a set of devices with a wide range of functionality—from plugs to cameras. They are also widely used: these devices are popular and they come from well-known vendors. The first 12 (highlighted in green) are

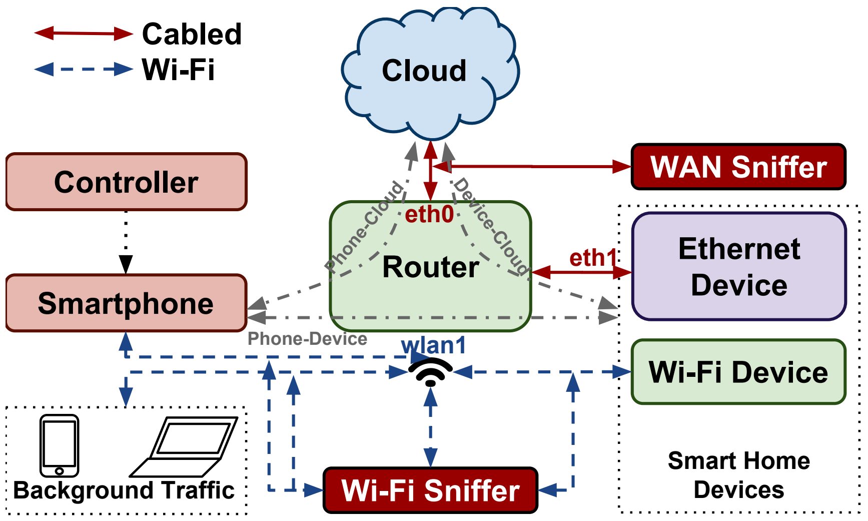


Figure 2.1: Our experimental setup for studying smart home devices. "Wi-Fi Device" is any smart home device connected to the router via Wi-Fi (*e.g.*, Amazon and WeMo plugs). "Ethernet Device" is any smart home device connected to the router via Ethernet (*e.g.*, SmartThings hub that relays the communication of Zigbee devices). Smart home device events may result in communication between Phone-Cloud, Device-Cloud, or Phone-Device. There may also be background traffic from additional computing devices in the home.

the most popular on Amazon [33]: (1) each received the most reviews for its respective device type and (2) each had at least a 3.5-star rating—they are both popular and of high quality (*e.g.*, the Nest T3007ES and Ecobee3 thermostats are the two most-reviewed with 4-star rating for thermostats). Some devices are connected to the router via Wi-Fi (*e.g.*, the Amazon plug) and others through Ethernet. The latter includes the SmartThings, Sengled, and Hue hubs that relay communication to/from Zigbee/Z-Wave devices: the SmartThings plug, Kwikset doorlock, Sengled light bulb, and Hue light bulb.

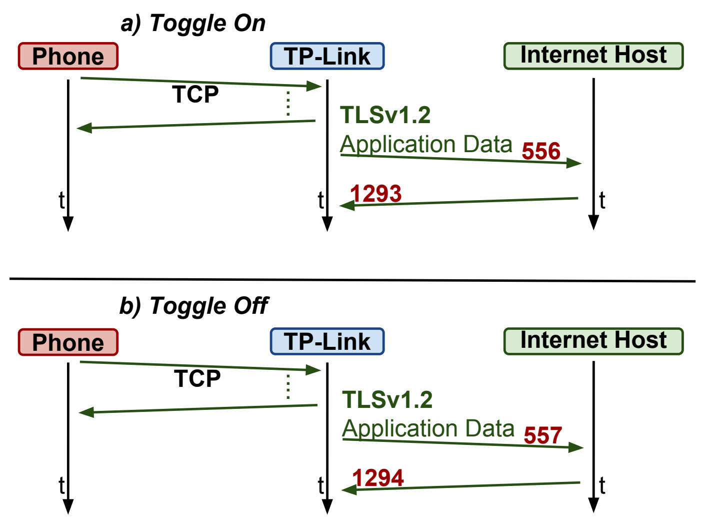
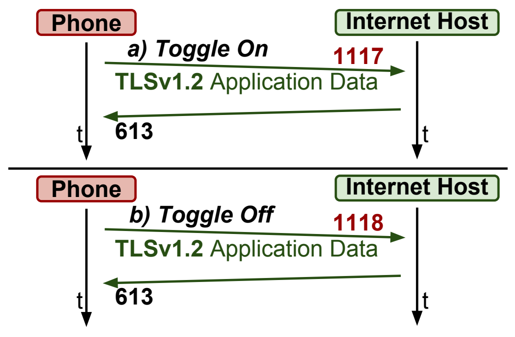
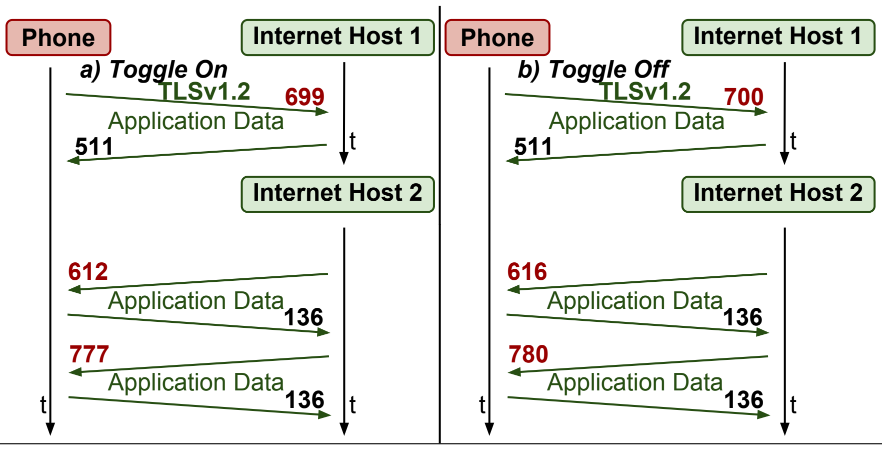
Figure 2.1 shows that smart home devices are controlled from the smartphone using its vendor's official Android application. The smartphone is connected to a local network, which the devices are also connected to. When the smartphone is connected to a remote network, only the Device-Cloud communication is observable in the local network—the smartphone controls a device by communicating with its vendor-specific cloud, and the cloud relays the command to the device. The controller represents the agent that operates the smartphone to control the smart home device of interest. This may be done manually by a human

Table 2.1: The set of smart home devices considered in this paper. Devices highlighted in green are among the most popular on Amazon.

No.	Device Name	Model Details
1.	Amazon plug	Amazon Smart Plug
2.	WeMo plug	Belkin WeMo Switch
3.	WeMo Insight plug	Belkin WeMo Insight Switch
4.	Sengled light bulb	Sengled Element Classic
5.	Hue light bulb	Philips Hue white
6.	LiFX light bulb	LiFX A19
7.	Nest thermostat	Nest T3007ES
8.	Ecobee thermostat	Ecobee3
9.	Rachio sprinkler	Rachio Smart Sprinkler Controller Generation 2
10.	Arlo camera	Arlo Q
11.	Roomba robot	iRobot Roomba 690
12.	Ring alarm	Ring Alarm Home Security System
13.	TP-Link plug	TP-Link HS-110
14.	D-Link plug	D-Link DSP-W215
15.	D-Link siren	D-Link DCH-S220
16.	TP-Link light bulb	TP-Link LB-130
17.	SmartThings plug	Samsung SmartThings Outlet (2016 model)
18.	Kwikset lock	Kwikset SmartCode 910
19.	Blossom sprinkler	Blossom 7 Smart Watering Controller

(as in Section 2.2.3) or through software (as in Section 2.3). Additionally, there are other computing devices (*e.g.*, laptops, tablets, phones) in the house that also generate network traffic, which we refer to as "Background Traffic". The router runs OpenWrt/LEDE [155], a Linux-based OS for network devices, and serves as our vantage point for collecting traffic for experiments. We run `tcpdump` on the router's WAN interface (`eth0`) and local interfaces (`wlan1` and `eth1`) to capture Internet traffic as well as local traffic for all Wi-Fi and Ethernet devices. We use the testbed to generate training data for each device, from which we in turn extract signatures (Section 2.4.1). In Section 2.4.2, the same testbed is used for testing, *i.e.*,

Table 2.2: Packet-level signatures of TP-Link, D-Link, and SmartThings smart plugs observable by the WAN sniffer. The numbers represent packet lengths, with red indicating that the length is different for ON vs. OFF, and the arrows represent packet directions.

TP-Link	D-Link
 a) Toggle On Phone TP-Link Internet Host TCP TLSv1.2 Application Data 556 1293 t b) Toggle Off Phone TP-Link Internet Host TCP TLSv1.2 Application Data 557 1294 t	 Phone Internet Host a) Toggle On 1117 TLSv1.2 Application Data 613 t Phone Internet Host b) Toggle Off 1118 TLSv1.2 Application Data 613 t
SmartThings	
 Phone Internet Host 1 a) Toggle On TLSv1.2 699 Application Data 511 Internet Host 2 612 Application Data 136 777 Application Data 136 t Phone Internet Host 1 b) Toggle Off TLSv1.2 700 Application Data 511 Internet Host 2 616 Application Data 136 780 Application Data 136 t	

to detect the presence of the extracted signatures in traffic generated by all the devices as well as by other computing devices (background traffic).

Communication. Smart home device events may result in communication between three different pairs of devices, as depicted in Figure 2.1: (1) the smartphone and the smart home device (*Phone-Device*); (2) the smart home device and an Internet host (*Device-Cloud*), and (3) the smartphone and an Internet host (*Phone-Cloud*). The idea behind a passive inference attack is that network traffic on any of these three communication paths may contain unique traffic signatures that can be exploited to infer the occurrence of events.

2.2.3 Motivating Case: Smart Plugs

As an illustrative example, let us discuss our manual analysis of 3 smart plugs: the TP-Link plug, the D-Link plug, and the SmartThings plug. Data for the manual analysis was collected using the setup in Figure 2.1. For each device, we toggled it ON, waited for approximately one minute, and then toggled it OFF. This procedure was repeated for a total of 3 ON and 3 OFF events, separated by one minute in between. Timestamps were manually noted for each event. The PCAP files logged at the router were analyzed using a combination of scripts and manual inspection in Wireshark.

New Observation: Packet Pairs. We identified the traffic flows that occurred immediately after each event and observed that certain pairs of packets with specific lengths and directions followed each ON/OFF event: the same pairs consistently showed up for all events of the same type (*e.g.*, ON), but were slightly different across event types (ON vs. OFF). The pairs were comprised of a *request* packet in one direction, and a *reply* packet in the opposite direction. Intuitively, this makes sense: if the smart home device changes state, this information needs to be sent to (request), and acknowledged by (reply), the cloud server to enable devices that are not connected to the home network to query the smart home device's current state. These exchanges resemble the ball that moves back and forth between players in a game of pingpong, which inspired the name for our software tool.

Table 2.2 illustrates the observed packet exchanges. For the TP-Link plug, we observed an exchange of 2 TLS Application Data packets between the plug and an Internet host where the packet lengths were 556 and 1293 when the plug was toggled ON, but 557 and 1294 for

OFF. We did not observe any pattern in the D-Link plug's own communication. However, for ON events, the controlling smartphone would always send a request packet of length 1117 to an Internet host and receive a reply packet of length 613. For OFF, these packets were of lengths 1118 and 613, respectively. Similarly for the SmartThings plug, we found consistently occurring packet pairs in the smartphone's communication with two different Internet hosts where the lengths of the request packets were different for ON and OFF events. Thus, this request-reply pattern can occur in the communication of any of the three pairs: Phone-Device, Device-Cloud, or Phone-Cloud (see Figure 2.1).

Key Insight. This preliminary analysis indicates that each type of event is uniquely identified by the exchange of pairs (or longer sequences) of packets of specific lengths. To the best of our knowledge, this type of network signature has not been observed before, and we refer to it as a *packet-level signature*.

2.3 PingPong Design

The key insight obtained from our manual analysis in Section 2.2.3 was that unique sequences of packet lengths (for packet pairs or longer packet sequences) typically follow simple events (*e.g.*, ON vs. OFF) on smart plugs, and can potentially be exploited as signatures to infer these events. This observation motivated us to investigate whether: (1) more smart home devices, and potentially the smartphones that control them as well, exhibit their own unique packet-level sequences following an event, (2) these signatures can be learned and automatically extracted, and (3) they are sufficiently unique to accurately detect events. In this section, we present the design of PingPong—a system that addresses the above questions with a resounding YES!

PingPong automates the collection of training data, extraction of packet-level signatures, and detection of the occurrence of a signature in a network trace. PingPong has two components:

(1) training (Section 2.3.1), and (2) detection (Section 2.3.2). Figure 2.2 shows the building blocks and flow of PingPong on the left-hand side, and the TP-Link plug as an example on the right-hand side. We use the TP-Link plug as a running example throughout this section.

2.3.1 Training

The training component is responsible for the extraction of packet-level signatures for a device the attacker wants to profile and attack. It consists of 5 steps (see Figure 2.2).

Data Collection. The first step towards signature generation is to collect a *training set* for the device. A training set is a network trace (a PCAP file) that contains the network traffic generated by the device and smartphone as a result of events; this trace is accompanied by a text file that contains the set of event timestamps.

PingPong partially automates training set collection by providing a shell script that uses the Android Debug Bridge (`adb`) [35] to issue touch inputs on the smartphone's screen. The script is run on a laptop that acts as the controller in Figure 2.1. The script is tailored to issue the sequence of touch events corresponding to the events for which a training set is to be generated. For example, if a training set is desired for a smart plug's ON and OFF events, the script issues a touch event at the screen coordinates that correspond to the respective buttons in the user interface of the plug's official Android app. As device vendors may choose arbitrary positions for the buttons in their respective Android applications, and since the feature sets differ from device to device, the script must be manually modified for the given device. The script issues the touch sequence corresponding to each specific event n times, each separated by m seconds.[1] The results reported in this paper use $n = 50$ or $n = 100$ depending on the event type (see Section 2.4.1).

[1] We selected $m = 131$ seconds to allow sufficient time such that there is no overlap between events. Section 2.4.7 provides more explanation for this choice with respect to other parameters.

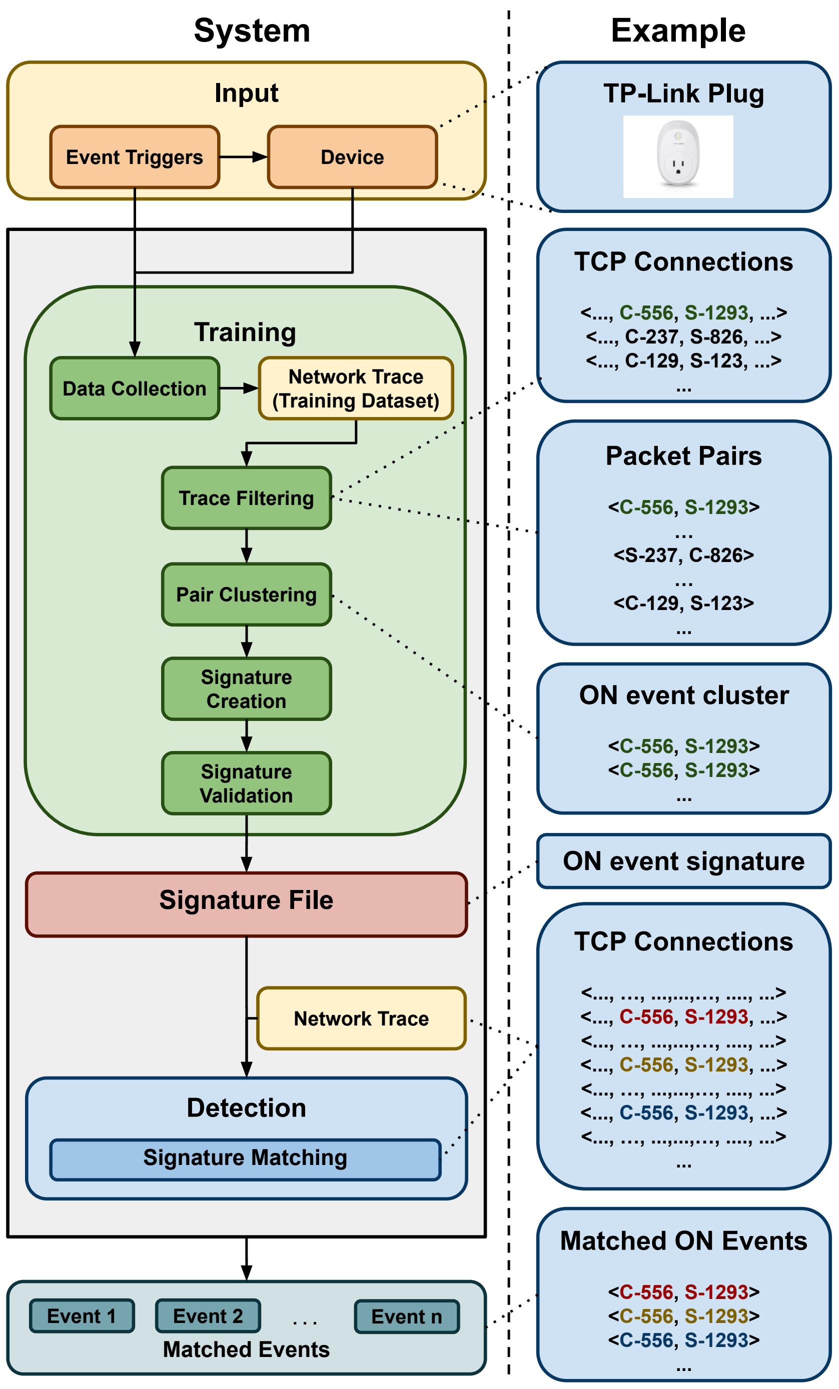


Figure 2.2: Left: PingPong Overview. Right: TP-Link plug is used as a running example throughout this section.

The script also outputs the current timestamp to a file on the laptop when it issues an event.

To collect a training set, we do the following: (1) start `tcpdump` on the router's interfaces;

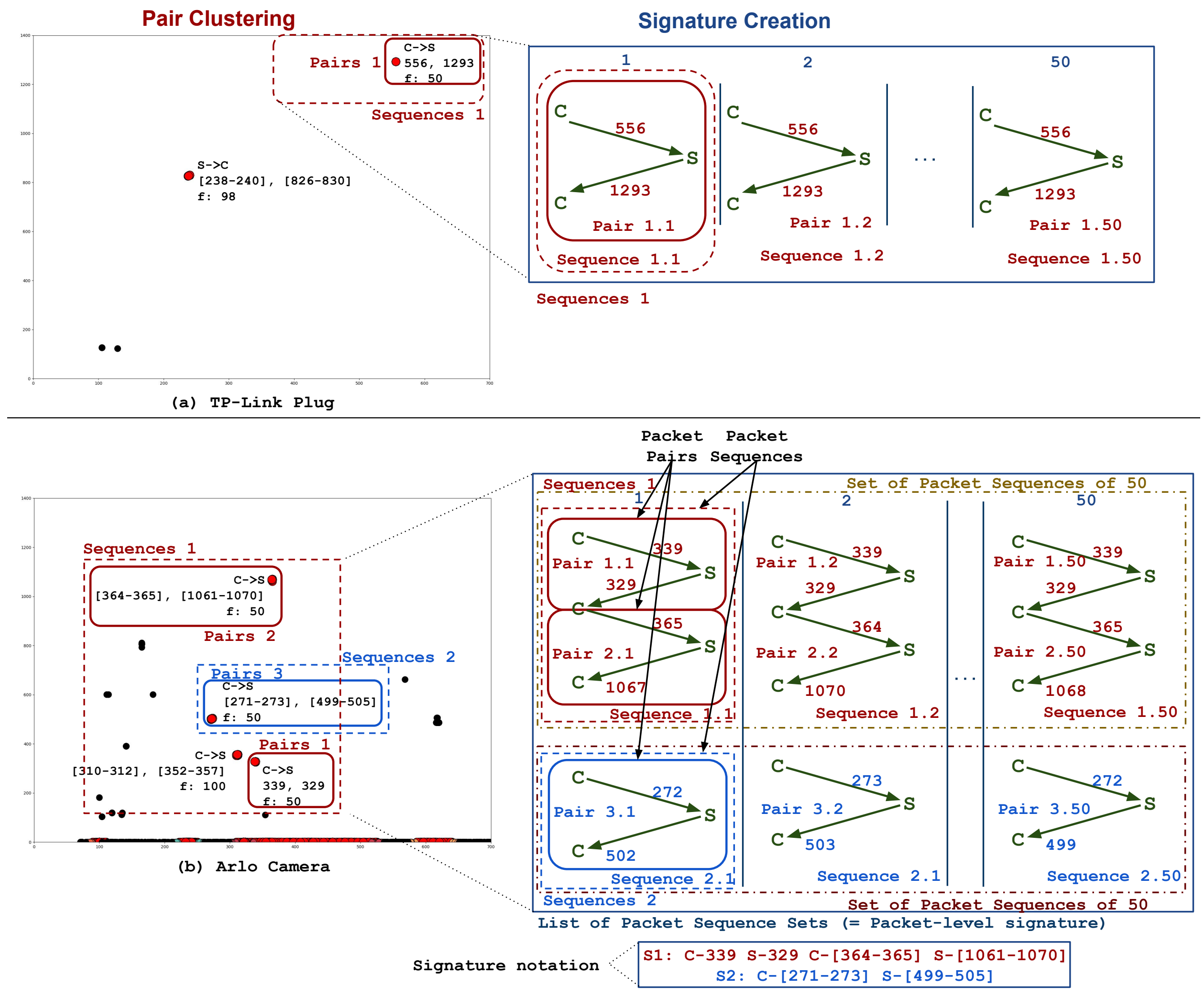


Figure 2.3: Pair clustering and signature creation for 2 extreme cases—TP-Link plug has the simplest signature with only 1 pair (see our initial findings in Table 2.2). The Arlo camera has a more complex signature with 1 sequence of 2 pairs and 1 sequence of 1 pair. The left subfigure, in every row, depicts the packet lengths in one packet pair (P_{c_1}, P_{c_2}). Notation: `C->S` means a pair where the first packet's direction is Client-to-Server and the second packet's direction is server-to-client, and vice versa for `S->C`; `f: 50` means that the pair appears in the clustering with a *frequency of 50*; `Signature notation` shows a summary of 2 sets of 50 instances of packet sequences. Example: `C->S 556, 1293 f: 50` means that the pair of packets with lengths 556 (client-to-server) and 1293 (server-to-client) appear 50 times in the cluster.

(2) start the script; (3) terminate `tcpdump` after the n-th event has been issued. This leaves us with a set of PCAP files and event timestamps, which constitute our raw training set.

We base our signature generation on the traces collected from the router's local interfaces as they are the vantage points that provide the most comprehensive information: they include both local traffic and Internet traffic. This allows PingPong to exhaustively analyze all network packets generated in the communications between the device, smartphone, and Internet hosts on a per device basis. As signatures are based entirely on packet lengths and directions, signatures present in Internet traffic (*i.e.*, Device-Cloud and Phone-Cloud traffic) are applicable on the WAN side of the router, despite being extracted from traces captured within the local network.

Trace Filtering. Next, PingPong filters the collected raw training set to discard traffic that is unrelated to a user's operation of a smart home device. All packets, where neither the source nor destination IP matches that of the device or the controlling smartphone, are dropped. Additionally, all packets that do not lie within a time window t after each timestamped event are discarded. We selected $t = 15$ seconds to allow sufficient time for all network traffic related to the event to complete. Our sensitivity study also confirmed that this was a conservative choice (see Section 2.4.7).

PingPong next reassembles all TCP connections in the filtered trace. Given the set of reassembled TCP connections, we now turn our attention to the packets P that carry the TCP payload. For TLS connections, P is limited further to only be the subset of packets that are labeled as "Application Data" in the unencrypted TLS record header [168]. By only considering packets in P, we ensure that the inherently unpredictable control packets (*e.g.*, TCP ACKs and TLS key negotiation) do not become part of the signature as P only contains packets with application layer payload.

We next construct the set P' by forming *packet pairs* from the packets in P (see Definition 2.1). This is motivated by the following observation: the deterministic sequence of packets that make up packet-level signatures often stem from a *request-reply exchange* between the device, smartphones, and some Internet hosts (see Section 2.2.3). Furthermore, since a packet pair is the simplest possible pattern, and since longer patterns (*i.e.*, packet sequences—see Definition 2.2) can be reconstructed from packet pairs, we look for these packet pairs in the training set. For the TP-Link plug example in Figure 2.2, PingPong reassembles `<..., C-556, S-1293, ...>`, `<..., C-237, S-826, ...>`, *etc.* as TCP connections. Then, PingPong extracts `<C-556, S-1293>`, `<C-237, S-826>`, *etc.* as packet pairs.

Definition 2.1. ***Packet Pair.*** *Let P_c be the ordered set of packets with TCP payload that belong to TCP connection c, let P_{c_i} denote the i-th packet in P_c, and let C and S each denote client-to-server and server-to-client packet directions, respectively, where a client is a smartphone or a device. A packet pair p is then $p = (C{-}P_{c_i},\ S{-}P_{c_{i+1}})$ or $p = (S{-}P_{c_i},\ C{-}P_{c_{i+1}})$* iff *$P_{c_i}$ and $P_{c_{i+1}}$ go in opposite directions. Otherwise, if P_{c_i} and $P_{c_{i+1}}$ go in the same direction, or if P_{c_i} is the last packet in P_c, the packet pair $p = (C{-}P_{c_i}, \text{nil})$ or $p = (S{-}P_{c_i}, \text{nil})$ is formed, and packet $P_{c_{i+1}}$, if any, is paired with packet $P_{c_{i+2}}$.*

Pair Clustering. After forming a set of packet pairs, relevant packet pairs (*i.e.*, those that consistently occur after an event) must next be separated from irrelevant ones. This selection also needs to take into account that the potentially relevant packet pairs may have slight variations in lengths. Since we do not know in advance the packet lengths in the pairs, we use an unsupervised learning algorithm: DBSCAN [82].

DBSCAN is provided with a distance function for comparing the similarity of two packet pairs, say p_1 and p_2. The distance is maximal if the packet directions are different, *e.g.*, if p_1 is comprised of a packet going from a local device to an Internet host followed by a packet going from an Internet host to a local device, while p_2 is comprised of a packet going from an Internet host to a local device followed by a packet going from a local device to an Internet host. If the packet directions match, the distance is simply the Euclidean distance between the two pairs, *i.e.*, $\sqrt{(p_1^1 - p_2^1)^2 + (p_1^2 - p_2^2)^2}$, where p_j^i refers to the packet length of the i-th element of pair j. DBSCAN's parameters are ϵ and `minPts`, which specify the neighborhood radius to consider when determining core points and the minimum number of points in that neighborhood for a point to become a core point, respectively. We choose $\epsilon = 10$ and $\texttt{minPts} = \lfloor n - 0.1n \rfloor$, where n is the total number of events. We allow a slack of $0.1n$ to `minPts` to take into account that event-related traffic could occasionally have missing pairs, for example caused by the phone app not responding to some of the automated events. We report the study of PingPong parameter values in Section 2.4.7.

Figure 2.3(a) illustrates the pair clustering process for TP-Link plug. There are 50 ON and 50 OFF actions, and there must be at least 45 ($n = 50 \implies \texttt{minPts} = \lfloor 50 - 0.1 \times 50 \rfloor = 45$) similar packet pairs to form a cluster. Two clusters are formed among the data points, *i.e.*, those with frequencies `f: 50` and `f: 98`, respectively. Since these two clusters contain similar packet pairs that occur during t, this indicates with high confidence that the packets are related to the event.

Signature Creation. Given the output produced by DBSCAN, PingPong next drops all clusters whose frequencies are not in the interval $[\lfloor n - 0.1n \rfloor, \lceil n + 0.1n \rceil]$ in order to only include in the signature those clusters whose frequencies align closely with the number of events n. Intuitively, this step is to deal with *chatty devices*, namely devices that communicate continuously/periodically while not generating events. Consequently, PingPong only picks the cluster `Pairs 1` with frequency 50 for the TP-Link plug example in Figure 2.3 as a signature candidate since 50 is in $[\lfloor n - 0.1n \rfloor, \lceil n + 0.1n \rceil] = [45, 55]$ when $n = 50$, whereas 98 is not. As a pair from this cluster occurs exactly once during t, there is high confidence that the pair is related to the event.

PingPong next attempts to concatenate packet pairs in the clusters so as to reassemble the longest *packet sequences* possible (see Definition 2.2), which increases the odds that a signature is unique. Naturally, packet pair concatenation is only performed when a device has more than one cluster. This is the case for the Arlo camera, but not the TP-Link plug. Packet pairs in clusters x and y are concatenated *iff* for each packet pair p_x in x, there exists a packet pair p_y in y such that p_x and p_y occurred consecutively in the same TCP connection. If there are more pairs in y than in x, the extra pairs of y are simply dropped. The result is referred to as a *set of packet sequences* (see Definition 2.3) and is considered for further concatenation with other clusters if possible.

Definition 2.2. ***Packet Sequence.*** *A packet sequence s is formed by joining packet pairs p_1 and p_2* iff *p_1 and p_2 are both in P_c (same TCP connection)* and *the packets in p_1 occur immediately before the packets in p_2 in P_c. Note that the packet sequence s resulting from joining p_1 and p_2 can be of length 2, 3, or 4, depending on whether or not the second element of p_1 and/or p_2 is nil.*

Definition 2.3. ***Set of Packet Sequences.*** *A set of packet sequences S is a set of similar packet sequences. Two packet sequences s_1 and s_2 are similar and thus belong to the same set S* iff *they (1) contain the same number of packets, (2) the packets at corresponding indices of s_1 and s_2 go in the same direction, and (3) the Euclidean distance between the packet lengths at corresponding indices of s_1 and s_2 is below a threshold—packet lengths in packet sequences inherit the slight variations that stem from packet pairs.*

Figure 2.3(b) shows how pair clustering produces 3 clusters around the pairs `<C-339, S-329>` (*i.e.*, cluster `Pairs 1`), `<C-[364-365], S-[1061-1070]>` (*i.e.*, cluster `Pairs 2`), and `<C-[271-273], S-[499-505]>` (*i.e.*, cluster `Pairs 3`) for the Arlo camera. The notation `C-`$[l_1 - l_2]$ or `S-`$[l_1 - l_2]$ indicates that the packet length may vary in the range between l_1 and l_2. Each pair from cluster `Pairs 1` and each pair from cluster `Pairs 2` are then concatenated into a sequence in `Sequences 1` (a set of packet sequences) as they appear consecutively in the same TCP connection, *i.e.*, `Pair 1.1` with `Pair 2.1`, `Pair 1.2` with `Pair 2.2`, ..., `Pair 1.50` with `Pair 2.50`. The cluster `Pairs 3` is finalized as the set `Sequences 2` as its members appear in different TCP connections than the members of `Sequences 1`. Thus, the initial 3 clusters of packet pairs are reduced to 2 sets of packet sequences. For the TP-Link plug, no concatenation is performed since there is only a single cluster, `Pairs 1`, which is finalized as the set `Sequences 1`.

Finally, PingPong sorts the sets of packet sequences based on the timing of the sets' members to form a *list of packet sequence sets* (see Definition 2.4). For example, for the Arlo camera, this step produces a list in which the set `Sequences 1` precedes the set `Sequences 2` because there is always a packet sequence in `Sequences 1` that precedes a packet sequence in `Sequences 2`. The purpose of this step is to make the temporal order of the sets of packet sequences part of the final signature. If no such order can be established, the set with the shorter packet sequences is discarded. Manual inspection of some devices suggests that the earlier sequence will often be the control command sent from an Internet host followed by the device's acknowledgment of the command, while the later sequence will stem from the device initiating communication with some other Internet host to inform that host about its change in status.

Definition 2.4. ***List of Packet Sequence Sets.*** *A list of packet sequence sets is a list that contains sets of packet sequences that are sorted based on the occurrence of the set members in time. Set S_x goes before set S_y* iff *for each sequence s_x in S_x, there exists a sequence s_y in S_y that occurred after s_x within t.*

Signature Validation. Before finalizing the signature, we validate it by running the detection algorithm (see Section 2.3.2) against the raw training set that was used to generate the signature. If PingPong detects at most n events, and the timestamps of detected events match the timestamps for events recorded during training, the signature is finalized as a valid *packet-level signature* (see Definition 2.5) and stored in a signature file. A signature can fail this check if it detects more events than the actual number of events in the training set (*i.e.*, false positives). This can happen if the packet sequences in the signature frequently appear outside t.

Definition 2.5. ***Packet-level Signature.*** *A packet-level signature is then a list of packet sequence sets that has been validated and finalized.*

Signature File. A signature file stores a packet-level signature. Figure 2.3 shows that the TP-Link plug signature consists of 50 instances of packet sequences in set `Sequences 1`, but only one instance will be used during detection since all 50 are identical. Figure 2.3(b) shows the signature file (on the right-hand side) for the Arlo camera. It is a list that orders the two sets of packet sequences, `Sequences 1` and `Sequences 2`. `Sequences 1` is comprised of 50 packet sequences, each comprised of two packet pairs. `Sequences 2` is comprised of another 50 packet sequences, each comprised of a single packet pair. Since the sequences vary slightly in each set, all unique variations are considered during detection.

2.3.2 Detection

PingPong's detection component determines if a packet-level signature is present in a captured network trace, or real-time network traffic. The implementation differs slightly for the two adversaries described in Section 2.2.1.

Layer-2 Information. The Wi-Fi sniffer has the disadvantage of only having access to layer-2 header information due to WPA2 encryption. This means that it cannot reconstruct TCP connections, but can only separate traffic into flows of packets exchanged between pairs of MAC addresses, referred to as *layer-2 flows.* On layer 2, the encryption added by WPA2 does not pad packet lengths. Thus, our signatures, extracted from TCP/IP traffic, can be directly mapped to layer 2 if the IEEE 802.11 radiotap header, frame header, the AES-

CCMP IV and key identifier, and FCS are accounted for. In our testbed, these consistently add 80 bytes to the packet length.

Packet Sequence Matching. A state machine is maintained for each packet sequence of the signature for each flow, *i.e.*, TCP connection for the WAN sniffer or layer-2 flow for the Wi-Fi sniffer. Each packet in the stream of packets is presented to the state machines associated with the flow that the packet pertains to. If the packet's length and direction match that of the packet for the next state, the state machine advances and records the packet. The detection algorithm operates differently for layer-2 and layer-3 detections when the packet does not match the expected next packet. For layer-2 detection (Wi-Fi sniffer), the packet is simply ignored, and the state machine remains in the same state. Out-of-order packets can in theory cause failures for our current layer-2 signature matching implementation. In practice, out-of-order packets occur rarely enough that they are unlikely to be a significant issue. There are other mitigating factors that further lower the likelihood of out-of-order packets posing a problem. Signatures that strictly follow the pingpong pattern do not provide an opportunity for their packets to be reordered. Out-of-order packets often result in TCP ACKs in the other direction that would cause retransmissions. Thus, another event packet would be seen right after the first one and the first one would be ignored. For layer-3 detection (WAN sniffer), the packet causes the state machine to discard any partial match—layer-3 detection does not deal with interleaving packets as it considers individual TCP connections and can filter out TCP retransmissions. When a state machine matches its first packet and advances to the next state, a new state machine is created for the same packet sequence, but in the initial state. This is to ensure that the state machine starts at the correct first packet, *e.g.*, when a packet of that length appears in other traffic. To bound the number of active state machines, and to minimize the number of false positives resulting from retransmissions, any state machine that advances from state s to state $s+1$ replaces any existing state machine in state $s+1$ *iff* the last packet of the newly advanced state machine has a later timestamp than that of the existing state machine. Once a state machine reaches its terminal state, the set of recorded packets is reported as a sequence match.

Matching Strategies. For every packet in a sequence, there are two possible matching strategies: *exact* and *range-based* matching. In exact matching, the state machines only consider exactly those packet lengths that were observed during training as valid. In the range-based matching strategy, the state machines allow the packet lengths to lie between the minimum and maximum packet lengths (plus a small delta) observed during training. As such, range-based matching attempts to accommodate packet sequences that have slight variations where all permutations may not have been observed during training. For range-based matching, the lower and upper bounds for each packet of a packet sequence are derived from the core points of the packet pair clustering (see Section 2.3.1). ϵ is then applied to these bounds analogous to the clustering technique used in the DBSCAN algorithm—we, therefore, consistently use the same $\epsilon = 10$. For example, for $\epsilon = 10$ and core points `<C-338, S-541>` and `<C-339, S-542>`, a state machine that uses range-based matching will consider client-to-server packets with lengths in `[328, 349]`, and server-to-client packets with lengths in `[531,552]` as valid.

Exact matching is used when no variations in packet lengths were observed during training, and range-based matching is used if variations in packet lengths were observed during training. However, range-based matching is not performed when the signature only consists of 2 packets and/or there is an overlap between the signatures that represent different types of events (*e.g.*, the D-Link plug's signatures for ON and OFF in Table 2.3) as range-based matching for 2-packet signatures has a high risk of generating many false positives.

Declaring a Signature Match. A sequence match does not necessarily mean that the full signature has been matched. Some signatures are comprised of multiple packet sequences and all of them have to be matched (*e.g.*, Arlo camera, see Section 2.3.1). Sequence matches are therefore reported to a secondary module that verifies if the required temporal constraints are in place, namely that the sequence match for packet sequence set i occurs *before* the sequence match for packet sequence set $i+1$ *and* that the time between the first packet of the sequence match corresponding to packet sequence set 1 and the last packet of the sequence match corresponding to packet sequence set k (for a signature with k packet sequence sets) is below a threshold. A signature match is declared when all matching packet sequence sets occur within the duration $\lceil t + 0.1t \rceil$ with t being the maximum observed signature duration (see Tables 2.3, 2.4, 2.5, and 2.6).

Simultaneous Events. Finally, during signature matching, PingPong's algorithm first separates incoming packets into different sets on a per device basis: individual TCP connections in the WAN sniffer matching, or individual substreams based on source/destination MAC addresses in the Wi-Fi sniffer matching. Since the devices we have seen only generate events sequentially, sequences that correspond to a certain event will also appear sequentially in the network trace: there will never be an overlap of events in a single device. If two devices or more generate events simultaneously, the corresponding sequences will be either in separate TCP connections for the WAN sniffer adversary or separate substreams for the Wi-Fi sniffer adversary: there will never be overlap of events generated by different devices. This also implies that changing the amount and types of background traffic (e.g., video or audio streaming) does not affect our signatures—other traffic will be in different TCP connections/substreams.

2.4 Evaluation

In this section, we present the evaluation of PingPong. In Section 2.4.1, we show that PingPong automatically extracted event signatures for 18 devices as summarized in Tables 2.3, 2.4, 2.5, and 2.6—11 of which are the most popular devices on Amazon (see Table 2.1). In Section 2.4.2, we used the extracted signatures to detect events in a trace

Table 2.3: Smart plugs found to exhibit Phone-Cloud, Device-Cloud, and Phone-Device signatures. Prefix PH indicates Phone-to-device direction and prefix D indicates Device-to-phone direction in Signature column. Column **Dura.** reports numbers in the form of **Min./Avg./Max.**, namely minimum, average, and maximum durations respectively.

Device	**Event**	**Signature**	**Comm.**	**Dura. (ms)**	**Matching (Per 100 Events)**					
					No Defense				**STP+VPN**	
					WAN Snif.	**FP**	**Wi-Fi Snif.**	**FP**	**WAN Snif.**	**FP**
Amazon plug	ON	**S1:** S-[443-445] **S2:** C-1099 S-235	Device-Cloud	1,232 / 2,465 / 4,537	98	0	99	0	99	0
	OFF	**S1:** S-[444-446] **S2:** C-1179 S-235 **S3:** C-1514 C-103 S-235								
WeMo plug	ON/ OFF	**S1:** PH-259 PH-475 D-246	Phone-Device	33 / 42 / 134	-	-	100	0	-	-
WeMo Insight plug	ON/ OFF	**S1:** PH-259 PH-475 D-246	Phone-Device	32 / 39 / 97	-	-	99	0	-	-
TP-Link plug	ON	**S1:** C-556 S-1293	Device-Cloud	75 / 85 / 204	99	0	-	-	98	3
	OFF	**S1:** C-557 S-[1294-1295]								
	ON	**S1:** PH-112 D-115 **S2:** C-556 S-1293	Phone-Device & Device-Cloud	225 / 325 / 3,328	-	-	99	0	-	-
	ON ON	**S1:** PH-112 D-115 **S2:** C-557 S-[1294-1295]								
D-Link plug	ON/ OFF	**S1:** S-91 S-1227 C-784 **S2:** C-1052 S-647	Device-Cloud	4 / 1,194 / 8,060	95	0	95	0	95	0
	ON	**S1:** C-[1109-1123] S-613	Phone-Cloud	35 / 41 / 176	98	0	98	0	98	0
	OFF	**S1:** C-[1110-1124] S-613								
Smart-Things plug	ON	**S1:** C-699 S-511 **S2:** S-777 C-136	Phone-Cloud	335 / 537 / 2,223	92	0	92	0	92	0
	OFF	**S1:** C-700 S-511 **S2:** S-780 C-136								

collected from a realistic experiment on our smart home testbed. Section 2.4.3 discusses the results of negative control experiments: it demonstrates the uniqueness of the PingPong signatures in large (*i.e.*, with hundreds of millions of packets), publicly available, packet traces from smart home and office environments. Section 2.4.4 discusses the results of our experi-

Table 2.4: Smart light bulbs found to exhibit Phone-Cloud, Device-Cloud, and Phone-Device signatures. Prefix PH indicates Phone-to-device direction and prefix D indicates Device-to-phone direction in Signature column. Column **Dura.** reports numbers in the form of **Min./Avg./Max.**, namely minimum, average, and maximum durations respectively.

Device	**Event**	**Signature**	**Comm.**	**Dura. (ms)**	**Matching (Per 100 Events)**					
					No Defense				**STP+VPN**	
					WAN Snif.	**FP**	**Wi-Fi Snif.**	**FP**	**WAN Snif.**	**FP**
Sengled light bulb	ON	**S1:** S-[217-218] C-[209-210] **S2:** C-430 **S3:** C-466	Device-Cloud	4,304 / 6,238 / 8,145	97	0	-	-	97	0
	OFF	**S1:** S-[217-218] C-[209-210] **S2:** C-430 **S3:** C-465								
	ON	**S1:** C-211 S-1063 **S2:** S-1277	Phone-Cloud	4,375 / 6,356 / 9,132	93	0	97	0	96	1
	OFF	**S1:** C-211 S-1063 S-1276								
	Inte-nsity	**S1:** S-[216-220] C-[208-210]	Device-Cloud	16 / 74 / 824	99	2	-	-	99	5
	Inte-nsity	**S1:** C-[215-217] S-[1275-1277]	Phone-Cloud	3,916 / 5,573 / 7,171	99	0	99	0	98	2
Hue light bulb	ON	**S1:** C-364 **S2:** D-88	Device-Cloud & Phone-Device	11,019 / 12,787 / 14,353	-	-	-	-	-	-
	OFF	**S1:** C-365 **S2:** D-88								
TP-Link light bulb	ON	**S1:** PH-198 D-227	Phone-Device	8 / 77 / 148	-	-	100	4	-	-
	OFF	**S1:** PH-198 D-244								
	Inte-nsity	**S1:** PH-[240-242] D-[287-289]	Phone-Device	7 / 84 / 212	-	-	100	0	-	-
	Color	**S1:** PH-317 D-287	Phone-Device	6 / 89 / 174	-	-	100	0	-	-

ments when devices are triggered remotely from a smartphone and via a home automation service. Section 2.4.5 shows the uniqueness of signatures for devices from the same vendor. Section 2.4.6 discusses our findings when we used PingPong to extract signatures from a public dataset [166]. Finally, Section 2.4.7 discusses our study on parameters selection and sensitivity.

Table 2.5: Smart thermostats and sprinklers found to exhibit Phone-Cloud, Device-Cloud, and Phone-Device signatures. Prefix PH indicates Phone-to-device direction and prefix D indicates Device-to-phone direction in Signature column. Column **Dura.** reports numbers in the form of **Min./Avg./Max.**, namely minimum, average, and maximum durations respectively.

Device	**Event**	**Signature**	**Comm.**	**Dura. (ms)**	**Matching (Per 100 Events)**					
					No Defense				**STP+VPN**	
					WAN Snif.	**FP**	**Wi-Fi Snif.**	**FP**	**WAN Snif.**	**FP**
Thermostats										
Nest thermostat	Fan ON	**S1:** C-[891-894] S-[830-834]	Phone-Cloud	91 / 111 / 1,072	93	0	93	1	93	2
	Fan OFF	**S1:** C-[858-860] S-[829-834]								
Ecobee thermostat	HVAC Auto	**S1:** S-1300 C-640	Phone-Cloud	121 / 229 / 667	100	0	99	0	99	0
	HVAC OFF	**S1:** C-1299 C-640								
	Fan ON	**S1:** S-1387 C-640	Phone-Cloud	117 / 232 / 1,776	100	0	100	0	100	1
	Fan Auto	**S1:** C-1389 C-640								
Sprinklers										
Rachio sprinkler	Quick Run	**S1:** S-267 C-155	Device-Cloud	1,972 / 2,180 / 2,450	100	0	100	0	100	1
	Stop	**S1:** C-496 C-155 C-395								
	Standby / Active	**S1:** S-299 C-155 C-395	Device-Cloud	276 / 690 / 2,538	100	0	100	0	100	0
Blossom sprinkler	Quick Run	**S1:** C-326 **S2:** C-177 S-505	Device-Cloud	701 / 3,470 / 8,431	96	0	96	0	96	3
	Stop	**S1:** C-326 **S2:** C-177 S-458 **S3:** C-238 C-56 S-388								
	Quick Run	**S1:** C-649 S-459 C-574 S-507 **S2:** S-[135-139]	Phone-Cloud	70 / 956 / 3,337	93	0	93	0	93	0
	Stop	**S1:** C-617 S-431								
	Hibernate	**S1:** C-621 S-493	Phone-Cloud	121 / 494 / 1,798	95	0	93	0	93	1
	Active	**S1:** C-622 S-494 **S2:** S-599 C-566 S-554 C-566								

Table 2.6: Smart home security devices and others found to exhibit Phone-Cloud, Device-Cloud, and Phone-Device signatures. Prefix PH indicates Phone-to-device direction and prefix D indicates Device-to-phone direction in Signature column. Column **Dura.** reports numbers in the form of **Min./Avg./Max.**, namely minimum, average, and maximum durations respectively.

Device	**Event**	**Signature**	**Comm.**	**Dura. (ms)**	**Matching (Per 100 Events)**					
					No Defense				**STP+VPN**	
					WAN Snif.	**FP**	**Wi-Fi Snif.**	**FP**	**WAN Snif.**	**FP**
Home Security Devices										
Ring alarm	Arm	**S1:** S-99 S-254 C-99 S-[181-183] C-99	Device-Cloud	275 / 410 / 605	98	0	95	0	95	0
	Disarm	**S1:** S-99 S-255 C-99 S-[181-183] C-99								
Arlo camera	Stream ON	**S1:** C-[338-339] S-[326-329] C-[364-365] S-[1061-1070] **S2:** C-[271-273] S-[499-505]	Phone-Cloud	46 / 78 / 194	99	2	98	3	97	4
	Stream OFF	**S1:** C-[445-449] S-442								
D-Link siren	ON	**S1:** C-1076 S-593	Phone-Cloud	36 / 37 / 65	100	0	98	0	97	0
	OFF	**S1:** C-1023 S-613								
Kwikset door lock	Lock	**S1:** C-699 S-511 **S2:** S-639 C-136	Phone-Device	173 / 395 / 2,874	100	0	100	0	100	0
	Unlock	**S1:** C-701 S-511 **S2:** S-647 C-136								
Others										
Roomba robot	Clean	**S1:** S-[1014-1015] C-105 S-432 C-105	Phone-Cloud	123 / 2,038 / 5,418	91	0	94	0	94	1
	Back-to-station	**S1:** S-440 C-105 S-[1018-1024] C-105								
Average (Tables 2.3, 2.4, 2.5, and 2.6)					97.05	0.18	97.48	0.32	96.77	1.09

2.4.1 Extracting Signatures from Smart Home Devices

Training Dataset. In order to evaluate the generalizability of packet-level signatures, we first used PingPong to automate the collection of training sets (see Section 2.3.1) for all 19 smart home devices (see Table 2.1). Training sets were collected for every device under test, individually without any background traffic (see Figure 2.1). The automation script generated a total of 100 events for the device. For events with binary values, the script generated $n = 50$ events for each event type (*e.g.*, 50 ON and 50 OFF events). For events with continuous values, the script generated $n = 100$ events (*e.g.*, 100 intensity events for the Sengled light bulb).

Results Summary. For each training set, we used PingPong to extract packet-level signatures (see Section 2.3.1) for each event type of the respective device. In summary, PingPong extracted signatures from 18 devices (see Tables 2.3, 2.4, 2.5, and 2.6). The signatures span a wide range of event types: binary (*e.g.*, ON/OFF) and non-binary (*e.g.*, light bulb intensity, color, etc.). Similar to our manual observation described in Section 2.2.3, we again see that these events are identifiable by the request-reply pattern.

Tables 2.3, 2.4, 2.5, and 2.6 present the signatures that PingPong identified.[2] Each line in a signature cell represents a packet sequence set, and the vertical positioning of these lines reflects the ordering of the packet sequence sets in the signature (see Section 2.3.1 for the notation).

PingPong performed well in extracting signatures: it has successfully extracted packet-level signatures that are observable in the device's Phone-Cloud, Device-Cloud, and Phone-Device communications (see Tables 2.3, 2.4, 2.5, and 2.6). Although the traffic is typically encrypted using TLSv1.2, the event still manifests itself in the form of a packet-level signature in the Phone-Cloud or Device-Cloud communication. PingPong also extracted signatures from the Phone-Device communication for some of the devices. These signatures are extracted typically from unencrypted local TCP/HTTP communication between the smartphone and the device.

[2]Phone-Device signatures are observable only by the Wi-Fi sniffer. The Sengled light bulb's Device-Cloud signatures are sent by the Zigbee hub to the cloud through the Ethernet interface; thus, they are not observable by the Wi-Fi sniffer. The Hue light bulb has unique signatures; they consist of a pair, in which one is a Device-Cloud packet coming from the Zigbee hub to the cloud—this is observable only by the WAN sniffer since the hub is an Ethernet device—and the other one is a Phone-Cloud packet—this is observable only by the Wi-Fi sniffer; thus we did not use the signatures to perform detection since they partially belong to both adversaries.

Smart Plugs PingPong extracted signatures from all 6 plugs: the Amazon, WeMo, WeMo Insight, TP-Link, D-Link, and SmartThings plugs. The Amazon, D-Link, and SmartThings plugs have signatures in the Phone-Cloud or Device-Cloud communication, or both. The TP-Link plug has signatures in both the Device-Cloud and Phone-Device communications. Both the WeMo and WeMo Insight plugs have signatures in the Phone-Device communication. In general, the signatures allow us to differentiate ON from OFF except for the WeMo, WeMo Insight, TP-Link plug's Phone-Device communication, and D-Link plug's Device-Cloud communication (see Table 2.3).

Light Bulbs. PingPong extracted signatures from 3 light bulbs: the Sengled, Hue, and TP-Link light bulbs. The Sengled light bulb has signatures in both the Phone-Cloud and Device-Cloud communications. The Hue light bulb has signatures in both Device-Cloud and Phone-Device communications. The TP-Link light bulb has signatures only in the Phone-Device communication. Table 2.4 shows that PingPong also extracted signatures for events beyond ON/OFF, *e.g.*, Color.

Thermostats. PingPong extracted signatures for both the Nest and Ecobee thermostats. Both thermostats have Phone-Cloud signatures. The signatures allow us to differentiate Fan ON/OFF/Auto events. The Ecobee thermostat's signatures also leak information about its HVAC Auto/OFF events.

Sprinklers. PingPong extracted signatures from both the Rachio sprinkler and Blossom sprinkler. Both sprinklers have signatures in both the Device-Cloud and Phone-Cloud communications. The signatures allow us to differentiate Quick Run/Stop and Standby/Hibernate/Active events.

Home Security Devices. A highlight is that PingPong extracted signatures from home security devices. Notably, the Ring alarm has signatures that allow us to differentiate Arm/Disarm events in the Device-Cloud communication. The Arlo camera has signatures for

Stream ON/OFF events, the D-Link siren for ON/OFF events, and the Kwikset lock for Lock/Unlock events in the Phone-Cloud communication.

Roomba Robot. Finally, PingPong also extracted signatures from the Roomba robot in the Phone-Cloud communication. These signatures allow us to differentiate Clean/Back-to-station events.

Signature Validity. Recall that signature validation rejects a signature candidate whose sequences are present not only in the time window t, but also during the subsequent idle period (see Section 2.3.1). We saw such a signature candidate for one device, namely the LiFX light bulb. PingPong captured a signature candidate that is present also in the idle period of the TCP communication and then rejected the signature during the validation phase. Manual inspection revealed that the LiFX light bulb uses unidirectional UDP communication (*i.e.*, no request-reply pattern) for events.

2.4.2 Smart Home Testbed Experiment

Testing Dataset. To evaluate the effectiveness of packet-level signatures in detecting events, we collected a separate set of network traces and used PingPong to perform detection on them. We used the setup in Section 2.2.2 to collect one dataset for every device. Our smart home setup consists of 13 of the smart home devices presented in Table 2.1: the WeMo plug, WeMo Insight plug, Hue light bulb, LiFX light bulb, Nest thermostat, Arlo camera, TP-Link plug, D-Link plug, D-Link siren, TP-Link light bulb, SmartThings plug, Blossom sprinkler, and Kwikset lock. This fixed set of 13 devices was our initial setup—it gives us the flexibility to test additional devices without changing the smart home setup and needing to rerun all the experiments, yet still includes a variety of devices that generate background traffic. While collecting a dataset, we triggered events for the device under test. At the same time, we also connected the other 12 devices and turned them ON before we started the experiment—this allows the other devices to generate network traffic as they communicate with their cloud servers. However, we did not trigger events for these other devices. For the other 6 devices (the Amazon plug, Sengled light bulb, Ecobee thermostat, Rachio sprinkler, Roomba robot, and Ring alarm), we triggered events for the device under test while having all the 13 devices turned on. To generate additional background traffic as depicted in Figure 2.1, we set up 3 general purpose devices: a Motorola Moto g^6 phone that would play a YouTube video playlist, a Google Nexus 5 phone that would play a Spotify song playlist, and an Apple MacBook Air that would randomly browse top 10 websites [29] every 10-500 seconds. We used this setup to emulate the network traffic from a smart home with many active devices.

Results Summary. Tables 2.3, 2.4, 2.5, and 2.6 present the summary of our results (see column "**Matching**"). We collected a dataset with 100 events for every type of event—for binary events (*e.g.*, ON/OFF), we triggered 50 for each value. We performed the detection for both the WAN sniffer and Wi-Fi sniffer adversaries: we have a negligible False Positive Rate (FPR) of 0.25 (0.18 for the WAN sniffer and 0.32 for the Wi-Fi sniffer) per 100 events for every event type.

Later, we added 6 more devices into our smart home setup: the Amazon plug, Sengled light bulb, Ecobee thermostat, Rachio sprinkler, Roomba robot, and Ring alarm.[3] For these newly added devices, we still generated events for the device under test while collecting a dataset. However, this time we used all of our first 13 devices and the 3 general purpose devices in our initial setup to generate background traffic.

[3]Our initial setup had already allowed us to observe this packet-level signature in a mixed group of most popular and less popular devices. We then decided to further confirm our findings by adding 6 more most popular devices that eventually gave us a set of 19 devices with 12 devices being the most popular smart home devices on Amazon [33] (see Section 2.2.2).

2.4.3 Negative Control Experiment

If the packet-level signatures are to be used to detect events in traffic in the wild, they must be sufficiently unique compared to other traffic to avoid generating false positives. We evaluated the uniqueness of the signatures by performing signature detection on 3 datasets. The first 2 datasets serve to evaluate the uniqueness of the signatures among traffic generated by similar devices (*i.e.*, other smart home devices), while the third dataset serves to evaluate the uniqueness of the signatures among traffic generated by general purpose computing devices.

Dataset 1: UNSW Smart Home Traffic Dataset. The first dataset [180] contains network traces for 26 smart home devices that are *different* from the devices that we generated signatures for. The list can be found in [199]. The dataset is a collection of 19 PCAP files, with a total size of 12.5GB and a total of 23,013,502 packets.

Dataset 2: YourThings Smart Home Traffic Dataset. The second dataset [31, 32] contains network traces for 45 smart home devices. The dataset is a collection of 2,880 PCAP files, with a total size of 270.3GB and 407,851,830 packets. There are 3 common devices present in both YourThings and our set of 18 devices: the WeMo plug, Roomba robot, and TP-Link light bulb.

Dataset 3: UNB Simulated Office-Space Traffic Dataset. The third dataset is the Monday trace of the CICIDS2017 dataset [177]. It contains simulated network traffic for an office space with two servers and 10 laptops/desktops with diverse operating systems. The dataset we used is a single PCAP file of 10.82GB, with a total of 11,709,971 packets observed at the WAN interface.

False Positives. For datasets 1 and 3, we performed signature detection for all devices. For dataset 2, we only performed signature detection for the 15 of our devices that are *not* present in YourThings to avoid the potential for true positives. We used WAN sniffer detection for devices with Phone-Cloud and Device-Cloud signatures, and Wi-Fi sniffer detection for all devices.

WAN Sniffer. There were no false positives across 23,013,502 packets in dataset 1, 1 false positive for the Sengled light bulb across 407,851,830 packets in dataset 2, and 1 false positive for the Nest thermostat across 11,709,971 packets in dataset 3.

Wi-Fi Sniffer. PingPong detected some false positives due to its more relaxed matching strategy (see Section 2.3.2). The results show that the extracted packet-level signatures are unique: the average FPR is 11 false positives per signature across a total of 442,575,303 packets from all three datasets (*i.e.*, an average of 1 false positive per 40 million packets).

Further analysis showed that signatures comprised of a single packet pair (*e.g.*, the D-Link plug's Phone-Cloud signatures that only have one request and one reply packet) contributed the most to the average FPR—FPR is primarily impacted by signature length, not device type. Five 3-packet signatures generated 5, 7, 16, 26, and 33 false positives, while one 4-packet signature generated 2 false positives. There were also three outliers: two 4-packet signatures generated 46 and 33 false positives, and a 6-packet signature generated 18 false positives. This anomaly was due to PingPong using the range-based matching strategy for these signatures (see Section 2.3.2). Furthermore, the average of the packet lengths for the signatures that generated false positives is less than 600 bytes: the packet lengths distribution for our negative datasets shows that there are significantly more shorter packets than longer packets.

2.4.4 Events Triggered Remotely

Our main dataset, collected using our testbed (see Section 2.4.1), contains events triggered by a smartphone that is part of the local network. However, smart home devices can also

Table 2.7: Device-Cloud signature in 9 devices triggered by IFTTT home automation.

Device	**Event**	**Device-Cloud Signature**	**Duration (ms) Min./Avg. /Max.**	**Matching (Per 100 Events)**			
				WAN Sniffer	**FP**	**Wi-Fi Sniffer**	**FP**
Plugs							
WeMo plug	ON/OFF	**S1:** S-146 **S2:** C-210 S-134 S-286 C-294	226 / 345 / 2,236	100	0	100	0
WeMo Insight plug	ON	**S1:** S-146 **S2:** C-210 S-134 S-286 C-294	216 / 253 / 473	99	0	94	0
	OFF	**S1:** S-146 **S2:** C-210 S-134 S-350 C-294					
TP-Link plug	ON	**S1:** C-592 S-1234 S-100	71 / 76 / 139	100	0	100	0
	OFF	**S1:** C-593 S-1235 S-100					
D-Link plug	ON/OFF	**S1:** C-256 **S2:** C-1020 S-647	269 / 750 / 6,364	93	1	93	1
Light Bulbs							
Hue light bulb	ON	**S1:** S-[227-229] C-[857-859] C-365	37 / 44 / 76	99	1	-	-
	OFF	**S1:** S-[227-230] C-[857-860] C-366					
	Intensity	**S1:** S-[237-240] C-[895-899] **S2:** C-[378-379]	36 / 40 / 96	97	0	-	-
TP-Link light bulb	ON	**S1:** S-[348-349] C-[399-400]	11 / 15 / 78	100	0	100	0
	OFF	**S1:** S-[348-349] C-[418-419]					
	Intensity	**S1:** S-[438-442] C-[396-400]	12 / 16 / 53	100	0	99	0
	Color	**S1:** S-[386-388] C-[397-399]	12 / 17 / 60	99	0	97	0
Others							
Rachio sprinkler	Quick Run	**S1:** S-267 C-155	1,661 / 2467 / 4,677	95	3	95	5
	Stop	**S1:** C-661 **S2:** C-155					
Arlo camera	Start Recording	**S1:** C-704 S-215	156 / 159 / 195	100	0	99	0
D-Link siren	ON	**S1:** S-[989-1005] C-616 **S2:** C-216	162 / 181 / 281	99	1	98	1
			Average	98.4	0.5	97.5	0.7

be controlled remotely, using home automation frameworks or a remote smartphone. In this section, we show that even with such remote triggers, the Device-Cloud communication exhibits similar signatures, which can be extracted using PingPong.

Table 2.8: Comparison of Device-Cloud signatures for three devices (TP-Link plug, D-Link plug, and Rachio sprinkler) triggered in three different ways: via (i) a local phone, (ii) a remote phone, and (iii) IFTTT home automation.

Device	**Event**	**Device-Cloud Signatures**		
		Local-Phone	**Remote-Phone**	**IFTTT**
TP-Link plug	ON	**S1:** C-592 S-1234 S-100	**S1:** C-592 S-1234 S-100	**S1:** C-592 S-1234 S-100
	OFF	**S1:** C-593 S-1235 S-100	**S1:** C-593 S-1235 S-100	**S1:** C-593 S-1235 S-100
D-Link plug	ON/OFF	**S1:** S-91 S-1227 C-784 **S2:** C-1052 S-647	**S1:** S-91 S-1227 C-784 **S2:** C-1052 S-647	**S1:** C-256 **S2:** C-1020 S-647
Rachio sprinkler	Quick Run	**S1:** S-267 C-155	**S1:** S-267 C-155 C-837 C-448	**S1:** S-267 C-155
	Stop	**S1:** C-496 C-155 C-395	**S1:** S-219 S-235 C-171 C-661 C-496 C-155 C-395	**S1:** C-661 **S2:** C-155

Home Automation Experiment (IFTTT). In this section, we trigger events using the IFTTT (If-This-Then-That) home automation platform [23], which supports most of our devices. IFTTT is one of the most popular web-based home automation frameworks: in 2019 it provides around 700 services for 17 million users [186]. IFTTT allows users to set up automation rules that work in a *Trigger-Action* fashion. Each automation rule typically connects a trigger (*i.e.*, *This*) to an action (*i.e.*, *That*). The occurrence of the trigger causes the action to be performed by the framework. For example, "if motion is detected, then turn on the smart plug" [27].

In this experiment, we use IFTTT to trigger events. We integrate IFTTT into our existing infrastructure for triggering device events via Android widget button presses. For each event type, we develop an IFTTT rule that triggers the event using an Android button widget. For example, we set up a button widget to toggle ON the TP-Link plug: "if the widget button is pressed, then turn on the smart plug". Then, we install the IFTTT app on our smartphone, log in to the app using our IFTTT account, and add the button widget onto the home screen of the smartphone. Then, instead of using the official Android app, we use the button widget to trigger events from the smartphone. The smartphone is connected to a different network than the smart home testbed network to simulate controlling the devices from a remote location.

For every device: (1) we use IFTTT to trigger events to collect a new training dataset, and we run PingPong to extract packet-level signatures from the Device-Cloud communication in this dataset; (2) we collect a new smart home testbed dataset (as in Section 2.4.2, but using the IFTTT widget to trigger events), which we then use for testing, *i.e.*, to detect events by matching on the IFTTT signatures extracted from the aforementioned training dataset.

Signatures Found in Device-Cloud Communication. IFTTT provides support for 13 out of our 18 devices: no support was provided at the time of the experiment for the Amazon plug, Blossom sprinkler, Roomba robot, Ring alarm, and Nest thermostat. The main finding is that, from the supported 13 devices, PingPong successfully extracts Device-Cloud signatures for nine devices and 12 event types, which are summarized in Table 2.7.[4] Three out of the nine supported devices (the TP-Link plug, the D-Link plug, and the Rachio sprinkler) already have Device-Cloud signatures when triggered by a local phone: the phone is connected to the smart home testbed, where the device is also connected to (see Tables 2.3, 2.4, 2.5, and 2.6). Interestingly, six out of the nine supported devices have Device-Cloud signatures when triggered via IFTTT, but did not have Device-Cloud signatures when triggered by a local phone.

Comparison of Device-Cloud Signatures. Having answered the main question (*i.e.*, that there are indeed Device-Cloud signatures even when devices are triggered by IFTTT), a secondary question is whether the signature varies depending on the way the device is triggered. To answer this question, we consider the TP-Link plug, the D-Link plug, and the Rachio sprinkler, and we trigger events in three different ways:

1) Local-Phone: signatures are extracted from the previous experiment (see Tables 2.3, 2.4, 2.5,

[4]PingPong did not extract Device-Cloud signatures from 4 devices: the Sengled light bulb, Ecobee thermostat, SmartThings plug, and Kwikset lock. For the Ecobee thermostat, SmartThings plug, and Kwikset lock. PingPong extracted signatures from the Phone-Cloud communication (not from the Device-Cloud communication) in the previous experiment (see Tables 2.3, 2.4, 2.5, and 2.6). For the Sengled light bulb, the device was recently forced to update its firmware—PingPong in its current state did not extract signatures in the Device-Cloud communication anymore although there is potentially a new signature.

and 2.6), using the vendor's official Android application to trigger events. The phone is connected to the smart home testbed network.

2) Remote-Phone: signatures are extracted from training datasets we collect using a remote phone setting (without using IFTTT). We use the vendor's official Android application to trigger events for each device. We connect the phone to a different network than the smart home testbed network.
3) IFTTT: signatures are extracted from a training dataset collected with the IFTTT home automation experiment described in this section.

Table 2.8 lists all the Device-Cloud signatures we extract. We can see that the majority of Device-Cloud signatures are the same or very similar across the IFTTT, Local-Phone, and Remote-Phone experiments. For the TP-Link plug, the Device-Cloud signatures from the three experiments are the same or similar (same packet sequences within 1B).[5] For the D-Link plug, the Local-Phone and the Remote-Phone Device-Cloud signatures are the same, but the IFTTT Device-Cloud signatures are partially similar to the Local-Phone and Remote-Phone Device-Cloud signatures. For the Rachio sprinkler, the Device-Cloud signatures from the three experiments are subsets of one another.

2.4.5 Devices from the Same Vendor

Since the signatures reflect protocol behavior, a natural question to ask is whether devices from the same vendor, which probably run similar protocols, have the same signature. In our testbed experiment, we had already extracted signatures from 2 TP-Link devices: the TP-Link plug and TP-Link light bulb (see Tables 2.3, 2.4, 2.5, and 2.6). We also acquired, and experimented with, 4 additional devices from TP-Link. We report the detailed results in Table 2.9. In summary, we found that packet-level signatures have some similarities (*e.g.*, the TP-Link two-outlet plug and TP-Link power strip have similar functionality and have

[5]We repeated our previous experiment and found that the Device-Cloud signatures presented in Table 2.3 have evolved over time and since the original experiment.

Table 2.9: Signatures extracted from different TP-Link devices. ∗These are the latest signatures for the TP-Link plug and TP-Link light bulb (per December 2019).

Device	**Model**	**Event**	**Signature**	**Duration (ms) Min./Avg./Max.**
Existing TP-Link Devices				
TP-Link plug	HS-110	ON	∗**S1:** PH-172 D-115 **S2:** C-592 S-1234 S-100	406 / 743 / 10,667
		OFF	∗**S1:** PH-172 D-115 **S2:** C-593 S-1235 S-100	
TP-Link light bulb	LB-130	ON	∗**S1:** PH-258 D-288	8 / 77 / 148
		OFF	∗**S1:** PH-258 D-305	
		Intensity	**S1:** PH-[240-242] D-[287-289]	7 / 84 / 212
		Color	**S1:** S1: PH-317 D-287	6 / 89 / 174
Newly Added TP-Link Devices				
TP-Link two-outlet plug	HS-107	ON	**S1:** PH-219 D-103 **S2:** C-300 C-710 S-1412 S-88	1,083 / 1593 / 2,207
		OFF	**S1:** PH-219 D-103 **S2:** C-300 C-711 S-1413 S-88	
TP-Link power strip	HS-300	ON	**S1:** PH-219 D-103 **S2:** C-301 C-1412 S-[1405-1406] S-88	976 / 1,537 / 4,974
		OFF	**S1:** PH-219 D-103 **S2:** C-301 C-1413 S-[1406-1407] S-88	
TP-Link white light bulb	KL-110	ON	**S1:** S-[414-415] C-[331-332] **S2:** C-648 S-[1279-1280] S-88	1,892 / 2,021 / 2,157
		OFF	**S1:** S-[414-415] C-[350-351] **S2:** C-649 S-[1280-1281] S-88	
		Intensity	**S1:** S-[479-483] C-[329-332] **S2:** C-[654-656] S-[1285-1288] S-88	2,418 / 2,540 / 3,610
TP-Link camera	KC-100	ON	**S1:** PH-256 D-162 PH-624 D-256 PH-72 D-111 PH-608 D-371 PH-97 S-100 **S2:** C-1288 S-[1161-1162] S-100	804 / 1,105 / 1,739
		OFF	**S1:** PH-256 D-162 PH-624 D-256 PH-72 D-111 PH-614 D-371 PH-97 **S2:** C-1289 S-[1162-1163] S-100	

packet lengths 1412B and 88B). However, they are still distinct across different device models and event types, even for devices with similar functionality (*e.g.*, the TP-Link plug, TP-Link two-outlet plug, and TP-Link power strip).

2.4.6 Public Dataset Experiment

In this section, we apply the PingPong methodology to a state-of-the-art, publicly available IoT dataset: the Mon(IoT)r dataset [166]. First, we show that PingPong successfully

Table 2.10: Signatures extracted from the cameras and light bulbs only in the Mon(IoT)r [166] dataset.

Device	Event	Signature	Duration (ms)
Cameras			
Amazon camera	Watch	**S1:** S-[627-634] C-[1229-1236]	203 / 261 / 476
Blink hub	Watch	**S1:** S-199 C-135 C-183 S-135	99 / 158 / 275
	Photo	**S1:** S-199 C-135 C-183 S-135	87 / 173 / 774
Lefun camera	Photo	**S1:** S-258 C-[206-210] S-386 C-206 **S2:** C-222 S-198 C-434 S-446 C-462 S-194 C-1422 S-246 C-262 **S3:** C-182	17,871 / 19,032 / 20,358
	Recording	**S1:** S-258 C-210 S-386 C-206 **S2:** C-222 S-198 C-434 S-446 C-462 S-194	13,209 / 15,279 / 16,302
	Watch	**S1:** S-258 C-210 S-386 C-206 **S2:** C-222 S-198 C-434 S-446 C-462 S-194	14,151 / 15,271 / 16,131
Microseven camera	Watch	**S1:** D-242 PH-118	1 / 5 / 38
ZModo doorbell	Photo	**S1:** C-94 S-88 S-282 C-240 / **S1:** S-282 C-240 C-94 S-88	1,184 / 8,032 / 15,127
	Recording	**S1:** C-94 S-88 S-282 C-240 / **S1:** S-282 C-240 C-94 S-88	305 / 7,739 / 15,137
	Watch	**S1:** C-94 S-88 S-282 C-240 / **S1:** S-282 C-240 C-94 S-88	272 / 7,679 / 15,264
Light Bulbs			
Flex light bulb	ON/OFF	**S1:** PH-140 D-[346-347]	4 / 44 / 78
	Intensity	**S1:** PH-140 D-346	4 / 18 / 118
	Color	**S1:** PH-140 D-346	4 / 12 / 113
Wink hub	ON/OFF	**S1:** PH-204 D-890 PH-188 D-113	43 / 55 / 195
	Intensity	**S1:** PH-204 D-890 PH-188 D-113	43 / 50 / 70
	Color	**S1:** PH-204 D-890 PH-188 D-113	43 / 55 / 106

extracted signatures from new devices in this dataset, thus validating the generality of the methodology and expanding our coverage of devices. Then, we compare the signatures extracted from the Mon(IoT)r dataset to those extracted from our testbed dataset, for devices that were present in both.

The Mon(IoT)r Dataset. The Mon(IoT)r dataset [166] contains network traces from 55 distinct IoT devices.[6] Each PCAP file in the dataset contains traffic observed for a single device during a short timeframe surrounding a single event on that device. Moreover,

[6]The paper [166] reports results from 81 physical devices, but 26 device models are present in both the US and the UK testbed, thus there are only 55 distinct models.

Table 2.11: Signatures extracted from the voice command devices only in the Mon(IoT)r [166] dataset.

Device	**Event**	**Signature**	**Duration (ms)**
Allure speaker	Audio ON/OFF	**S1:** C-658 C-412	89 / 152 / 196
	Volume	**S1:** C-[594-602] **S2:** C-[92-100]	217 / 4,010 / 11,005
Amazon Echo Dot	Voice	**S1:** C-491 S-[148-179]	1 / 23 / 61
	Volume	**S1:** C-[283-290] C-[967-979] **S2:** C-[197-200] C-[147-160]	1,555 / 2,019 / 2,423
Amazon Echo Plus	Audio ON/OFF	**S1:** S-100 C-100	1 / 5 / 28
	Color	**S1:** S-100 C-100	1 / 4 / 18
	Intensity	**S1:** S-100 C-100	1 / 4 / 11
	Voice	**S1:** C-[761-767] S-437 **S2:** C-172 S-434	1,417 / 1,871 / 2,084
	Volume	**S1:** C-172 S-434	2 / 13 / 40
Amazon Echo Spot	Audio ON/OFF	**S1:** S-100 C-100	1 / 8 / 233
	Voice	**S1:** C-246 S-214 **S2:** C-172 S-434	1,220 / 1,465 / 1,813
	Volume	**S1:** C-246 S-214 **S2:** C-172 S-434	1,451 / 1,709 / 1,958
Google Home	Voice	**S1:** C-1434 S-136	9 / 61 / 132
	Volume	**S1:** C-1434 S-[124-151] **S2:** C-521 S-[134-135]	8,020 / 9,732 / 10,002
Google Home Mini	Voice	**S1:** C-1434 S-[127-153]	1 / 29 / 112
	Volume	**S1:** C-1434 S-[135-148]	5 / 47 / 123
Harman Kardon Invoke speaker	Voice	**S1:** S-1494 S-277 C-1494 **S2:** S-159 S-196 C-1494	2,199 / 2,651 / 3,762
	Volume	**S1:** S-159 S-196 C-1418 C-1320 S-277 **S2:** S-196 C-[404-406]	223 / 567 / 793

the authors provide timestamps for when they performed each event. As a result, we can merge all PCAP files for each device and event type combination into a single PCAP file, and directly apply PingPong to extract signatures, similarly to how we extracted signatures from the training set we collected using our testbed. We only considered a subset of the 55 devices in the Mon(IoT)r dataset, due to a combination of limitations of the dataset and of our methodology. In particular, we did not apply PingPong to the following groups of devices in the Mon(IoT)r dataset: (1) 3 devices with nearly all PCAP files empty; (2) 6 devices with a limited number (three or less) of event samples;[7] and (3) 13 devices that only communicate

[7]We consider this to be too few samples to have confidence in the extracted signatures. In contrast, the traces for the remaining devices generally had 30–40 event samples for each device and event type combination.

Table 2.12: Signatures extracted from the smart TVs and other devices only in the Mon(IoT)r [166] dataset.

Device	**Event**	**Signature**	**Duration (ms)**
		Smart TVs	
Fire TV	Menu	**S1:** C-468 S-323	16 / 18 / 20
LG TV	Menu	**S1:** PH-204 D-1368 PH-192 D-117	43 / 90 / 235
Roku TV	Remote	**S1:** PH-163 D-[163-165] **S2:** PH-145 D-410 **S2:** PH-147 D-113	578 / 1,000 / 1,262
Samsung TV	Menu	**S1:** PH-[237-242] D-274	2 / 7 / 15
		Other Types of Devices	
Honeywell thermostat	ON	**S1:** S-635 C-256 C-795 S-139 C-923 S-139	1,091 / 1,248 / 1,420
	OFF	**S1:** S-651 C-256 C-795 S-139 C-923 S-139	
	Set	**S1:** C-779 S-139	86 / 102 / 132
Insteon hub	ON/OFF	**S1:** S-491 C-623 **S2:** C-784 C-234 S-379	76 / 100 / 1,077
Samsung fridge	Set	**S1:** C-116 S-112	177 / 185 / 185
	View Inside	**S1:** C-116 S-112	177 / 197 / 563

Table 2.13: Common devices that have the same signatures in the Mon(IoT)r and our testbed experiments. ∗ signature: training on our testbed. † signature: training on Mon(IoT)r [166]. Matching: training on testbed, detection on Mon(IoT)r. The number of events vary (around 30-40) per event type—the result is presented in % for convenience.

Device	**Event**	**Signature**	**Duration (ms) Min./Avg./Max./St.Dev.**	**Matching**			
				WAN Sniffer	**FP**	**Wi-Fi Sniffer**	**FP**
WeMo Insight plug	ON/ OFF	∗**S1:** PH-475 D-246	29 / 33 / 112 / 9	-	-	98.75%	0
		†**S1:** PH-475 D-246	31 / 42 / 111 / 15				
Blink camera	Watch	∗**S1:** C-331 S-299 C-139	267 / 273 / 331 / 8	100%	0	100%	0
		†**S1:** C-331 S-299 C-139	170 / 269 / 289 / 19				
	Photo	∗**S1:** C-331 C-123 S-139 S-123 S-187 C-1467	281 / 644 / 1,299 / 348	97.37%	0	97.50%	0
		†**S1:** C-331 C-123 S-139 S-123 S-187 C-1467	281 / 742 / 2,493 / 745				

via UDP (PingPong's current implementation only considers TCP traffic). Next, we report results from applying PingPong to the remaining 33 devices in the Mon(IoT)r dataset. Out

Table 2.14: Common devices that have similar signatures in the Mon(IoT)r and our testbed experiments. ∗ signature: training on our testbed. † signature: training on Mon(IoT)r [166]. Matching: training on testbed, detection on Mon(IoT)r. The number of events vary (around 30-40) per event type—the result is presented in % for convenience.

Device	**Event**	**Signature**	**Duration (ms) Min./Avg./Max./ St.Dev.**	**Matching**			
				WAN Snif.	**FP**	**Wi-Fi Snif.**	**FP**
TP-Link plug (Device-Cloud)	ON	∗**S1:** C-592 S-1234 S-100	70 / 74 / 85 / 2	100%	0	-	-
	OFF	∗**S1:** C-593 S-1235 S-100					
	ON	†**S1:** C-605 S-1213 S-100	16 / 19 / 29 / 2				
	OFF	†**S1:** C-606 S-1214 S-100					
TP-Link plug (Phone-Device & Device-Cloud)	ON	∗**S1:** PH-172 D-115 **S2:** C-592 S-1234 S-100	406 / 743 / 10,667 / 1,417	-	-	100%	0
	OFF	∗**S1:** PH-172 D-115 **S2:** C-593 S-1235 S-100					
	ON	†**S1:** PH-172 D-115 **S2:** C-605 S-1213 S-100	197 / 382 / 663 / 165				
	OFF	†**S1:** PH-172 D-115 **S2:** C-606 S-1214 S-100					
Sengled light bulb	ON	∗**S1:** S-[217-218] C-[209-210] **S2:** C-430 **S3:** C-466	4,304 / 6,238 / 8,145 / 886	-	-	-	-
	OFF	∗**S1:** S-[217-218] C-[209-210] **S2:** C-430 **S3:** C-465					
	ON	†**S1:** S-219 C-210 **S2:** C-428 **S3:** C-[478-479]	354 / 2,590 / 3,836 / 859				
	OFF	†**S1:** S-219 C-210 **S2:** C-428 **S3:** C-[478-480]					
TP-Link light bulb	ON	∗**S1:** PH-258 D-288	8 / 77 / 148 / 42	-	-	-	-
	OFF	∗**S1:** PH-258 D-305					
	ON	†**S1:** PH-258 D-227	17 / 92 / 224 / 46				
	OFF	†**S1:** PH-258 D-244					

of those, 26 are exclusive to the Mon(IoT)r dataset, while seven are common across the Mon(IoT)r dataset and our testbed dataset.

Devices only in the Mon(IoT)r Dataset. We ran PingPong's signature extraction on the traces from the 26 new devices from the Mon(IoT)r dataset. PingPong successfully extracted signatures for 21 devices and we summarize those signatures in Tables 2.10, 2.11, and 2.12.[8] Some of these devices provide similar functionality as those in our testbed dataset (*e.g.*, bulbs, cameras). Interestingly, we were also able to successfully extract signatures for many new types of devices that we did not have access to during our testbed experiments. Examples include voice-activated devices, smart TVs, and even a fridge. This validates the generality of the PingPong methodology and greatly expands our coverage of devices.

There were also 5, out of 26, new devices that PingPong originally appeared to not extract signatures from. However, upon closer inspection of their PCAP files and PingPong's output logs, we observed that those devices did actually exhibit a new type of signature that we had not previously encountered in our testbed experiments: a sequence of packet pairs with the exact same pair of packet lengths for the same event. The default configuration of PingPong would have discarded the clusters of these packet pairs during the signature creation of the training phase (see Section 2.3.1), because the number of occurrences of these pairs is higher than (in fact a multiple of) the number of events. However, based on this intuitive observation, PingPong can easily be adapted to extract those signatures as well: it can take into account the timing of packet pairs in the same cluster instead of only across different clusters, and concatenate them into longer sequences. We note that these frequent packet pairs can be either new signatures for new devices, or can be due to periodic background communication. Unfortunately, the Mon(IoT)r dataset does not include network traces for idle periods (where no events are generated), thus we cannot confirm or reject this hypothesis.

Common Devices. We next report our findings for devices that are present in both the Mon(IoT)r dataset and in our own testbed dataset, referred to as common devices. There were already 6 common devices across the 2 datasets, and we acquired an additional device after consulting with the authors of the paper: the Blink camera. We excluded 2

[8]For some of the devices, we had to relax the training time window, *i.e.*, $t = 30$s, because there is usually a gap of more than 20s from the provided timestamps to the appearance of event-related traffic. As a result, PingPong extracted longer signatures for certain devices, *e.g.*, the Lefun camera. Further, we also had to exclude some PCAP files that are empty or contain traffic that look inconsistent with the majority (*e.g.*, some only contain DNS or IGMP traffic).

common devices: (1) the Nest thermostat as it was tested for different event types; and (2) the Hue light bulb as it has a unique signature that PingPong cannot use to perform matching—it is a combination of Device-Cloud (visible only to the WAN sniffer) and Phone-Device communications (visible only to the Wi-Fi sniffer). Tables 2.13 and 2.14 summarize the results for the 5 remaining common devices. First, we report the complete signatures extracted from each dataset. The signatures reported in Table 2.3 were obtained from data collected throughout 2018. For the WeMo Insight plug and TP-Link plug, we repeated our testbed data collection and signature extraction in December 2019 to facilitate a better comparison of signatures from the same devices across different points in time. Then, we compare the signatures extracted from the two datasets for the common devices: some of the signatures are identical and some are similar. Such a comparison provides more information than simply training on one dataset and testing on the other.

Identical Signatures. For the WeMo Insight plug and Blink camera, the signatures extracted from the Mon(IoT)r dataset and our dataset (December 2019) were identical. Since the signatures obtained from our own dataset do not have any variations in packet lengths, we used PingPong's exact matching strategy (see Section 2.3.2) to detect events in the Mon(IoT)r dataset, and we observed a recall rate of 97% or higher for both devices (see Table 2.13).

Similar Signatures. For the TP-Link plug and Sengled light bulb, the signatures extracted from the Mon(IoT)r dataset are slightly different from those extracted from our own dataset: some packet lengths at certain positions in the sequence are different (by a few and up to tens of bytes), and these differences appear to be consistent (*i.e.*, all signatures from both datasets appear to be completely deterministic as they do not contain packet length ranges). For example, the TP-Link plug's ON event is `C-592 S-1234 S-100` in our experiment vs. `C-605 S-1213 S-100` in Mon(IoT)r. To understand the cause of these discrepancies, we examined the TP-Link plug in further detail—between the two devices, its signatures exhibit the largest difference in packet lengths across datasets. Through additional experiments on the TP-Link plug, we identified that changes to configuration parameters (*e.g.*, user credentials of different lengths) could cause the packet lengths to change. However, the packet lengths are deterministic for each particular set of user credentials.

For devices that exhibit this kind of behavior, an attacker must first train PingPong multiple times with different user credentials to determine to what extent these configuration changes affect the packet lengths in the signatures. Moreover, the signature matching strategy should not be exact, but must be relaxed to allow for small variations in packet lengths. To this end, we implemented *relaxed matching* that augments the matching strategies discussed in Section 2.3.2.[9] We ran PingPong with relaxed matching on the TP-Link plug with a delta of 21B, and successfully detected 100% of events. Furthermore, by performing the negative control experiments described in Section 2.4.3, we verified that the increase in FPR due to relaxed matching is negiligible. For dataset 1, relaxed matching results in two FPs for the Wi-Fi sniffer. For dataset 3, relaxed matching results in seven FPs for the Wi-Fi sniffer and one FP for the WAN sniffer. In comparison, exact matching only produces one false positive for the Wi-Fi sniffer for dataset 3. We note that the total number of packets across these datasets is 440 million. However, relaxed matching may eliminate the ability to distinguish event types for signatures that only differ by a few bytes (*e.g.*, the packet lengths for the TP-Link plug's ON and OFF signatures differ by one byte).

Signature Evolution. We observed that some signatures change over time, presumably due to changes to the device's communication protocol. The WeMo Insight plug's signature changed slightly from our earlier dataset from 2018 (see Table 2.3) to our latest dataset collected in December 2019 (see Table 2.13): the first PH-259 packet is no longer part of the signature. Both of these datasets were collected using the same testbed with the same

[9]In relaxed matching, a delta equal to the greatest variation observed in packet lengths is applied to the packets that vary due to configuration changes. For the TP-Link plug, we observed that the first packets differ by 13B in the the Device-Cloud signatures from the two datasets (*i.e.*, $13 = 605 - 592 = 606 - 593$) and the second packets differ by 21B (*i.e.*, $21 = 1234 - 1213 = 1235 - 1214$), thus a delta of 21B is used.

user accounts, but with different device firmware versions. Therefore, the change is probably due to changes in the communication protocol, introduced in firmware updates. This is further backed by the observation that the WeMo Insight plug's signature extracted from the Mon(IoT)r dataset (collected in April 2019) is identical to the signature extracted from our December 2019's dataset. This implies that there has been a protocol change between 2018 and April 2019, but the protocol has then remained unchanged until December 2019.

Similarly, the TP-Link light bulb's signature has changed slightly from our first to our second in-house dataset (see Tables 2.4 and 2.14), and is also slightly different for the Mon(IoT)r dataset.[10] The signatures from our 2018 dataset and those from the Mon(IoT)r dataset differ in the first packet (`PH-198` vs. `PH-258`, an offset of 60 bytes), and the signatures from the Mon(IoT)r dataset and those from our December 2019 differ in the second packet (`D-227` vs. `D-288` and `D-244` vs. `305`, an offset of 61 bytes). Thus, we also suspect that there is a signature evolution due to firmware updates for the TP-Link light bulb. Signature evolution is a limitation of our approach, and is elaborated on in Section 6.2. Nevertheless, an attacker can easily overcome this limitation simply by repeating PingPong's training to extract the latest signatures from a device right before launching an attack.

2.4.7 Parameters Selection and Sensitivity

Clustering Parameters. We empirically examined a range of values for the parameters of the DBSCAN algorithm. We tried all combinations of $\epsilon \in \{1, 2, ..., 10\}$ and `minPts` $\in \{30, 31, ..., 50\}$. For those devices that exhibit no variation in their signature related packet lengths, *e.g.*, the TP-Link plug, the output of the clustering remains stable for all values of ϵ and `minPts` < 50. For such devices, keeping ϵ at a minimum and `minPts` close to the number of events n reduces the number of noise points that become part of the resulting clusters. However, our experiments show that there is a tradeoff in applying strict bounds to

[10]We also repeated our experiments for the TP-Link light bulb to further understand this phenomenon.

devices with more variation in their packet lengths (*e.g.*, the D-Link plug), strict bounds can result in losing clusters that contain packet pairs related to events. For the D-Link plug, this happens if $\epsilon < 7$ and `minPts` > 47. In our experiments, we used our initial values of $\epsilon = 10$ and `minPts` $= 45$ (*i.e.*, `minPts` $= \lfloor n - 0.1n \rfloor$ with n = number of expected events) from our smart plugs experiment (*i.e.*, the TP-Link plug, D-Link plug, and SmartThings plug) that allowed PingPong to produce the packet-level signatures we initially observed manually (see Section 2.2.3). We then used them as default parameters for PingPong to analyze new devices and extracted packet-level signatures from 15 more devices.

Time Window and Signature Duration. We also measured the duration of our signatures—defined as the time between the first and the last packets of the signature. Tables 2.3, 2.4, 2.5 and 2.6 report all the results. The longest signature duration measured is 9,132 ms (less than 10 seconds) for the Sengled light bulb's ON/OFF signatures from the Phone-Cloud communication. This justifies our choice of training time window $t = 15$ seconds during trace filtering and signature validation (see Section 2.3.1). This conservative choice also provides slack to accommodate other devices that we have not evaluated and that may have signatures with a longer duration. This implies that events can be generated every 15 seconds or longer. We conservatively chose this duration to be 131 seconds to give ample time for an event to finish, and to easily separate false positives from true positives.

2.5 Possible Defenses against Packet-Level Signatures

In this section, we discuss possible defenses against passive inference attacks such as packet-level signatures. There are several broad approaches that can obfuscate network traffic to defend against passive inference attacks that analyze network traffic metadata:

1) *Packet padding* adds dummy bytes to each packet to confuse inference techniques that rely on individual packet lengths, and less so volume. Packets can be padded to a fixed length (*e.g.*, MTU) or with a random number of bytes.

2) *Traffic shaping* purposely delays packets to confuse inference techniques that rely on packet inter-arrival times and volume over time.

3) *Traffic injection* adds dummy packets in patterns that look similar (*e.g.*, have the same lengths, inter-arrival times or volume signature *etc.*) as the real events, thus hiding the real event traffic in a crowd of fake events.

The above approaches can be implemented in different ways and can also be combined (*e.g.*, on the same VPN). Since our signatures rely on unique sequences of individual packet lengths, packet padding is the most natural defense and therefore discussed in depth below. We provide a brief overview of packet padding in the literature in Chapter 5.

In the next sections, we discuss packet padding more thoroughly. We first discuss how packet padding may be implemented to obfuscate packet-level signatures. Then, we evaluate the efficacy of packet padding for the TP-Link plug. Finally, we discuss traffic shaping and traffic injection in Section 2.5.5.

2.5.1 Possible Implementations

Next, we discuss the potential benefits and drawbacks of different padding implementations. We consider a VPN-based implementation, padding at the application layer, and TLS-based padding.

VPN. One option is to route traffic from the smart home devices and the smartphone through a VPN that pads outbound tunneled packets with dummy bytes and strips the padding off of inbound tunneled packets: a technique also considered in [40]. The smart home end of the VPN may be implemented either directly on each device and smartphone or on a middlebox, *e.g.*, the home router. The former provides protection against *both* the WAN and Wi-Fi sniffers as the padding is preserved on the local wireless link, whereas the latter only defends against the WAN sniffer. However, an on-device VPN may be impractical on devices with limited software stacks and/or computational resources. The middlebox-based approach may be used to patch existing devices without changes to their software. Pinheiro et al. [160] provide an implementation in which the router is the client-side end of the VPN, and where the padding is added to the Ethernet trailer.

Application Layer and TLS. Another option is to perform the padding at the application layer. This has at least three benefits: (1) it preserves the padding across all links, thus provides protection against both the WAN and Wi-Fi sniffers; (2) it imposes no work on the end user to configure their router to use a VPN; and (3) it can be implemented entirely in software. An example is HTTPOS by Luo et al. [136], which randomizes the lengths of HTTP requests (*e.g.*, by adding superfluous data to the HTTP header). One drawback of application layer padding is that it imposes extra work on the application developer. This may be addressed by including the padding mechanism in libraries for standardized protocols (*e.g.*, OkHttp [187]), but a separate implementation is still required for every proprietary protocol. A better alternative is to add the padding between the network and application layers. This preserves the benefits of application layer padding highlighted above, but eliminates the need for the application developer to handle padding. As suggested in [77], one can use the padding functionality that is already available in TLS [167].

2.5.2 Residual Side-Channel Information

Even after packet padding is applied, there may still be other side-channels, *e.g.*, timing and packet directions, and/or coarse-grained features such as total volume, total number of packets, and burstiness, as demonstrated by [77]. Fortunately, timing information (*e.g.*, packet inter-arrival times and duration of the entire packet exchange) is highly location dependent (see the comparison of signature durations in Tables 2.13 and 2.14), as it is impacted by the propagation delay between the home and the cloud, as well as the queuing and transmission delays on individual links on this path. Exploiting timing information requires a much more powerful adversary: one that uses data obtained from a location close to the smart home under attack. The work of Apthorpe *et al.* on traffic shaping [43] and stochastic traffic padding (STP) [40] may aid in obfuscating timing, volume, and burstiness.

2.5.3 Efficacy of Packet Padding

The discussion has been qualitative so far. Next, we perform a simple test to empirically assess the efficacy of packet padding for the TP-Link plug.

Setup. We simulated padding to the MTU by post-processing the TP-Link plug testbed trace from Section 2.4.2 (50 ON and 50 OFF events, mixed with background traffic) using a simplified version of PingPong's detection that only considers the order and directions of packets, but pays no attention to the packet lengths. We focus on the WAN sniffer because it is the most powerful adversary: it can separate traffic into individual TCP connections and eliminate the confusion that arises from multiplexing. We used the TP-Link plug's two-packet signatures for ON and OFF events (see Table 2.3) as the Phone-Device communication is not visible on the WAN. We consider the packet padding's impact on transmission and processing delays to be negligible. We assume that the adversary uses *all* available information to filter out irrelevant traffic. Specifically, the adversary uses the timing information observed during training to only consider request-reply exchanges that comply with the signature duration.[11] Moreover, since the TP-Link plug uses TLSv1.2 (which does not encrypt the SNI), the adversary can filter the trace to only consider TLS Application Data packets to the relevant TP-Link host(s) in the no-VPN scenarios.

VPN-Based Padding. To simulate VPN-based packet padding, we consider all packets in the trace as multiplexed over a single connection and perform signature detection on this tunnel. This results in a total of 193,338 positives, or, put differently, more than 1,900 false

[11] $t = 0.204\text{s} \implies \lceil 0.204 + 0.1 \times 0.204\text{s} \rceil = 0.224\text{s}$ (see Table 2.3 and Section 2.3.2)

positives for every event. This demonstrates that VPN-based packet padding works well for devices with short signatures (*e.g.*, a single packet pair).

TLS-Based Padding. From the training data, the adversary knows that the signature is present in the TP-Link plug's communication with `events.tplinkra.com`. To simulate TLS-based packet padding, we performed signature detection on the TLS Application Data packets of each individual TLSv1.2 connection with said host. As expected, this produced a total of 100 detected events, with no FPs. Intuitively, this is because the only TLS Application Data packets of these connections are exactly the two signature packets, and the device only communicates with this domain when an event occurs.

Hybrid. We next explore how multiplexing all of the TP-Link plug's traffic over a single connection affects the false positives (the plug communicates with other TP-Link hosts).[12] This is conceptually similar to a VPN, but only tunnels application layer protocols and can be implemented in user space (without TUN/TAP support). To simulate such a setup, we filtered the trace to only contain IPv4 unicast traffic to/from the TP-Link plug, and dropped all packets that were not TLS Application Data. We then performed detection on the TLS Application Data packets, treating them as belonging to a single TLS connection. For this scenario, we observed 171 positives. While significantly better than TLS-based padding for individual TLS connections, the attacker still has a high probability (more than 50%) of guessing the occurrence of each event (but cannot distinguish ON from OFF).

[12]We envision that this could be implemented by maintaining a single TLS connection between the device and a single TP-Link endpoint, T, that would then carry all application layer messages, each prepended with an additional header that identifies the type and order of that particular request/response, and padded to MTU using TLS record padding. For each request, T would interpret its type based on the application layer header, and forward it to an appropriate backing server responsible for that part of the application logic (*i.e.*, T is analogous to a load balancer).

2.5.4 Recommendations for Padding

Based on the above insights, we recommend VPN-based packet padding due to its additional obfuscation (encryption of the Internet endpoint and multiplexing of IoT traffic with other traffic) as TLS-based padding seems insufficient for devices with simple signatures and little background traffic. For more chatty devices, multiplexing all device traffic over a single TLS connection to a single server may provide sufficient obfuscation at little overhead.

2.5.5 Traffic Shaping and Injection

Stochastic traffic padding (STP)[13] is a state-of-the-art defense for passive inference attack on smart home devices [40].[14] STP shapes real network traffic generated by IoT events and injects fake event traffic randomly into upload and download traffic to imitate volume-based signatures. We contacted the authors of [40], but they did not share their STP implementation. Therefore, we simulated the VPN-based STP implementation, and performed a simple but conservative test. We used OpenVPN to replay packets from our pre-recorded smart home device events. Alongside 100 real events, we injected 100 dummy STP events of the same type distributed evenly and randomly throughout the experiment. Our experiments with OpenVPN reveal that it consistently adds a header of 52 bytes for client-to-server packets and 49 bytes for server-to-client packets: thus our signatures remain intact. However, all traffic is now combined into one flow between two endpoints, and this could potentially increase the FPs.

In our STP experiments, PingPong performs well and STP has very little effect on our signatures: it does not generate many FPs: the average recall remains around 97% and the

[13]It is worth clarifying that despite its name, STP does not perform packet padding. It performs traffic shaping and injection (referred to as "traffic padding" in [40]) of fake traffic that resembles the real IoT traffic.

[14]Other examples of traffic injection include [158] that injects fake/spurious HTTP requests to defend against website fingerprinting.

FPR increases to 1.09 per 100 dummy events of the same event type: the FPR increase is minimal. Tables 2.3, 2.4, 2.5, and 2.6 report additional false positives for 3 devices: Arlo camera, Nest thermostat, and TP-Link light bulb. Further inspection revealed that these FPs were caused only by 2-packet signatures, while longer signatures were more resilient. The small increase occurs for two reasons: (1) the VPN tunnel combines all traffic into one flow, and (2) the dummy event packets are sometimes coincidentally sent from both end-points simultaneously, allowing the request and reply packets to be in the same signature duration window (see Section 2.4.7). Thus, a VPN-based STP implementation is not effective in defending against our PingPong; this is expected as STP was designed to defend against volume-based signatures, while our signatures consist of sequences of packet lengths, which survive both traffic shaping and injection. Furthermore, this defense is currently only applicable against the WAN sniffer adversary, while the Wi-Fi sniffer is not affected by this router-based implementations of STP.

Chapter 3

Securing Smart Home Edge Computing

In this chapter, we present our findings on smart home platforms vulnerabilities (in the context of SmartThings), threat model and guarantees. We also present Vigilia, a system that we developed to harden smart home platforms, and its evaluation.

3.1 Smart Home System Vulnerabilities

The SmartThings platform has the following vulnerabilities [70, 89, 103, 153, 200, 93, 131, 86]:

1) **Device Vulnerabilities:** Many IoT devices connect directly to the home Internet connection and communicate with the hub via the LAN or the cloud. Many of these devices either intentionally trust communication from the local area network (*e.g.*, WeMo, LiFX), use inadequate authentication mechanisms (*e.g.*, a short PIN in the case of D-Link), or have backdoors (*e.g.*, Blossom sprinkler) that make them vulnerable to attack.
2) **Trusted Codebases with Bad Security Records:** The SmartThings system executes device drivers and applications on a *Java Virtual Machine* (JVM)—recall that Groovy is a managed language that is running on top of the JVM, and relies on the JVM to provide safety. Bugs in the JVM could potentially allow applications to subvert the capability system and access arbitrary devices.

3) **Excessive Access Granted to Cloud Servers:** The SmartThings system executes most applications and device handlers on their cloud servers and uses the hub to relay commands to the local devices. The hub punches through the home firewall to give the SmartThings cloud servers arbitrary access to communicate with any local device. Note that while compromised firmware updates could conceptually be used to obtain similar access, the scenarios are fundamentally different because firmware updates are often signed. Thus, with appropriate key protection mechanisms, they can be made difficult for attackers to compromise.
4) **Excessive Access Granted to Device Handlers or SmartApps:** SmartThings device handlers have the ability to capture all SSDP network traffic to the hub [115], communicate with arbitrary IP addresses and ports by reconfiguring the device's network address, and send arbitrary commands to arbitrary Zigbee devices [86].

When a homeowner purchases a new IoT device, they first make it available to their SmartThings hub. SmartThings provides drivers for a wide range of third party devices; users can also write their own drivers or import third-party driver code. Some popular devices such as the Nest thermostat can only be integrated into SmartThings via third-party drivers that are not subject to any code review process.

When a SmartApp is first installed, the user configures it by selecting the devices to be monitored and controlled. This process grants the SmartApp the capabilities to access those devices. While the SmartThings capability system appears at first glance to provide strong security assurances, it can be easily subverted. For example, a SmartApp can conspire with a device handler to subscribe to all SSDP traffic to the hub, open arbitrary connections to cloud servers, or obtain arbitrary access to LAN and Zigbee devices.

3.2 Vigilia Approach

To overcome the aforementioned vulnerabilities, we propose the Vigilia approach. Our initial goal was to implement this novel approach in SmartThings. However, SmartThings is closed source—we could not directly enhance it as we do not have access to its source code. As a result, we had to develop a new distributed IoT infrastructure that closely follows the programming and computation model of SmartThings. We demonstrate the viability of our approach by implementing Vigilia on top of this new system. Our idea is generally applicable to SmartThings and any other smart home IoT infrastructure that uses similar models.

First, Vigilia restricts network access—Vigilia uses a similar programming model as SmartThings but leverages the configuration information that is already available to also restrict network access. Vigilia makes the network primarily responsible for the security of IoT devices—Vigilia implements a default deny policy for all IoT devices and smart home applications. Access is only granted when user has explicitly configured a smart home application to use a specific device. A key advantage of this approach is that it becomes less critical that end users keep every IoT device fully patched. At the same time, by leveraging the configuration information that is already present, Vigilia's security mechanisms never get in the way of legitimate computations.

Second, Vigilia provides more fine-grained access control to specific devices. In Vigilia, a smart home application controls a specific device via a device driver. The interaction between the smart home application and the device driver occurs through *remote method invocation* (RMI). Device features are exposed as API methods in the device's driver class. This is implemented as *capability-based RMI* that only allows a limited set of API methods to be called depending on the configuration. Thus, this mechanism provides more fine-grained access control to specific devices on top of the network policy restrictions.

3.3 Threat Model and Guarantees

Vigilia protects IoT devices from attacks resulting from overprivileged network access. We use the following threat model: 1) the IoT devices have *vulnerabilities*, 2) attackers have *full knowledge* of Vigilia, and 3) attackers have *access* to the home network via a *compromised* laptop or device, *not* physical access.

Our threat model is *stronger* than those assumed in the existing IoT-defense systems that we are aware of. Commercial systems [79, 78] typically assume that threats come from the outside network and the home network is well-guarded. In the research community, systems such as HomeOS and HanGuard [73, 68] assume that attacks can come from the home network, but they focus on PC and smartphone apps vulnerabilities. IoTSec [184] mainly safeguards against arbitrary port accesses. Our comparison (see Section 3.8.2) between Vigilia, and commercial and research systems demonstrates that the threat model we use enables stronger protection of IoT devices than these systems.

We do *not* trust application developers—Vigilia ensures that applications can only perform network, Zigbee, and file accesses allowed by the user configuration. We do *not* assume that application processes are trusted. The attacker may tamper with the source/binary code of the IoT program or the language runtime such as the JVM, *e.g.*, exposing device driver objects to applications that are not supposed to access those devices. In such cases, the unspecified communications will be blocked by the Wi-Fi router or the Zigbee gateway—we trust the integrity of the Vigilia Wi-Fi router and the Zigbee gateway. We do *not* trust the wireless stack of any smart home device. This includes not trusting devices to use the assigned MAC or IP addresses.

We assume a partial trust of the OSs (*i.e.*, TOMOYO Linux [189]) running on Raspberry Pi nodes. Defending against attacks on the OS is out of scope. Note that even if the OS is

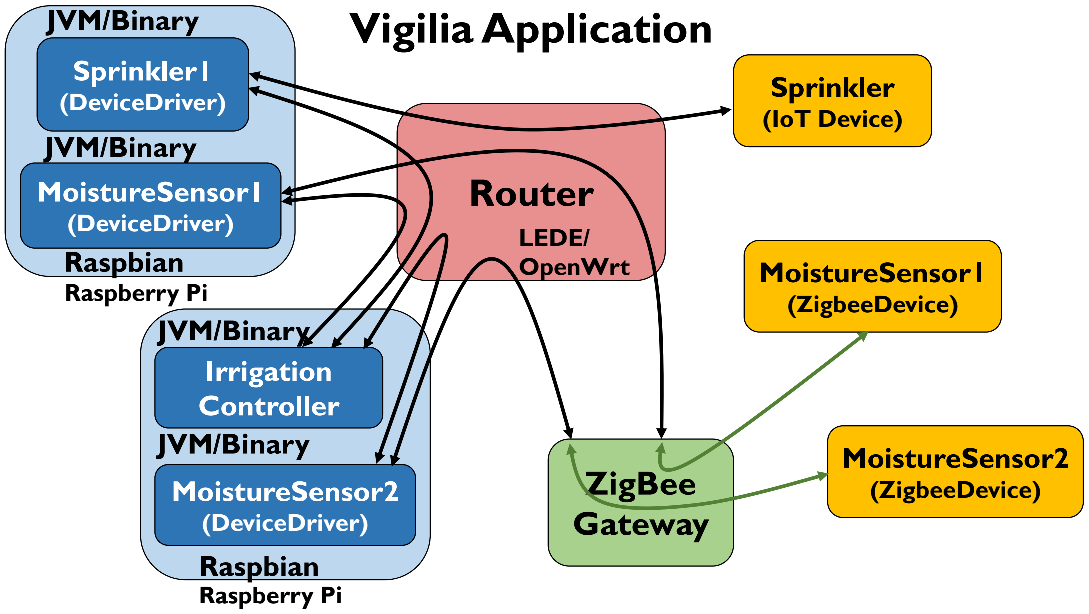


Figure 3.1: A closer examination of an irrigation system.

compromised, the attacker can only obtain the permissions of other Vigilia components on the same device as the router enforces inter-device permissions.

Vigilia provides the following *guarantees*: (1) all communications that are *not* explicitly configured by the user with a Vigilia component or IoT device are blocked; (2) a Vigilia component is only allowed to perform actions permitted by the capabilities it is granted; (3) smart home applications developed by honest developers will never be blocked by Vigilia's checks.

3.4 Example

Figure 3.1 presents a block diagram of an example smart irrigation system that connects a set of IoT devices, Raspberry Pis, Zigbee gateways, and Zigbee devices with a router. An *IoT device* is a smart home device connected to the Wi-Fi network such as a sprinkler. Similar to the SmartThings system, each device has a *device driver* that interfaces between the device and smart home applications. The device driver runs on a Raspberry Pi running Raspbian. Similar to SmartThings, we expect that device drivers will be written either by the device manufacturer, Vigilia developers, or third-party hobbyists. For standard classes of devices, we expect that the Vigilia developers would define standardized APIs to support compatibility much like SmartThings ecosystem. For each smart home application, there is an *application* that interacts with the drivers of the involved devices to achieve certain smart home functionalities. In our example, the application talks to a set of moisture sensors (discussed shortly) to measure soil moisture, which will be used to adjust the irrigation schedule for the sprinkler. The application thus needs to communicate with the drivers of the sprinkler and the moisture sensors.

One significant difference between SmartThings and Vigilia is that SmartThings components (*e.g.*, apps and device handlers) typically run on the SmartThings cloud, whereas Vigilia runs its components entirely on local compute nodes. This has significant advantages in that Vigilia applications can operate even if Internet connectivity is lost. The application also runs on a Raspberry Pi, which may or may not be the same one that hosts the drivers. A smart home system often has multiple applications and thus multiple applications may exist simultaneously. The drivers and the application may be developed by different developers and/or in different languages. For example, if they are written in Java, they are executed by JVMs; if they are C++ programs, their binary code is directly executed. In this paper, we refer to device drivers or applications as *components*.

Zigbee is a standard communication protocol that connects devices with small, low-power radios. A smart home system may also contain Zigbee devices that connect to the home Wi-Fi through a Zigbee gateway. In this case, the Zigbee gateway has an IP address from the LAN while the Zigbee devices do not support TCP/IP and only have Zigbee addresses. Hence, device drivers must communicate with Zigbee devices via requests made to the Zigbee gateway.

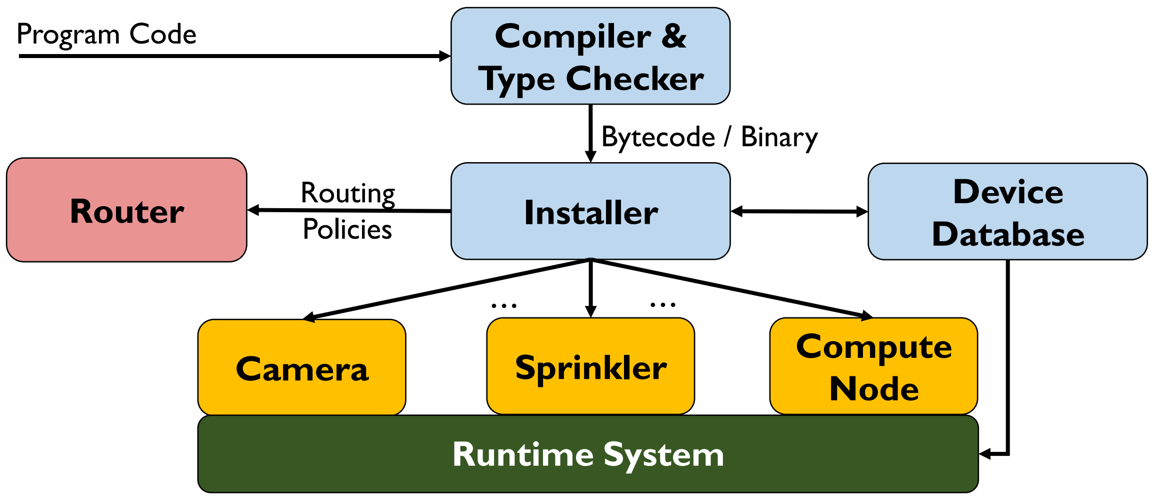


Figure 3.2: Vigilia system architecture.

3.5 Architecture and Programming Model

Figure 3.2 depicts Vigilia's architecture. Vigilia implements key components of the SmartThings programming model and system architecture to ensure that our techniques are applicable to real smart home systems.

Applications are compiled using the Vigilia tool chain. The tool chain checks that applications will never violate the declared permissions at runtime. Applications are then deployed using the Vigilia installer. The deployment process involves the end user specifying how the application should be configured for the given house. For example, this process might specify which switches and light bulbs an application has access to, and which switches should control which light bulbs. The Vigilia installer then computes a set of permissions that is required for the given installation. Finally, the Vigilia runtime enforces these permissions.

To enable applications to fully realizing the potential benefits of smart home systems, systems like SmartThings implement and expose rich APIs—potentially increasing their attack surface. Complex interactions among different devices requires a programming framework that makes it easy for components to interact when desired while at the same time blocking undesired interactions. Like SmartThings, Vigilia users implicitly grant permissions to a smart home application when they configure the application to implement the desired functionality. The permissions required are partly determined by the application's intended function—thus, some information about the nature of the permissions required by smart home applications must be specified by the developer. However, the developer does not know the specifics of a given deployment. For example, the developer would typically not know how many light bulbs or cameras an end user has installed (or what rooms these devices are installed in). Instead, developers only have a high-level view of which type of device the application needs and the required relationships between devices (*e.g.*, that they are in the same room).

Like SmartThings, Vigilia employs an object-oriented component model. Each device driver or smart home application has a corresponding *class*. Vigilia classes can declare *sets* and *relations*. These sets and relations are declared as data fields in these classes. Sets represent abstract communication permissions. In the SmartThings programming model, the same information is specified using the `preferences` keyword. Vigilia extends the SmartThings programming model by using RMI to both isolate components and to support distributed applications. Communication with devices or other smart home applications are implemented using *remote method invocation* or RMI. As IoT systems may contain components written in different languages, Vigilia provides cross-language support for RMI. Vigilia contains a RMI compiler that parses policy files defining the capabilities of a component to generate code that implements the RMI stubs and skeletons.

Irrigation Application Code. To better explain the programming model, we show a code example for a smart irrigation application. Figure 3.3 presents Java code for the example. In this example, the application communicates with a set of sprinklers to water the lawn and a set of moisture sensors to monitor soil moisture. The `IrrigationController` class

```java
1    public class IrrigationController extends
2      Application implements Irrigation {
3    @config Set<Sprinkler> sprinklers;
4    @config Set<MoistureSensor>
5      moisturesensors;
6    @config Relation<MoistureSensor,
7      Sprinkler> sensortosprinklers;
8    @config Set<Gateway> phone;
9    @config Set<Address> weatherforecast;
10   //Interface method containing initialization logic
11   public void init() {
12     ...
13   }
14   //Other computation methods
15   private void turnOn() {
16     ...
17   }
18 }
```

Figure 3.3: Example application code in Java.

implements the smart irrigation application. The irrigation application uses information from several moisture sensors to adjust watering schedules and thus must communicate with the moisture sensors. Each application class extends the Vigilia `Application` class and implements the `init` method. This method will be invoked by the Vigilia runtime during application startup.

Abstract Permissions. In general, developers only know which types of devices an application needs to communicate with; the exact device instances in each class are specified during the site-specific installation process. Vigilia provides the developer with an *abstract permission* model to specify the permissions required by a given application. These abstract permissions are specified in terms of members of sets (similar to SmartThings preferences). The Vigilia installer (like the SmartThings installer) then *instantiates* these permissions by specifying the exact members of the sets.

For example in Line 4 of Figure 3.3, a developer specifies an abstract permission that allows communication between the application and the generic type of moisture sensor by declaring

`@config Set<MoistureSensor> moisturesensors` in the `IrrigationControl- ler` application class. This declares that the application has the abstract permission that allows it to talk to moisture sensors at runtime. The Vigilia programming model uses annotations either in the code (Java) or in a separate file (C++) to allow the developer to express this information.

In the above example, the developer does not need to worry about how to create the set object and the contained `MoistureSensor` objects in the program as these objects are created by the runtime system. For example, if the end user configures two moisture sensors for the application, then the Vigilia runtime would create two `MoistureSensor` objects and insert both objects into the `moisturesensors` set. When the program is executed, the Vigilia runtime system initializes this set with references to the appropriate sensor objects. Vigilia components such as applications and device drivers run in separate processes (*i.e.*, JVM/binary). Since communication between components is implemented via RMI, a reference from the `moisturesensors` set can be used to directly communicate with the sensor. Components in Vigilia can only communicate with other components that are specified by this set-based model.

Application Installation. During the installation process, the end user configures the application for their home. This configuration process is not unique to Vigilia, most smart home systems include a similar process in which the end user must specify which devices should be controlled and how they should be. Moreover, the Vigilia installation process for an application is similar to SmartThings. The Vigilia installer asks the end user to configure the concrete device instances to be used by an application. For example, a sprinkler controller may ask which *moisture sensors* should be used to monitor soil moisture. The end user specifies which specific moisture sensors the application should use by defining the devices that comprise the set of moisture sensors. Finally, the Vigilia installer uses abstract permissions and user configuration to generate concrete permissions. Abstract permissions

```
class SpruceSensor : public Device,
                     public MoistureSensor {
   private:
      Set<ZigbeeAddress*> sprucesensor;
      Set<DeviceAddress*> zigbeegateway;
      double moisture;
      double temp;
   public:
      void init();
      double getMoisture();
      double getTemperature();
}
```

Figure 3.4: Example device driver header in C++.

are generic for the application, while concrete permissions are specific to installations and grant access to physical devices.

Vigilia extends the set-based model with *relations*, specifying relations between devices and communicating configuration information. For our irrigation example, the application must know which sprinklers are located near which moisture sensors. During installation, the user provides this information in relations as it is specific to their installation. Line 6 of Figure 3.3 declares the `sensortosprinklers` relation that maps moisture sensors to the nearby sprinklers. Similar to sets, relation objects are also constructed by the runtime system.

Communication. Line 8 of Figure 3.3 declares a set of gateways for smartphones. Devices like tablets/smartphones/laptops can be used to provide a user interface, through which users can input application parameters. Finally, Line 9 declares a set of addresses of cloud-based servers that provide weather forecast information. Vigilia uses an oblivious cloud-based key-value store to provide secure storage and communication even in the presence of malicious cloud servers.

Device Drivers. Figure 3.4 presents a device driver class declaration in C++ for the moisture sensor used by our irrigation example. Our irrigation example uses a Spruce moisture sensor [185], which is a Zigbee-based wireless sensor. To communicate with the sensor, the device driver must send packets to the sensor via a Zigbee gateway. Thus the driver needs two addresses: (1) the IP address for the Zigbee gateway and (2) the Zigbee network address for the Spruce sensor.

Device drivers use the same set-based mechanism to obtain direct access to network-based devices. The installation process stores the system configuration parameterized by the devices' MAC addresses, and the Vigilia runtime maps the MAC addresses to the corresponding IP addresses. Network access is only permitted via runtime provided IP address/port pairs, and thus the Vigilia runtime knows which devices a driver may communicate with. The Vigilia runtime uses this information to configure the routing policies. Device drivers may declare a set of public methods such as `getMoisture` for the application to get/set information from/to the device.

3.6 Vigilia Security Mechanisms

We next discuss the security mechanisms Vigilia implements for the programming model.

Checking. One challenge is *how to statically eliminate permission bugs*, in which an application accidentally exceeds its declared permissions and thus fails at runtime when the Vigilia runtime enforcement framework blocks the illegal access. The Vigilia static checking framework is designed to help honest developers ensure that their applications never fail at runtime because of Vigilia's runtime enforcement framework. *It is important to note that Vigilia does not rely on the static checks for security—applications that attempt to violate their permissions will be blocked by runtime checks.* The static checker needs to notify the developer of any network accesses that are doomed to be blocked by runtime checks. For example, an application could potentially violate its permissions if it were to obtain a reference to a device object from some other component and then attempt to use that reference to access the underlying device. Such an access would fail at runtime and potentially cause the application to crash.

Vigilia supports both Java and C++. One goal of Vigilia is to make it easy to support new languages and thus we minimize the dependence on specialized compiler passes for static checking. To the degree possible Vigilia uses the existing language type system to check for permission violations. Vigilia implements these checks via the Vigilia RMI compiler. The Vigilia RMI compiler uses the declared types to ensure that the existing language type system will catch any accidental sharing of references to device objects by an application.

SmartThings applications have full Internet access. A malicious app can easily leak private information. Internet access may also provide a conduit to attack benign applications. On the other hand, some functionality requires Internet access to implement. Thus, Vigilia supports managed access to TCP/IP sockets. This ensures that Vigilia is aware of any potential TCP/IP accesses. If a program were to attempt other accesses, they would be blocked by the Vigilia enforcement framework. The Java implementation of Vigilia's checker uses a type checker to ensure that Java Vigilia applications do not attempt to directly use raw TCP/IP sockets for communication. The C++ implementation does not implement this particular check—note that this does not impact security, but developers could attempt direct network accesses that would be blocked at runtime.

Vigilia Installer. The Vigilia installer manages the installation of new devices and smart home applications. A major issue with the SmartThings system is that it trusts that devices on the home network are not malicious. Under SmartThings, a single malicious device on the home network has full network access to all other devices. Vigilia fully isolates each IoT device from every other device on the network, permitting communication only when applications are explicitly configured to use a device during the installation process. When a new device is installed, Vigilia must update its database to include a record of the device's MAC address and type. To prevent MAC address spoofing or sniffing attacks from circumventing Vigilia's access control, Vigilia assigns a unique *pre-shared key* (PSK) to each device. The Vigilia router ties each unique PSK to a specific device MAC address. Note that while some Android and iOS devices implement MAC randomization, it is used only when probing for wireless networks. Thus, our approach is compatible with modern smart phones. Finally, the installer maps the device to a specific driver.

The Vigilia installer also manages the addition of new smart home applications. Installing a new smart home application requires specifying the device instances that the smart home application can control. For each type of abstract permission the smart home application has requested, the Vigilia installer presents the list of devices that could provide those capabilities. The user then selects the subset of devices she wishes the application to use. For relations, the user specifies the pairs that comprise the relation (*e.g.*, that a moisture sensor is close to a given sprinkler head).

Enforcement. Vigilia implements its security model by combining a range of known techniques. It begins with a modified wireless router based on LEDE—now merged with OpenWrt [155]. Many commercially available routers are built using a similar core code base, so it should be relatively straightforward to modify existing routers to implement the necessary functionality. The Vigilia router allows wireless devices on the same wireless network to have different PSKs. This allows the router to prevent both MAC spoofing and sniffing attacks: Vigilia can trust the MAC address of a device and that the wireless communications between the router and other devices are secure. Vigilia then uses firewall rules to prevent IP spoofing so that it can trust IP addresses.

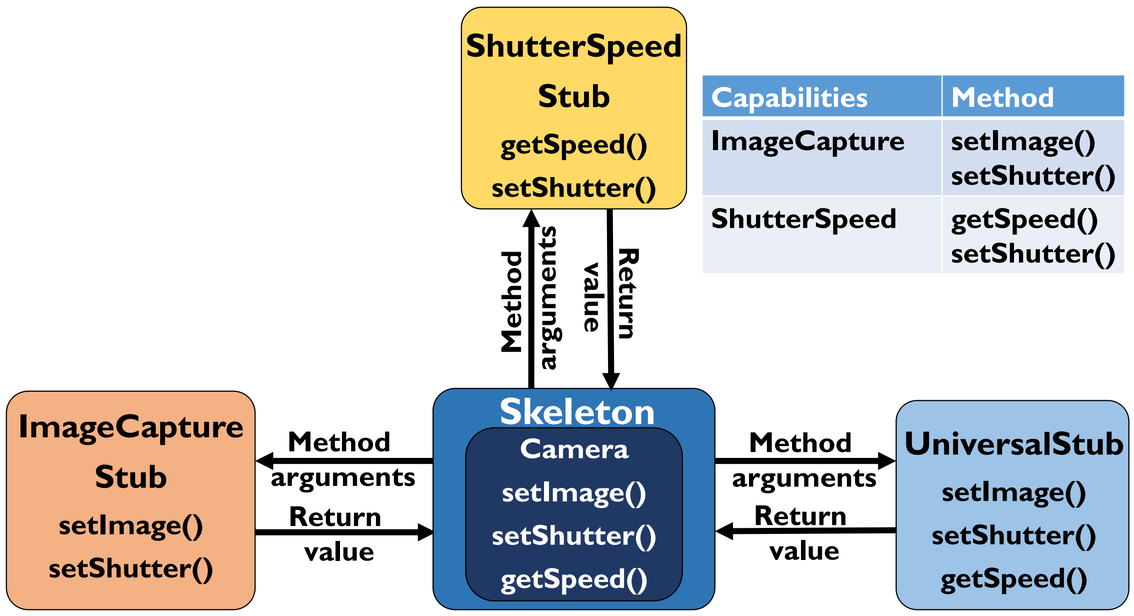
Compute nodes can run more than one computation and these computations may have different permissions. Vigilia assigns different ports to different computations on the same node so that other devices can identify a communication's source. Vigilia sandboxes client code using TOMOYO Linux to ensure that client processes cannot fake port numbers. TOMOYO Linux also ensures that processes do not access the files of other processes.

Vigilia implements concrete permission checks by translating each access permission into a corresponding firewall rule. Vigilia's *default* policy is to *block communications—e.g., unused smart home devices are not allowed to communicate with anything.*

So far we have only discussed restricting network accesses. However, devices may have many features (*e.g.*, read temperature and set temperature), and it is important to restrict accesses to only the necessary features. To support restrictive feature access, Vigilia employs a *capability-based RMI*—device features are often exposed as API methods in the device's driver class and thus accessing device features is often done through remote invocations on the corresponding methods. A *capability* in Vigilia is a device feature that consists of a set of methods from its corresponding class. A component can declare multiple capabilities and the capabilities can contain overlapping methods.

Components declare the capabilities they require from other components. The RMI compiler uses these policy files to generate stubs and skeletons that only provide access to the declared capabilities. Although Vigilia's security guarantees for capabilities are enforced dynamically, this code generation strategy enables the existing C++ or Java compiler to statically check that a component does not exceed its declared capabilities. This ensures that a well-behaved component will never fail a runtime security check.

Figure 3.5 shows an interaction between a `Camera` object and the stubs generated from the original `Camera` interface. The `Camera` interface has two capabilities, namely `ImageCapture` and `ShutterSpeed`. Each of these capabilities has two methods and three stubs (`ShutterSpeed-Stub`, `ImageCaptureStub`, and `UniversalStub`) are generated based on each combination of the capabilities. The skeleton supports all the methods. Vigilia's capability-based model is complementary to firewall rules—while firewall rules restrict communications between components, the capability-based model restricts method invocations by component.

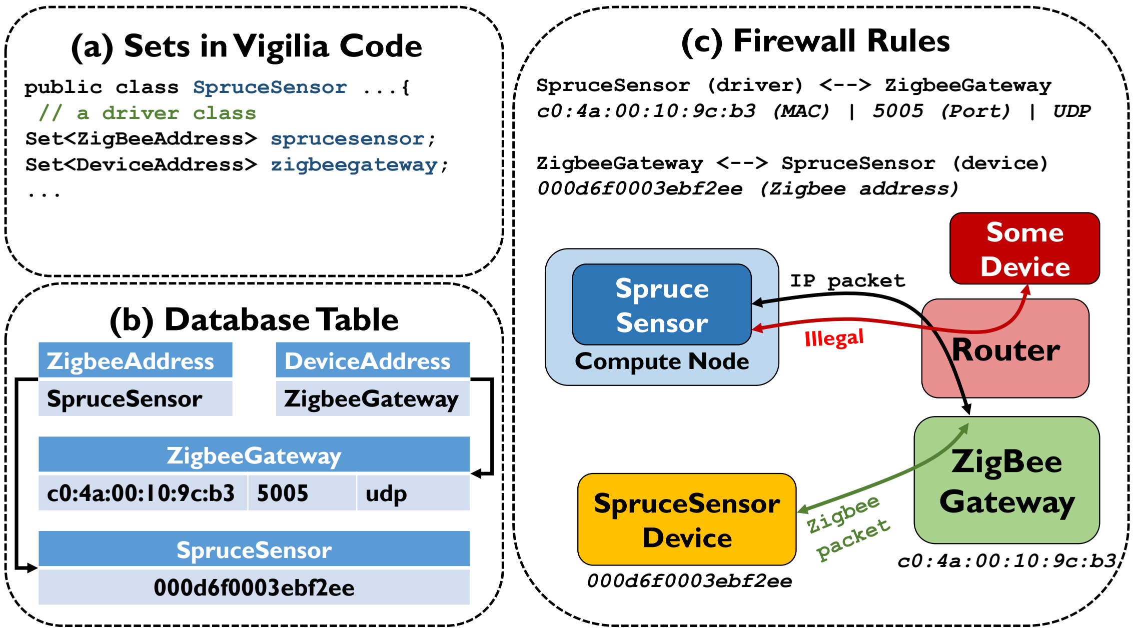


Capabilities	Method
ImageCapture	setImage() setShutter()
ShutterSpeed	getSpeed() setShutter()

Figure 3.5: Capability-based RMI example.

This means the problem of restricting feature accesses can be reduced to restricting remote method invocations. Vigilia enforces capabilities by using request filters in its RMI request server—these filters are automatically configured by Vigilia, and use the source port and IP address to determine whether a given request is allowed.

Figure 3.6 shows the relationship between the programming model, Vigilia's configuration database, and the firewall rules. The developer specifies that the Spruce sensor driver communicates with the Spruce sensor. Since the driver runs on a Raspberry Pi while the Spruce sensor is a Zigbee device that needs to communicate via a Zigbee gateway, the developer adds a second set that enables the driver to obtain a reference to appropriate the Zigbee gateway. These two abstract permissions have two separate effects. They mean that the code can only communicate with the Zigbee gateway specified by the `DeviceAddress` object and can only communicate with the `ZigbeeAddress` for the Spruce sensor. These abstract permissions will be concretized into concrete permissions at installation, which, together with the network configuration in the device database (Figure 3.6(b)), will be used by Vigilia to



```
public class SpruceSensor ...{
 // a driver class
Set<ZigBeeAddress> sprucesensor;
Set<DeviceAddress> zigbeegateway;
...
```

ZigbeeAddress	DeviceAddress
SpruceSensor	ZigbeeGateway

ZigbeeGateway		
c0:4a:00:10:9c:b3	5005	udp

SpruceSensor
000d6f0003ebf2ee

```
SpruceSensor (driver) <--> ZigbeeGateway
c0:4a:00:10:9c:b3 (MAC) | 5005 (Port) | UDP

ZigbeeGateway <--> SpruceSensor (device)
000d6f0003ebf2ee (Zigbee address)
```

Figure 3.6: Vigilia program (*i.e.*, irrigation system) (a), device database (b), and instantiated firewall rules (c).

generate the firewall rules for the router (Figure 3.6(c)). As a result, the router will block any communication inconsistent with these permissions.

3.7 Vigilia Runtime System

The Vigilia runtime system is a distributed system with a master and several slaves.

Startup. The master manages the application startup process. The master generates a deployment plan for an application, configures the appropriate firewall rules for both the router and every compute node, and then sends requests to slave processes to start up the components. Each component is started inside of a sandbox that constrains the component to the specified ports.

Wireless Network Filtering. In the default configuration, a standard firewall will not filter traffic between devices on the same wireless network as the traffic never passes through the firewall. Access points typically offer two modes of operation: the standard mode, which forwards all traffic between clients, and the client isolation mode, which blocks all traffic between clients. However, the Linux kernel firewall can be configured to filter these packets. This is implemented by: (1) enabling access point isolation, (2) turning on bridge hairpin mode (also called ‘reflective relay’) for the wireless LAN interface to force the traffic through the kernel firewall, and (3) then using iptables to filter the traffic.

Vigilia modifies the Wi-Fi stack to secure it against network-level attacks such as snooping, ARP-spoofing, and MAC-spoofing that would otherwise subvert Vigilia. Most IoT devices only support the pre-shared key (PSK) mode of WPA/WPA2 and do not support WPA/WPA2 Enterprise mode. This introduces a potential attack—even though each device eventually negotiates its own key, in the pre-shared key mode all devices on the same network know the same initial shared key. Any device that knows the pre-shared key and monitors the key negotiation can extract the private key.

Surprisingly, it turns out that it is possible to assign a unique PSK to each MAC address without breaking the WPA/WPA2 protocol. This prevents devices from computing the private keys of other devices, ensuring that malicious devices cannot masquerade as the router. This approach also effectively locks a physical device to a specific MAC address—malicious devices cannot spoof the MAC addresses of other devices as they do not know the MAC-specific PSK. The Vigilia router also enforces that MAC addresses are locked to the specific assigned IP address—any spoofed traffic is dropped.

Vigilia uses an Android app to configure new devices on the network. The app generates a new PSK and sends the PSK to the router using ssh. The router then changes the default password for the network to this PSK to allow the new device to join the network. It then detects the MAC address of the new device, adds the MAC address-PSK pair to its database, and reverts to the default PSK.

The shared group key, which is used for broadcasting messages, can also be misused by attackers. Vigilia addresses this issue by assigning a unique randomized group key to each device (the router then unicasts group packets) and combining this with proxy ARP [161]. Please note that while these options are present in the *hostapd* source code, they do not work properly and required us to fix them.

Application Sandboxing. Vigilia can run multiple applications on the same host. This brings the possibility that a malicious application can masquerade as another application on the same host by stealing the other application's port. Alternatively, a malicious application might try to access or modify files that are owned by another application. To prevent these attacks, Vigilia sandboxes applications using TOMOYO Linux [189]—components are restricted to their own ports and files.

Zigbee Support. An issue with SmartThings is that any driver that obtains the Zigbee address of any Zigbee device can send commands to it [86]. The problem is that device drivers explicitly build low-level Zigbee packets. These packets include the destination address for the commands and the address where responses should be sent. Thus, SmartThings trusts that device drivers are not malicious. Malicious device drivers can easily communicate with any Zigbee device whose address they have.

Vigilia guarantees that device drivers cannot interact with the wrong Zigbee devices. Vigilia's Zigbee support consists of four components: (1) language support for communicating Zigbee addresses to device drivers, (2) language support to ensure that honest device drivers do not manually produce Zigbee address objects, (3) a Zigbee abstraction that separates the specification of addresses from device commands, and (4) a Zigbee firewall that verifies that the given device driver has permission to communicate with the specific Zigbee device.

At the language level, Vigilia uses the same basic set-based abstraction that it uses for both RMI and IP addresses to check for permission bugs in Zigbee accesses. It then enforces these properties using runtime permission checks in the Zigbee gateway. The Zigbee gateway checks are configured automatically by the Vigilia master to implement the permissions granted by the end user. These checks use the source port and IP address to verify that a given Zigbee device driver has been granted permission to communicate with the specific Zigbee device address—this is a Zigbee firewall mechanism that is parallel to the firewall on the router.

Some Zigbee requests can leak information about other devices or configure a Zigbee device to interact with other devices. Thus the Zigbee gateway limits the types of messages a device driver can send to prevent the device driver from directly performing commands such as device discovery. The Zigbee gateway also filters incoming messages to ensure that device drivers only receive messages about the relevant device.

Incoming messages are often reports that are generated by a network node. For a node to receive information from another network node it must tell that node to send reports using a ZDO bind command. The Zigbee gateway remembers which driver performed a ZDO bind command, and to which node and cluster. When a report arrives from a Zigbee node, the gateway consults a table to determine which driver should receive it.

3.8 Evaluation

We deployed Vigilia on a test bed that consists of the following devices: 2 Raspberry Pi 2 compute nodes, a Google Nexus 5X smartphone, a Netgear Nighthawk R7800 wireless router, 2 LIFX Color 1000 bulbs, 4 Amcrest IP2M-841 ProHD 1080P cameras, a XBee S2C Zigbee module attached to a Raspberry Pi 1 (Zigbee gateway), a Spruce soil moisture Zigbee sensor, a Blossom sprinkler controller, 2 iHome iWS2 AirPlay speakers, a D-Link DCH-S220 siren, 3 Samsung SmartThings Zigbee sensors (motion, water-leak, and multi-purpose), and a Kwikset SmartCode 910 Zigbee lock. Table 3.1 presents the lines of code for our applications.

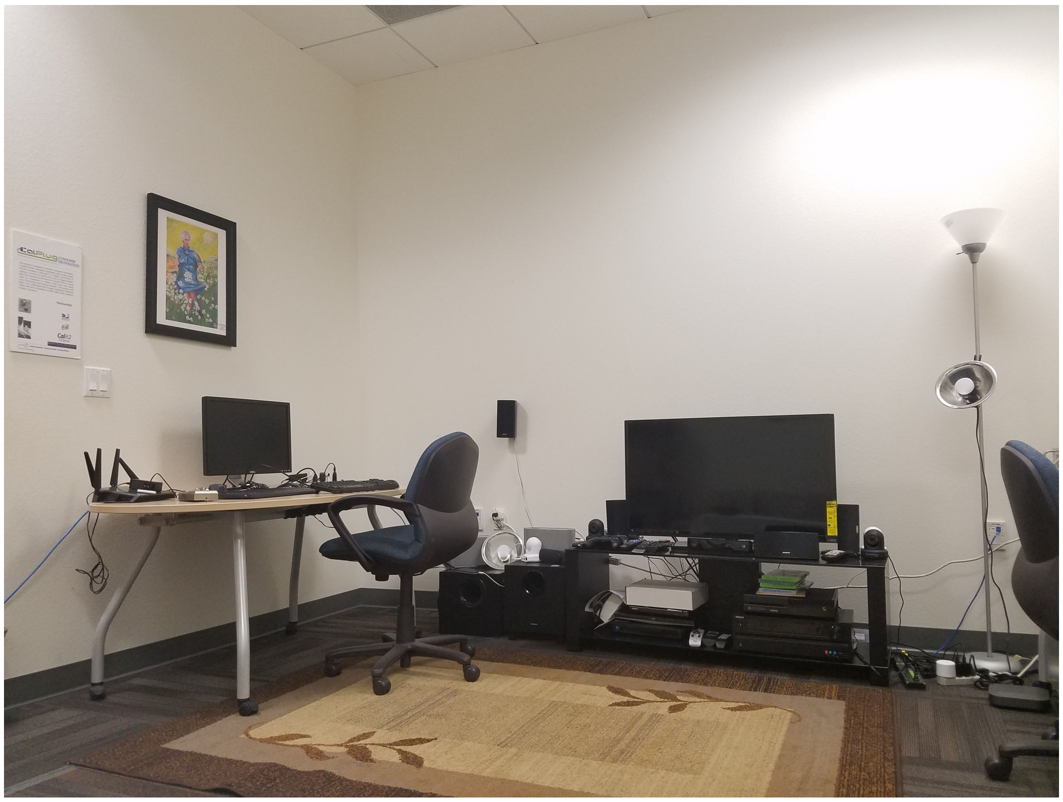


Figure 3.7: Vigilia hardware setup.

Our test bed is built in a smart home lab environment. Figure 3.7 shows the hardware setup in the lab.

3.8.1 Applications

We implemented four applications on our test bed. Table 3.2 presents the summary of these applications.

Irrigation. The irrigation application optimizes watering to conserve water. It uses the Spruce moisture sensor to measure soil moisture. The system makes use of weather forecasts to determine the expected precipitation. When people walk on a lawn, they stress the lawn and thus it requires more water [19, 99]. It uses cameras to monitor lawn usage and thus whether it requires extra water. The Spruce moisture sensor uses Zigbee to communicate; we have implemented a driver for this sensor that uses the sensor to monitor soil moisture.

Table 3.1: Lines of code in Vigilia applications.

Application	Application LOC	Driver LOC	Library LOC	Android LOC
Irrigation	4,075	2,975	401,843	208
Lights	1,683	3,456	401,843	N/A
Music	1,237	2,434	25,254	641
Home Security	2,299	4,177	401,843	187

Table 3.2: Summary of Vigilia applications.

Application	**Smart Home Devices**	**Security Properties**
Irrigation	1 Spruce soil moisture sensor 1 Blossom sprinkler controller 1 Amcrest camera 1 Google smartphone	This benchmark uses the device drivers for camera, Spruce moisture sensor, and sprinkler controller. It also includes a Zigbee gateway that relays messages to the Spruce sensor. Vigilia generates firewall rules that only allow the following communication: (1) the application can communicate with the drivers, phone, and the weather forecast website and (2) each device driver can communicate with its respective device. Each communication channel is isolated from the others and from all outside devices by (1) the compute node firewall and (2) the router firewall. The runtime system sends filtering rules also to the Zigbee gateway, ensuring that the Spruce driver can only communicate with the Spruce sensor.
Lights	2 LIFX light bulbs 2 Amcrest cameras	This benchmark uses the device drivers for camera and light bulb. Vigilia generates firewall rules that only allow the following communication: (1) the application can communicate with the device drivers and (2) each device driver can communicate with its respective device (*i.e.*, light bulb or camera). Each communication channel is isolated in a way similar to Irrigation.
Music	2 iHome speakers 1 Google smartphone	This benchmark uses a phone app and two speaker drivers. Vigilia generates firewall rules that only allow the following communication: (1) the main music application can communicate with the speaker drivers and the phone app, and (2) each of the device drivers can communicate with its respective speaker. Each communication channel is isolated in a similar manner.
Home Security	3 Samsung SmartThings sensors 1 Kwikset door lock 1 Amcrest camera 1 D-Link siren 1 Google smartphone	This benchmark uses the device drivers for camera, siren, door lock, and SmartThings sensors. Vigilia generates firewall rules that only allow the following communication: (1) the main home security application can communicate with it device drivers and the cloud, and (2) each device driver can communicate with its respective device. Each communication channel is isolated in a similar manner.

Table 3.3: Attacks performed on devices.

No.	Attack	Application	Detail
1.	Sprinkler attack	Irrigation	A rogue program that controls the sprinkler (*i.e.*, turn on valves, reconfigure wireless connectivity, and update the firmware based on a non-documented, non-secured RESTful API to port 80 [48]).
2.	Light bulb attack	Lights	A rogue program that issues commands to turn the light on and off (port 56700).
3.	Speaker attack	Music	A rogue program that sends and plays music file on the speaker (port 80).
4.	Camera attack	Home Security	A HTTP URL is used to view the main/sub stream via a web browser (port 80).
5.	Siren attack	Home Security	A rogue program that launches a brute-force attack to guess the PIN code of the siren; an attacker can use this PIN code to perform a valid authentication (port 80).
6.	Deauth. attacks	All	A jammer is used to deauthenticate a specific device (*i.e.*, sprinkler, light bulb, speaker, camera, or siren) from its original *access point* (AP) to let it join a malicious AP with the same SSID and PSK as the ones used for the actual AP. Thereafter, the device is attacked using the attack for the specific device (*i.e.*, attack 1, 2, 3, 4, or 5).

An Amcrest camera monitors the usage of the lawn to adjust the soil moisture target. An Android app provides the user interface. Finally, a Blossom sprinkler controller actuates the sprinklers.

Lights. The light application attempts to save energy by turning lights off in unoccupied spaces, and to improve sleep by adjusting brightness and color temperature to match the sun's color [105, 98, 62]. The application uses cameras combined with image processing to detect people. We use two Amcrest cameras to monitor rooms and control the two LiFX light bulbs.

Music. The music application tracks people using Wi-Fi-based indoor localization of their cell phone and plays music from the closest speakers. An Android phone is used to implement localization and play music through two iHome speakers.

Home Security. The home security application is modeled after commercial home security products. Such applications usually consist of multiple sensors that can detect intrusions/anomalies and sound an alarm. Our test bed uses an Amcrest camera, three Samsung SmartThings sensors (*i.e.*, motion, water-leak, and multi-purpose sensors), a Kwikset door lock, and a D-Link siren as the alarm. Sensor and door lock drivers communicate with the three sensors and the door lock through the Zigbee gateway. Finally, an Android app implements a UI through the secure cloud (see Section 3.5).

3.8.2 Comparisons

We next compare Vigilia with existing commercial (Norton Core [79] and Bitdefender BOX 2 [78]) and research systems (HanGuard [68] and IoTSec [184]).

Attacks. We designed a set of direct attacks, under our threat model (Section 3.3), against our smart home devices. The sprinkler, speaker, camera, and siren communicate through port 80 using the HTTP protocol. The speaker also uses other ports as it communicates using the AirPlay protocol [110]. The sprinkler particularly has a known vulnerability that can be exploited through a non-documented and non-secured RESTful API [48].

The light bulb communicates through port 56700, through which all LiFX bulbs listen [128]. The deauthentication attack is a more sophisticated attack that we use in combination with the first five attacks that directly target the devices. This attack deauthenticates a device, and makes it leave its router to join a malicious router that has the same SSID and PSK. When the device joins the other router, we can forcefully launch a direct attack to the device. Table 3.3 summarizes all of them.

For every system that we evaluated, we connected the smart home devices to the system and we performed the direct attacks. When a direct attack failed, we performed a combination attack. We first deauthenticated the device, let it join the malicious router that we have

Table 3.4: Vigilia comparison with other systems.

Attack	Normal	IoTSec	Vigilia
Sprinkler cont. attack	✓	✓	×
Light bulb attack	✓	✓	×
Speaker attack	✓	×	×
Camera attack	✓	✓	×
Siren attack	✓	×	×
Deauthentication + sprinkler cont. attack	N/A	N/A	×
Deauthentication + light bulb attack	N/A	N/A	×
Deauthentication + speaker attack	N/A	✓	×
Deauthentication + camera attack	N/A	N/A	×
Deauthentication + siren attack	N/A	✓	×
✓= successful attack	× = thwarted attack		

prepared, and performed the direct attack. Table 3.4 summarizes the results. We also performed the attacks on a normal router to establish a baseline. The normal router does not have any of the security properties that the Vigilia router has.

SmartThings. We implemented several previously known attacks against the SmartThings hub. In our first attack, we modified a device handler to subscribe to all LAN traffic. When we installed this device handler, there was no notification that it might access all SSDP network communications. We then ran the handler and could observe all SSDP packets in the network traffic that goes through the hub.

We next modified the service manager component of the WeMo Switch driver to change the IP address and port of a device after installation. This allowed us to control arbitrary devices on the LAN. Since third party drivers are commonly used to control smart home devices under SmartThings (*e.g.*, the only driver for Google Nest is a third party driver written by a hobbyist), this is a significant threat. This hack can be used to communicate with any device on the LAN.

We then implemented the same type of attack on Zigbee drivers and have discovered that Zigbee drivers can contact arbitrary Zigbee devices and send arbitrary Zigbee commands [86].

None of these attacks are possible under Vigilia. Vigilia blocks all network traffic by default and thus components can only access network traffic that they have been explicitly configured to access and that was explicitly intended for the component. Drivers under Vigilia are subject to the fine-grained access controls for both the TCP/IP and Zigbee networks and thus can only access the devices they were explicitly configured for. Moreover, our Zigbee framework prevents issuing commands that would cause a Zigbee device to interfere with other Zigbee devices.

Finally, as part of our general attacks reported later in this section, we sent commands directly to smart home devices. SmartThings does not block any such attacks. Vigilia blocks all such attacks.

Commercial Systems. We selected Norton Core and Bitdefender BOX 2, which are two leading secure routers that protect smart home IoT devices [79, 145, 78, 146]. They both use machine learning to learn the normal behavior of smart home devices. Their system compares device behavior against their database that contains information about vulnerabilities, attacks, viruses, malicious activities, etc., and warns users when it detects anomalies.

We first connected our devices to Norton Core and Bitdefender BOX 2. Subsequently, we performed a number of direct attacks against the smart home devices. All of these attacks were successful—the two routers were not able to protect the devices. Thus we categorize these systems under the normal router category in our results.

Further inspection revealed that these systems operate under a different threat model—they only defend against attacks that come from outside. A device inside the local network is considered safe and trusted—it is allowed to generate any traffic to any of the other local devices. Hence, they do not defend against our attacks that come from compromised local devices, *e.g.*, devices hacked and controlled by people with malicious intents.

Research Systems. For research systems, we evaluated HanGuard [68] and IoTSec [184]. To the best of our knowledge, these systems are the closest to Vigilia in terms of the threat model.

HanGuard uses SDN-like techniques to learn the normal traffic between smartphone apps and their respective smart home devices. A *Monitor* app runs on the phone to identify any attacks and inform the router through the system's *control plane*. The router then enforces policies in the *data plane* after verifying the party that attempts to access the device. Unfortunately, we could not obtain the implementation of HanGuard. Thus, we could not compare HanGuard with Vigilia. However, the paper [68] implies that HanGuard would leave IoT devices vulnerable to the combination attacks that can be thwarted by Vigilia.

IoTSec has two phases: *profiling* and *deployment*. During profiling, it attempts to learn the normal traffic of devices, *e.g.*, legitimate source and destination IP addresses, port numbers, protocols, etc. Then, a set of firewall rules will be generated and can be deployed on the router. Similarly to Vigilia, IoTSec reduces the attack surface with firewall while trying to maintain full functionality of devices.

To evaluate IoTSec, we connected our devices to a router running the IoTSec profiler. We then executed the four Vigilia applications, but turned off Vigilia's firewall protection. The IoTSec profiler learned the normal traffic of the four applications and generated a set of firewall rules for all devices. Finally, we deployed the firewall rules on the router and restarted the applications subsequently.

A key weakness of IoTSec is that it relies entirely on profiling. For most of our devices, this approach worked because they always use the same IP address, port numbers, and protocols. However, the iHome speaker randomly selects a port number and the generated firewall rules disrupted the speaker's operation—these rules assume devices always use the same port numbers. In addition, profiling may not exhibit all behaviors of a system. For example, during profiling, we did not trigger the siren to let it go off—deliberately triggering the home alarm to enable the home security system is not a normal behavior. The profiler did not learn the siren's traffic and thus the generated firewall rules disabled the siren.

We performed direct attacks on the devices. The attacks against the sprinkler, light bulb, and camera were successful because the generated firewall rules allowed them to communicate through their respective port numbers. During profiling, IoTSec does not learn the source IP addresses—it assumes that devices are allowed to communicate through their respective ports regardless of the source IP addresses. Hence, the firewall rules are not fine-grained enough to block communications from illegal sources.

The attacks against the speaker and siren failed because the incomplete firewall rules meant that they did not function at all. We then performed the deauthentication attack to both devices. After they joined our malicious router, we successfully attacked them.

Vigilia. We performed the same attacks against the devices under Vigilia. We connected every device using a unique PSK to Vigilia's router. We ran the four applications simultaneously and attacked them.

Under the protection of Vigilia's firewall and sandboxing mechanisms, all of the applications and devices were fully functional, and all of the attacks were successfully thwarted. The direct device attacks were blocked by the deployed firewall rules on the router and the compute nodes. The deauthentication attack also failed as none of the devices could join the malicious router. Even though the malicious router was configured with the same SSID and PSK as the Vigilia router, the devices did not use the router's default PSK—every device was connected to the Vigilia router using a unique PSK.

Table 3.5: Statistics of access attempts for the public IP experiment; 'A' is a placeholder for `128.200.150` and 'B' is for `calplug.uci.edu`; column **DS** reports the number of distinct sources; **TCP** and **UDP** reports numbers in the form of X/Y where X and Y represent the numbers of total and distinct addresses, respectively.

IP	Domain	Total	DS	TCP	UDP	ICMP
A.130	iot1.B	2,944	1,411	1,992 / 340	334 / 60	218
A.131	iot2.B	2,791	1,451	2,039 / 343	256 / 84	69
A.132	iot3.B	3,255	1,405	1,947 / 350	203 / 62	693
A.133	iot4.B	2,841	1,364	1,934 / 344	219 / 73	271
A.134	iot5.B	2,769	1,422	2,043 / 349	233 / 62	82
A.135	iot6.B	2,792	1,416	2,024 / 353	281 / 65	69
A.136	iot7.B	3,284	1,443	2,106 / 342	276 / 64	496
A.137	iot8.B	3,006	1,507	2,084 / 316	272 / 88	246
A.138	iot9.B	3,000	1,433	2,028 / 316	353 / 72	231
A.139	iot10.B	2,620	1,370	1,862 / 283	244 / 62	169
A.140	iot11.B	2,692	1,419	1,983 / 316	258 / 69	66
A.141	iot12.B	2,709	1,429	2,018 / 267	262 / 69	93
A.142	iot13.B	3,582	1,397	2,042 / 352	287 / 63	838
A.143	iot14.B	2	2	0 / 0	2 / 2	0
A.144	iot15.B	3	2	0 / 0	3 / 2	0
A.145	iot16.B	6	2	0 / 0	6 / 1	0
	Total	**38,296**				

3.8.3 Public IP

To further evaluate Vigilia, we conducted another experiment, in which we assigned public IP addresses to our devices. While other secure routers generally claim to protect smart home IoT devices when they are connected to a local network behind *Network Address Translation* (NAT), we let our devices be *exposed to the open Internet.* For this experiment, we assigned a public IP address for every device, ran the four applications, and let Vigilia set up firewall rules on the router. We ran this experiment for approximately 10 days.

Table 3.5 summarizes the results of the experiment. The table reports, for a device, the IP address, its domain name, the total number of access attempts for this device, the number of distinct sources these attempts came from, the number of total and distinct TCP attempts, the number of total and distinct UDP attempts, as well as the number of ICMP packets. The 16 public IP addresses generated 38,296 access attempts—approximately 3,629 access attempts per day and 240 access attempts per day per device.

Table 3.6: Statistics of public IP experiment on cameras; 'A' is for `128.200.150`; **Att, Src, Pkt** represent the number of access attempts, sources, and network packets, respectively; **U/T** stands for UDP/TCP.

IP	With Vigilia (Att / Src / Pkt)	Ports (U/T)	With pwd only (Att / Src / Pkt)	Ports (U/T)
A.134	106 / 96 / 114	6 / 23	5,337 / 117 / 9,658	39 / 48
A.135	111 / 100 / 115	7 / 23	20,172 / 124 / 40,998	47 / 46
A.136	206 / 97 / 208	6 / 22	1,201 / 98 / 2,039	19 / 43
A.137	128 / 109 / 135	7 / 21	4,520 / 119 / 8,889	17 / 51

All the attempts were thwarted by the Vigilia firewall rules set up on the router. No device responded to any of the sources, except for the ICMP packets. The network trace suggests that most of the access attempts were either *ICMP ping* or *TCP SYN/ACK port scanning* [208], which are the two approaches attackers commonly use to "test the waters". Since our devices only replied to ICMP pings, there were no further packets from more sophisticated attacks.

Real Attacks on Cameras. We conducted an additional experiment with our Amcrest cameras and exposed them to real attacks. This experiment was done under three scenarios: 1) cameras were protected under Vigilia, 2) cameras were protected with passwords, and 3) cameras were unprotected. Each scenario lasted for 14 hours.

Table 3.6 summarizes the results of the experiment for the first two scenarios. In the first scenario, the first camera with address `128.200.150.134` received 106 access attempts from 96 distinct sources with 114 packets of total traffic under Vigilia's protection—the attempts targeted 6 distinct UDP ports and 23 distinct TCP ports, and were all thwarted. In the second scenario, the same camera received many more access attempts. Although the camera

Table 3.7: Vigilia microbenchmark results.

	Node-to-Node	Overhead	Node-to-LAN	Overhead
Normal	2.91 MB/s	N/A	5.64 MB/s	N/A
Hairpin	2.78 MB/s	4.5%	5.62 MB/s	0.3%
Hairpin + Policies	2.75 MB/s	5.5%	5.62 MB/s	0.3%

had not been compromised, it could have been had we extended the duration. This is especially the case when people use generic/default passwords for their cameras, as shown in a study on the Mirai botnet attack [36]—*there was even a ... Mirai infection on Amcrest cameras despite strong passwords* [92].

In the third scenario, it took *just 15 minutes*, for *all of the four cameras* to be hacked and crippled—the user interface was completely broken although it was still able to stream out video. Each attack session for each camera just took around 172 - 362 packet exchanges between each camera and the attacker. The network trace in the log file suggests that the attackers used a technique called *XML-RPC attack* [176], which typically brings down web services by executing *remote procedure call* (RPC) commands via the HTTP protocol.

3.8.4 Performance Microbenchmarks

Vigilia's primary enforcement is implemented by firewall rules. The other components are not on the hot paths and should add minimal overhead. This subsection evaluated the overhead of Vigilia's routing policies on network throughput. We measured the network bandwidth under three different router configurations: normal mode, hairpin mode, and hairpin mode with policies. We performed each of these measurements under two different setups: (1) a node-to-node bandwidth measurement using the Apache HTTP server on a Raspberry Pi 2 and (2) a node-to-LAN bandwidth measurement using the Apache HTTP server on an Intel Core i7-3770 CPU 3.40GHz machine running Ubuntu. We ran `wget` on another Raspberry Pi 2 to retrieve a 30 MB file from both the Raspberry Pi 2 and the Ubuntu machine. All equipment was placed in a Faraday cage to limit interference. We report average bandwidth over 20 runs.

Table 3.7 reports the average bandwidths. Under the node-to-node scenario, hairpin mode introduces a 4.5% overhead since it forces traffic to exit the driver and go through the kernel firewall. Under the node-to-LAN scenario, the lower overhead is not surprising as node-to-LAN traffic already exits the driver before going through the firewall. The firewall policies introduce almost negligible overhead for both setups. Node-to-node results show lower bandwidths as communication must take two hops on the same Wi-Fi channel. Overall, the overheads are relatively small.

Chapter 4

Understanding and Detecting Conflicting Interactions between Smart Home IoT Applications

In this chapter, we present our wide scale study on smart home apps to understand the nature of the interactions between them. We have identified the following five research questions to guide our study.

RQ1: What kinds of *interactions* are there? We have collected and studied 198 official SmartThings apps and 69 third-party apps. Compared with recent studies of smart home apps [222, 188, 61, 60], we have among the largest app suite. To understand interactions and possible conflicts, we analyzed these apps in pairs and examined all pairs of apps that can potentially interact. We discovered three main categories of interactions: (1) interactions between apps that access the same device, (2) interactions between apps such that the output from one app interferes with the input of the other app (*e.g.*, via sensors), and (3) interactions between apps accessing global variables, *e.g.*, whether the home is in the *Home* or *Away* mode.

RQ2: What types of *conflicts* arise between smart home apps? For an app pair, we first inspected their source code and documentation to understand the intended behavior of each individual app and then reason about possible interactions between them. If there exists an interaction that can compromise the desired functionality of either app, we say that this pair has a *conflict*, *e.g.*, the functionality of the `FireCO2Alarm` app to door-lock is compromised by the `Lock-It-When-I-Leave` app. Our goal is to carefully inspect apps that interact, and understand whether they conflict and if they do, why.

RQ3: How prevalent are these conflicts? We summarized the results of our study to understand how prevalent the conflicts are. We found that almost 60% of pairs in the first category, more than 90% of pairs in the second category, and around 11% of pairs in the third category have conflicts.

RQ4: Are there common coding patterns that are unsafe in the presence of app interactions? During our study, we observed several common programming idioms that often result in problematic interactions between apps. Discovering and classifying these idioms can help developers mitigate potential conflicts by avoiding these idioms.

RQ5: How can we automatically detect conflicts? Based on our findings, we develop a tool called IoTCheck that can automatically detect conflicts—Section 4.5 presents the design and implementation of the IoTCheck tool. IoTCheck model-checks smart home apps and automatically detects conflicts between apps. Our tool is available under an open source license [192, 190, 191].

4.1 Methodology

This section describes our research methodology. We first define several terms. Next, we discuss our database of smart home apps and the way we structure them for the study. Our study focuses on pair-wise interactions. The rationale is that pair-wise interactions

$$
\begin{aligned}
\mathcal{X} \in \text{Execution} &= (\text{Action} \mid \text{Event} \mid \text{Update})^* \\[1ex]
\mathcal{A} \in \text{Action} &= \text{read}(\alpha,\, d,\, \tau,\, r) \;\mid\; \text{write}(\alpha,\, d,\, \tau,\, r,\, v) \;\mid \\
&\quad\ \text{moderead}(\alpha) \;\mid\; \text{modewrite}(\alpha,\, \mu) \;\mid \\
&\quad\ \text{schedule}(\alpha,\, t,\, m) \\[1ex]
\textit{event} \in \text{Event} &= \text{devEv}(\alpha,\, d,\, \tau,\, r,\, v) \;\mid\; \text{modeEv}(\alpha,\, \mu) \;\mid \\
&\quad\ \text{schedEv}(\alpha,\, m) \\[1ex]
\mathcal{U} \in \text{Update} &= \text{devUp}(\alpha,\, d,\, \tau,\, r,\, v) \;\mid\; \text{modeUp}(\alpha,\, \mu)
\end{aligned}
$$

$$
\alpha \in \text{App} \quad d \in \text{DeviceID} \quad \tau \in \text{DeviceType} \quad r \in \text{Feature}
$$
$$
v \in \text{Value} \quad t \in \text{Time} \quad \mu \in \text{Mode} \quad m \in \text{Method}
$$

Figure 4.1: SmartThings Execution Traces.

are fundamental for understanding multi-app interactions since multi-app interactions can be decomposed to pair-wise interactions for reasoning about. Although we have carefully observed how these apps interact in bigger groups, we have not seen any new interaction patterns that manifest only when three or more apps are involved.

4.1.1 Definitions

Execution Traces. We first formalize our notion of execution traces for SmartThings in Figure 4.1. The traces can be generated by one or more apps that run concurrently. An execution $\mathcal{X} \in$ Execution from a set of apps is a sequence of the following:

(1) Action: App α performs an action $\mathcal{A} \in$ Action by executing any of the following set of operations:

- read(α, d, τ, r) and write(α, d, τ, r, v), which read from and write a value v to a feature r of a device with ID d and device type τ, respectively;
- moderead(α) and modewrite(α, μ), which read from and write a new mode μ to the `location .mode` variable, respectively; and

- schedule(α, t, m), which schedules a method m to run at time t.

(2) Event: An event $event \in$ Event is either:

- devEv(α, d, τ, r, v), a device event is delivered to app α from device d to notify the app of device status update;
- modeEv(α, μ), a mode event is delivered to app α to notify it of a mode change; or
- schedEv(α, m), a schedule event denotes when the framework processes a schedule action and executes the method m in app α.

(3) Update: An update $\mathcal{U} \in$ Update is an external input to the smart home. It is either:

- devUp(α, d, τ, r, v), an update with a new value v generated from a device with ID d and type τ for feature r and value v, *i.e.*, a sensor reading a temperature change; or
- modeUp(α, μ); an update with a new mode $\mu_,$ *e.g.*, the homeowner manually setting a new mode.

Interacts-with Relation. We next define a relation *interacts-with* over the domain of Apps $\times$ Apps where Apps is the set of all smart home apps. A pair of apps (α_1, α_2) $\in$ *interacts-with* (*i.e.*, α_1 *interacts-with* α_2) if they interact with each other in one of the three ways:

(1) Access the same device capability: Apps α_1 and α_2 can access a shared device using the same capability; α_1 *updates* the device state (*i.e.*, feature r and value v) and α_2 accesses (*i.e.*, updates or reads) the device state. We refer to this relationship as a *device* interaction. For example, α_1 may turn on a switch based on the input of a light/illuminance sensor and α_2 may turn off the same switch based on a motion sensor, both calling methods on the same device handler object.

(2) Physical interaction: We say that two apps have a *physical-medium* interaction if the output of α_1 *physically* becomes an input for α_2 and affects the execution of α_2. For example, α_1 activates a robot vacuum cleaner at a certain time during the day, and the robot's movement becomes the input to a motion sensor that is used by α_2.

(3) Access the same global variable: Apps α_1 and α_2 can interact via the same global variable, whose value is stored on the cloud, *e.g.*, α_1 updates the variable and α_2 accesses it. This is referred to as a *global-variable* interaction. In this study, we focused on the `location.mode` variable because it is the only global variable in the SmartThings platform that allows for both *write* and *read* accesses. `location.mode` has three preconfigured values: *Home*, *Away*, and *Night*. An example scenario is that one app updates `location.mode` based on the input of the presence sensor while the second app reads it to determine whether a door should be locked/unlocked.

Conflict Relation. Apps α_1 and α_2 *conflict* if they interact (in one of the ways discussed above) and the interaction may compromise the *correctness* of the apps or produce an *unintended outcome*. Although the notion of a conflict is somewhat vague, we found that Definitions 4.1 and 4.2 worked well most of the time in practice.

Definition 4.1. Device/Global-Variable Conflict. *Two apps α_1 and α_2 conflict iff there exists an execution $\mathcal{X}$ of α_1 and α_2 and two actions $\mathcal{A}_1$ and $\mathcal{A}_2$ that update the same feature r or mode μ in $\mathcal{X}$ such that: (1) $\mathcal{A}_1$ and $\mathcal{A}_2$ are performed by different apps (α_1 and α_2), (2) $\mathcal{A}_1$ and $\mathcal{A}_2$ write different values (v_1 and v_2, or μ_1 and μ_2), (3) there is no such $\mathcal{A}_3$ that updates the same r or μ and that the update is ordered between $\mathcal{A}_1$ and $\mathcal{A}_2$, and (4) $\mathcal{A}_2$ was not initiated by a direct user action.*

Table 4.1: Groups of apps for device-type pairing.

Group	**Capability**	**Subgroup**	**App**	
			# Apps	**# Pairs**
Switches	`switch`	General	24	276
		Lights	32	496
		AC/fan/heat	3	3
		Vent	3	3
		Camera	2	1
Locks	`lock`		21	210
Thermostats	`thermostat`		19	171
Lights	`colorControl`	Hue	13	78
		Non-Hue	11	55
Dimmers	`switchLevel`		11	55
Alarms	`alarm`		10	45
Valves	`valve`		7	21
Music Players	`musicPlayer`		5	10
Relay	`relaySwitch`		5	10
Speech Synthesizers	`speechSynthesis`		3	3
Cameras	`imageCapture`		2	1
		Total	**171**	**1,438**

Definition 4.2. Physical-Medium Conflict. *Two apps α_1 and α_2 conflict iff one app performs an action that affects a physical medium (e.g., motion) and the other app reads from a sensor that can sense that physical medium (e.g., a motion sensor).*

4.1.2 Smart Home App Pairs

Choice of Apps. We studied 198 official and 69 third-party smart home apps that we have collected from the SmartThings official Github [181] and other third-party repositories. While the statistics of app usages and installations are proprietary, all the apps that we used in this study can be obtained easily from the aforementioned repositories. Today, the SmartThings official Github [181] has an active user community—it has been forked into personal repositories more than 70,000 times. Any user can get and upload any app's source code to the SmartThings Marketplace via the SmartThings Groovy IDE [26]. Thus, users can install and use any app.

App Pairing. These apps were initially developed to perform their specific functionality. There are no standardized guidelines either from SmartThings or from the community as to how to develop an app in a way so that it can safely interact with other apps.

Our process for manual examination was to independently examine each app pair by at least two of the authors. In the event that the two examiners disagreed about whether an app pair conflicted, they discussed their disagreement on the app-pair's classification and reached a consensus. There are 35,511 app pairs given the 267 apps we collected above. From this huge set of pairs, we identify 2,844 pairs of apps that potentially interact with each other. We next explain how we use the three interact-with conditions to identify these 2,844 pairs. We will then study how many of these 2,844 pairs contain conflicts in Sections 4.2–4.4.

Device-Type Pairing. To identify apps that have device interactions, we first divide the 267 apps into groups based on what type of device an app aims to manage, as shown in Table 4.1. Clearly, if two apps do not access a common device, it is impossible for them to have *device* interaction.

Out of the 267 apps, we excluded 132 apps for three reasons. First, we excluded apps that take inputs from outside of the SmartThings platform. For instance, the IFTTT (If-This-Then-That) [23] app functions as a bridge between the SmartThings platform and IFTTT, a *third-party* platform. These apps typically wait for a third-party application built on a third-party platform (*e.g.*, IFTTT and other similar platforms) to send commands and generate events through HTTP endpoints. We do not have access to the source code of such third-party applications; thus, it is not possible to accurately reason about potential interactions. Second, we excluded apps that only send messages to a smartphone about the state of sensors because these apps do not interact with other apps. Third, we also excluded apps that use third-party specific device handlers since these apps cannot share a device with other apps. Therefore, we included 135 apps for *device* interaction. Some of them access multiple devices and, thus, are included in multiple groups of devices—hence, a total of 171

Table 4.2: Groups of apps for physical-medium pairing.

Output	**# Apps**	**Sensor**	**# Apps**	**# Pairs**
Lights	42	Illum.	5	205
Moving Dev.	2	Motion	39	78
Water Valves	2	Water	11	21
Sound Dev.	21	Sound	1	21
			Total	**325**

apps. At the end, we identified a total of 1,438 pairs from the 171 apps classified in various device-type-based groups.

For some groups, we identify all pairs of apps from the group as *device*-interaction pairs. For example, the *Locks* group contains 21 apps, we inspected all the $\binom{21}{2} = 210$ pairs and confirmed them all to be *device*-interaction pairs.

For some groups that provide *generic* functionality, such as Switches and Lights, we further create sub-groups and only identify apps that belong to the same sub-group as having a *device* interaction. For example, for the Switches group, out of a total of 64 apps, 24 access general switches (276 pairs), 32 access light switches (496 pairs), 3 access AC/fan/heater (3 pairs), 3 access the ventilation system (3 pairs), and 2 access cameras (1 pair). We also found 8 apps (not included in Table 4.1) that control specific devices (*e.g.*, curling-iron) that are not shared by other apps; hence, no pairs were constructed for these apps. The Lights group consists of apps that use the light device handler (*i.e.*, `capability.colorControl`) to turn the lights on or off, set their illuminance level [20], or change their colors. Each group was divided into a subgroup of apps that controls Philips Hue lights and another subgroup that controls non-Hue lights. In the Lights group, there are 13 apps for Hue leading to 78 pairs and 11 apps for non-Hue leading to 55 pairs.

Physical-Medium Pairing. Two apps can interact via a *physical medium*; *e.g.*, one app generates an output that could be a *physical* input to the other app. To illustrate, consider an app that changes the state (*i.e.*, toggle on/off) of light bulbs. These changes also affect the illuminance produced by the light bulbs, which can become an input to apps that read from illuminance sensors.

Table 4.2 reports results for apps that interact physically. We grouped them based on the output-input relationships, such as lights (output) and illuminance sensors (input), moving devices (output) and motion sensors (input), water valves (output) and water sensors (input), or sound-generating devices (output) and sound sensors (input). For the light-illuminance-sensor relationship, for example, we constructed a total of 205 pairs for the 42 apps that control lights and the 5 apps that read from illuminance sensors.

Global-Variable Pairing. Apps can also interact if they access the same global variable. Currently, there is only one global variable in the SmartThings platform that multiple apps can read from and write into: `location.mode`. We grouped together all the 47 apps that access it for a total of 1,081 pairs.

4.1.3 Threats to Validity

External Validity. This study focused on Samsung's SmartThings platform and thus may miss interaction patterns specific to other platforms. However, we believe that most of the findings and insights revealed in this study are universal for smart home applications and frameworks. For instance, our results also apply to rule-based systems, *e.g.*, IFTTT—two rules: (1) "if the humidity is high, turn off the AC" and (2) "if the temperature is low, turn on the AC", have a conflict by our definition if the humidity is high and the temperature is low. Even for interactions that are specific to the SmartThings platform (*e.g.*, concurrent accesses to the `location.mode` variable), the patterns discovered under such interactions are general. For example, other platforms would also have global variables that serve similar purposes and hence our results can be generalized to these other platforms as well.

Internal Validity. This study covers *all of the 198 official apps* that we could find in the SmartThings official Github repository and the example set for SmartThings tutorials as of July 2018. We added 69 third-party apps that we gathered from various other sources.

While we studied the *complete set* of the official apps, the third-party apps used in the study may not be exhaustive. Nevertheless, our experience shows that the patterns that exist in the official apps are similar to those in the third-party apps. We believe adding new third-party apps would not change the main findings and insights.

In this study, we limited the scope of app interactions to pairs, and hence, there could be new types of interactions that manifest only when three or more apps are involved. However, we have already manually inspected a large number of triplets and not found any new interaction patterns that do not exhibit in pairs.

We manually inspected app pairs to determine whether the two apps in each pair can conflict. The manual determination is subjective in some cases—it reflects the authors' beliefs of whether the interactions between a pair of apps represent an unintended outcome. Whether this conflict represents a problem in the real world is a very complicated question and can depend on (1) the intended use of the homeowner and (2) the home environment. For example, if one app turns a light on and a second app based on the absence of motion from a sensor turns the light off, we classify this as a conflict. However, users may compose apps with the intention of this app interaction. As another example, certain interactions are made over physical mediums; for instance, the sound generated by a speaker app could become the input of a sound sensor used by a different app. In this case, whether the sensor can pick up the sound depends on whether it is physically close to the speaker generating the sound. In the study, we assume that this interaction can actually happen although the speaker and the sensor may be far away in a real-life deployment.

Conflicts that have safety or security aspects are certainly critical and could be harmful. However, it is somewhat difficult to determine the potential safety hazards or implications of a conflict as they can depend on the specific deployment. For example, if a conflict causes a smart outlet to remain on, whether it is a safety hazard depends on what is plugged into the smart outlet, *e.g.*, toaster versus LED light. Nevertheless, even benign conflicts can render apps useless—they make a smart home system unpredictable and difficult to rely on with any confidence, ultimately causing users to get rid of the system.

Our ultimate goal is to identify all *avoidable* conflicts and their possible sources so that actions can be taken in future development and/or deployment to mitigate potential conflicts. Some conflicts can be potentially handled by the development of API with support for common app interaction patterns. On the contrary, if physical proximity is a concern, we could develop an analysis that warns the user during installation. This explains why we treated these two scenarios differently.

4.2 Device Interaction

This section presents our findings for apps that form pairs with *device* interactions. When we first studied this category, we found that some apps monitor status changes but do not initiate any changes on devices. When such an app is paired with another device monitor app, both apps *concurrently read* the device status and neither of them makes any changes to the device status. We refer to such a pair of apps as having a *read-read* relationship. 128 (8.9%) pairs have this relationship and thus do not interact. We classified *device* interactions into *non-conflicting* and *conflicting* interactions; the statistics of the classification are reported in Table 4.3.

Table 4.3: Statistics for *device* interaction.

Relationship	**# Pairs**	**Percentage**
Read-read	128	8.9%
Non-conflicting Interactions		
Direct-direct	20	1.4%
Composable	319	22.2%
Different-feature	52	3.6%
Same-feature	90	6.3%
	481	**33.5%**
Conflicting Interactions		
Feature conflicts	632	43.9%
Invalid-local-state	76	5.3%
Dropped-update	121	8.4%
	829	**57.6%**
Total	**1,438**	

4.2.1 RQ1: Types of Non-Conflicting Interactions

We observed three types of non-conflicting interactions. First, although two apps can access the same device, their accesses can only be triggered *manually* by users. Consequently, whether they conflict with each other depends on how users operate them. For example, `Big-Turn-ON` is such an app: it turns on switches when the user touches the app's user interface [8]. Two users may concurrently initiate conflicting commands to a switch through two apps like `Big-Turn-ON`. We consider this type of conflicts out of the control of apps. We refer to this type of interaction as a *direct-direct* relationship. We found that this relationship holds for 20 (1.4%) app pairs in the *device* category (see Table 4.3).

Second, certain apps can work together to realize desired functionality, and hence are *intended* to interact with each other. We refer to this type of interaction as a *composable* relationship and corresponding apps as *composable* apps. We found that this composable relationship holds for 319 (22.2%) pairs in the *device* category.

Please note that many of these composable apps were developed independently. For example, the `FireCO2Alarm` app sets off the alarm and triggers door-unlocks when smoke/fire is detected [162], while the `Initial-State-Event-Streamer` app [18] monitors and forwards events from many devices including the alarm device handler to a website [24] that allows users to remotely monitor device activities. These two apps were independently developed, but they could interact to fulfill a desired functionality at run time—notifying a user through the specific website that an alarm is set off.

Third, some apps simultaneously access different features of the same device or the same feature of the same device in a consistent way, and hence do not conflict with each other.

An example we discovered for the former (*i.e.*, accesses to different features) is the `Keep-Me-Cozy` and `Thermostats` [12, 22] pair of apps from the Thermostats group. One app calls methods on the thermostat to set heating or cooling points (*e.g.*, `setHeatingSetpoint()` and `setCoolingSetpoint()`), while the other app sets the mode of the thermostat (*e.g.*, via `setThermostatMode()`). Although these two apps control the same shared device, they operate on different features of the device. Hence, although the first app *interacts-with* the second app, there is *no conflict* between them. We refer to this interaction as a *different-feature* relationship and found this relationship holds for 52 (3.6%) pairs in the *device* category.

An example we discovered for the latter (*i.e.*, consistent accesses to the same feature) is the following pair of apps from the Locks group: the `Lock-It-at-a-Specific-Time` and `Auto-Lock-Door` apps [5, 171]. Both apps call `lock.lock()` to door-lock. We consider this interaction non-conflicting, since these apps' actions would lead the shared device to the *same state* and hence the expected outcome is not compromised. We refer to this interaction as a *same-feature* relationship and found it to hold for 90 (6.3%) pairs in the *device* category.

4.2.2 RQ2: Types of Conflicting Interactions

Of the 1,438 app pairs in the *device* category, 829 pairs exhibit conflicting behaviors. We classified these conflicting behaviors as either *feature conflicts* and *saved-state conflicts*.

Feature Conflicts. There are many pairs where the two apps attempt to update the same device state with *incompatible values*. An example is the `FireCO2Alarm` and `Lock-It-When-I-Leave` pair discussed in Chapter 1. Recall that the `FireCO2Alarm` app attempts to door-unlock during a fire while the `Lock-It-When-I-Leave` app could potentially door-lock. We refer to these conflicts as *feature conflicts*. A majority of the app pairs: 632 (43.9%) pairs in the *device* category have feature conflicts.

Saved-State Conflicts. Many apps use their local variables to keep track of device states and guide their own device updates. These apps easily become broken when paired with other apps that can update the same devices—a concurrent update from the other app would make this app's variable inconsistent with the device state.

Consider `Auto-Humidity-Vent` that turns on/off a fan based on the humidity level [2]. This app conflicts with the `Big-Turn-OFF` app that allows a user to manually turn off the fan [7] for the following reason. When `Auto-Humidity-Vent` detects that the room humidity is above a threshold, it turns on the fan and simultaneously updates its local state variable `state.fansOn` to `true`. A user may then use the `Big-Turn-OFF` app to turn off the fan, causing the room humidity to increase above the threshold. Unfortunately, since the local variable `state.fansOn` remains *true* in the `Auto-Humidity-Vent` app, unaware of the fan being turned off by `Big-Turn-OFF`, `Auto-Humidity-Vent` would stop functioning, incorrectly assuming that the fan is already on. We refer to this scenario as *invalid-local-state* conflicts, and found that 76 (5.3%) pairs in the *device* category exhibit this pattern.

A common pattern we observed is an app that stores and restores the state of a device. For example, the `Thermostat-Auto-Off` app restores the state of the thermostat to a previously stored state. Consider an execution in which after the `Thermostat-Auto-Off` app saves the current state (*e.g.*, `off`) of the thermostat into a local variable, a second app changes the actual device state to a different value (*e.g.*, `"cool"`), which does not propagate to `Thermostat-Auto-Off`'s internal state. The next time `Thermostat-Auto-Off` tries to restore the thermostat state, the restoration will be based on the stale and wrong value saved in the local variable. Thus the update performed by the second app is dropped. We refer to this scenario as *dropped-update* conflicts, and found 121 (8.4%) pairs in the *device* category exhibit this pattern.

4.2.3 RQ3: Prevalence of Conflicts

As reported in Table 4.3, 91.1% of the pairs in *device* category have actual interactions (*i.e.*, at least one device updates the device state), while 8.9% of the pairs have *read-read* relationships and hence do not actually interact. Of the pairs that have actual interactions, the majority (57.6%) have conflicts.

4.2.4 RQ4: Unsafe Coding Patterns

We found there are at least two unsafe coding patterns for *device* interactions: (1) *blind-update* and (2) *saved-state*. The *blind-update* pattern occurs in apps that blindly update the same state of the same device without checking the current state of the device. The *saved-state* pattern occurs when an app that saves the state of a device feature into a local variable and later uses the saved value. This may cause updates from other apps to be discarded. In some cases, a check of the current state before doing the update could help the app verify that its local state is consistent with the device state. However, with the existing APIs, there is no way to do the check-and-update in an *atomic* way—an app could only retrieve the device state by invoking a method m_1 and then update the state by invoking another method m_2; the state could be changed by another app after m_1 returns but before m_2 is completed.

Table 4.4: Statistics for *physical-medium* interaction.

Medium	**# Pairs**	**Percentage**
Non-Conflicting Interactions		
Water	10	3.1%
Sound	21	6.4%
	31	**9.5%**
Conflicting Interactions		
Water	11	3.4%
Motion	78	24.0%
Light state	151	46.5%
Light color	20	6.2%
Light brightness	5	1.5%
Light combination	29	8.9%
	294	**90.5%**
Total	**325**	

4.3 Physical-Medium Interaction

This section presents our findings for apps that interact via the physical world. In this category, two apps are paired when the output from the first app can physically become the input of the second app and affect its operation. Table 4.4 reports our findings.

4.3.1 RQ1&2: Types of (Non-)Conflicting Interactions

Motion. The first set of physical interactions are due to motion. An example pair is `Neato-(Connect)` and `Forgiving-Security` [25, 1]. `Neato-(Connect)` is a third-party app that controls a Neato vacuum-cleaning robot. When the app activates the robot, the robot starts cleaning the house. While it is moving around the house, its movement could trigger a motion sensor used by the `Forgiving-Security` app and thus set off a security alarm—a *false alarm*. Of the 325 app pairs in the *physical-medium* category, 78 pairs (24.0%) interact via motion and all exhibit conflicts.

Light. A similar set of app pairs are based on interactions via light. The `Turn-On-at-Sunset` and `Light-Up-the-Night` apps [17, 13] are an example. Consider a deployment in which each app controls a different light bulb. At sunset, the `Turn-On-at-Sunset` app may turn on a light bulb whose light may affect the illuminance sensor of the `Light-Up-the-Night` app. The `Light-Up-the-Night` app is supposed to turn on a light bulb when its illuminance sensor detects that the surrounding is dark. If the light bulb controlled by the `Turn-On-at-Sunset` app is sufficiently close to the illuminance sensor used by the `Light-Up-the-Night` app, the sensor may pick up some light from the light bulb. This could cause the `Light-Up-the-Night` app to determine that there is no need to turn on the light bulb, and hence, the two apps conflict.

Some apps can control a light bulb by changing its on/off state, colors, or brightness levels. Any of these changes can potentially be detected by an illuminance sensor [20].

Table 4.4 summarizes our findings: 151 pairs (46.5%) have a conflict through the change of light's on/off state; 20 pairs (6.2%) conflict through the change of light's color; 5 pairs (1.5%) conflict through the change of light's brightness; and 29 pairs (8.9%) conflict through a combination of the three.

Water. Physical interactions can also occur via water. An example pair that interacts via water consists of the `Sprayer-Controller-2` and `Close-The-Valve` apps [6, 3]. The former schedules irrigation for a certain amount of time periodically, while the latter closes a water valve when the water sensor detects moisture. When the water coming from a water sprayer controlled by the `Sprayer-Controller-2` app reaches the water sensor used by the `Close-The-Valve` app, the two apps interact. This interaction potentially results in a conflict because a bad moisture sensor placement could cause the `Close-The-Valve` app to prevent the irrigation that has been scheduled by the `Sprayer-Controller-2` app.

Our results show that 21 pairs interact through water: 11 of them have conflict and 10 do not. In each of these 10 pairs, the app that controls the water valve actually closes it when it detects moisture through its sensor. Therefore, no water can be produced and detected by the water sensor of the other app.

Sound. Apps can also interact via sound. For example, an interesting app pair we discovered is `Bose-SoundTouch-Control` and `InfluxDB-Logger`, which reads from a sound sensor [21]. In fact, the latter can be paired with any other sound-producing apps, such as those that control speakers, alarms, or music players.

Our findings show that there are 21 pairs (6.4%) that interact via sound but we could not find any conflicts among them. Typically, a pair consists of a sound-producing app and the `InfluxDB-Logger` app. Since the `InfluxDB-Logger` app only logs the status of the sound sensor, the two apps are actually *composable*—similar to the *composable* relationship in the *device* interaction (see Section 4.2.1).

Physical Factors. The *physical-medium* interaction depends on certain physical factors. The position of the first app's actuator relative to the second app's sensor determines whether the output from the actuator could reach the sensor. If their proximity is sufficiently close for the actuator's output to affect the sensor, the two apps interact; otherwise, they do not. When we performed this study, we assumed that their locations are sufficiently close. Although it is a conservative approximation, this is the best we could do and our findings can help developers and users to avoid such conflicts.

4.3.2 RQ3&4: Prevalence of Conflicts/Unsafe Coding

Table 4.4 summarizes the statistics for the *physical-medium* interaction pairs. Our findings suggest that typically, when a pair of apps interact through a physical medium, they will most likely conflict. In most cases, the second app does not expect to receive any input from the first app. It normally expects sensor inputs from its surroundings. Out of the 325 pairs with *physical-medium* interaction, 90.5% (294 pairs) of them have a conflict. We did

Table 4.5: Statistics for *global-variable* interaction.

Relationship	**# Pairs**	**Percentage**
Read-read	405	37.5%
Non-Conflicting Interactions		
Direct-direct write-write	28	2.6%
App write-read	302	27.9%
Direct write-read	221	20.4%
App-app write-write	1	0.1%
	552	**51.0%**
Conflicting Interactions		
App-app write-write	44	4.1%
App-direct write-write	80	7.4%
	124	**11.5%**
Total	**1,081**	

not observe any coding patterns that cause conflicts in this category. Hence, we concluded that the *conflict* in pairs with *physical-medium* interaction is caused mainly by the physical proximity between the actuators and sensors of the conflicting apps.

4.4 Global-Variable Interaction

This section presents our findings for app pairs that have *global-variable* interactions. As discussed in Section 4.1.1, since SmartThings only has one global variable `location.mode` that allows both reads and writes, we consider two apps to have *global-variable* interaction if they both access `location.mode`. Our statistics are reported in Table 4.5. 405 (37.5%) of the pairs (reported as pairs with *read-read* relationships in Table 4.5) contain apps that only read from `location.mode`. These apps do not *actually* interact.

4.4.1 RQ1: Types of Non-Conflicting Interactions

The first type contains apps that only write `location.mode` and they are controlled manually by the user. We refer to this as a *direct-direct write-write* relationship. As discussed earlier in Section 4.2.1, we did not consider these apps as conflicting since the user controls them. This group contains 28 pairs (2.6%), reported as pairs with *direct-direct write-write* relationships in Table 4.5.

A second type, consisting of 302 app pairs (27.9%), exhibits *app write-read* relationships, exemplified by the `Greetings-Earthling` and `Hello,-Home-Phrase-Director` apps [11, 4]. The `Greetings-Earthling` app changes the value of `location.mode` when the presence sensor detects that the homeowner arrives home. On the other hand, the `Hello,-Home-Phrase-Director` app sends a greeting message to the homeowner depending on the value of `location.mode`. In this case, the two apps have a *composable* relationship: one app reads the variable updated by the other.

A third type, consisting of 221 app pairs (20.4%), exhibits *direct write-read* relationships: one app requires the user to manually control the app to write into `location.mode`, while the other reads from it. This is the intended usage scenario of `location.mode`, namely to facilitate interactions between apps through mode changes. Hence, these *write-read* interactions are not conflicts.

Finally, we found one pair in which both apps write into `location.mode` and yet do not conflict. This pair consists of the `Greetings-Earthling` and `Bon-Voyage` apps [11, 9]. The `Greetings-Earthling` app writes into `location.mode` when the user arrives at home, while the `Bon-Voyage` app writes into the same location when the user leaves. Hence, they do not conflict as they have disjoint intents and never write at the same time. This is an exception to our current formal definition that can be improved.

4.4.2 RQ2: Types of Conflicting Interactions

When two apps both write into `location.mode`, in most cases, conflicts would result. There are two types of write-write conflicts: *app-app write-write* and *app-direct write-write.* For example, there exists an *app-app write-write* conflict between the `Smart-Security` and

`Good-Night` apps [16, 10], which both attempt to write into `location.mode`. While the `Smart-Security` app updates `location.mode` with *Home*, the `Good-Night` app changes `location.mode` to *Night* or *Away*. In `Smart-Security`, the update to `location.mode` occurs when intrusion is detected. This is rather an important update and the user certainly does not want the result of the `Smart-Security` app to be compromised. There are 44 pairs (4.1%) of such conflicts.

An *app-direct write-write* conflict occurs when in one app the update of the global variable is triggered by a non-user input, *e.g.*, a sensor, while in the other app the user performs an operation that triggers the update. For example, the first app uses the motion sensor to detect if there is anyone home and updates `location.mode` based on the sensor input. The second app lets the user control the light—when the user turns on the light, `location.mode` is automatically updated. This category has 80 (7.4%) conflicting pairs.

4.4.3 RQ3&4: Prevalence of Conflicts and Unsafe Coding

There are a total of 124 (11.5%) conflicting app pairs. Thus, conflicts are *not* prevalent for this type of interaction.

We found that *concurrent-writes* to `location.mode` is an unsafe pattern, which is due to the SmartThings APIs that allow apps to directly change the value of `location.mode`. For instance, in the case of the `Smart-Security` app, a good practice would be to not allow other apps to write into `location.mode` when the alarm is sounding; otherwise, the alarm may be stopped abruptly before it is noticed. For modes, the combination of (1) changing the API to specify a duration for the mode change and (2) allowing the user to specify priorities would resolve many of the conflicts.

4.5 IoTCheck: Automated Conflict Detection

In this section we address **RQ5:** How can we automatically detect conflicts?

We developed IoTCheck, a tool that automatically identifies conflicts by *model-checking* pairs of apps. A model checker checks, exhaustively and automatically, if a system meets a specification. Model checking is particularly useful in detecting app conflicts due to its ability to exhaustively check all potential interactions between apps.

We begin by summarizing the key insights from our manual study that we used for designing IoTCheck. Our study shows that most device conflicts occur when two apps issue conflicting updates to the same device. We found that when one app writes to a device feature and another app reads from the same device feature, it typically does not represent a conflict; this scenario commonly occurs when apps compose. We also found that it is important to consider the reason why two apps perform conflicting updates. If both updates are performed in response to user requests, there is typically no conflict since the actions are triggered by the user. Finally, we found that conflicts on global variables occur only when two apps both write to the global variable; read-write interactions typically represent normal cooperation between apps, *not* conflicts. IoTCheck model-checks pairs of apps and monitors for conflicting updates to the same device or global variables from different apps. IoTCheck directly executes the original app code, eliminating the need to build models of the apps. IoTCheck extends the Java Pathfinder (JPF), an explicit state-based model checking infrastructure [205].

4.5.1 IoTCheck Design

Architecture. Figure 4.2 presents IoTCheck's architecture. The arrows represent the workflow of IoTCheck that starts from app code as an input to the IoTCheck configuration tool and IoTCheck preprocessor. Each SmartThings app has a configuration method that asks users for configuration information—while most of the configuration can be automati-

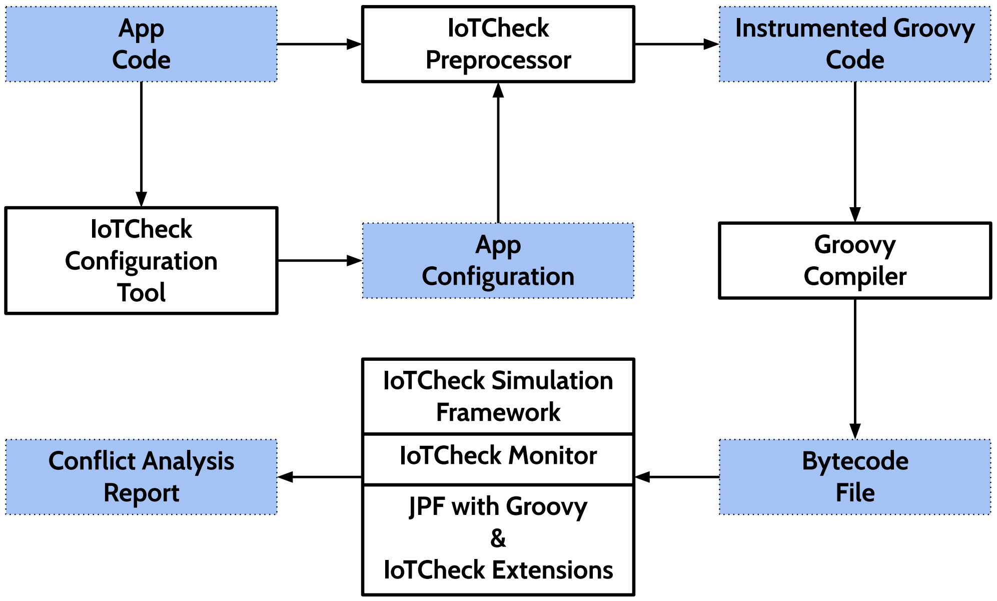


Figure 4.2: IoTCheck Architecture.

cally generated, apps can ask for arbitrary input and thus part of the configuration requires human help. The IoTCheck configuration tool runs this method, automatically configures most options, and asks the user for non-standard options. The IoTCheck configuration tool then outputs app configuration files, which, together with the original app, are processed by the IoTCheck preprocessor. The IoTCheck preprocessor generates model checker hooks to enable JPF to generate device events, combines multiple apps into the same program, and sets up the necessary configuration to run the program. It then outputs instrumented Groovy code which is compiled into bytecode by the Groovy compiler.

We developed a SmartThings simulation framework for IoTCheck. This framework contains virtualized devices (*i.e.*, device handlers) for all of the devices used by our benchmark apps. While an actual SmartThings device handler controls an actual device, a virtualized device handler changes the value of a state variable that represents the value of a device feature. Thus, a virtual device handler for a door lock changes the value of the door lock state variable instead of controlling an actual Zigbee door lock (see Figure 1.1). These device handlers are under the control of the JPF model checker—JPF triggers device events such as a motion detected by a motion sensor, or a temperature value change detected by a temperature sensor. For devices such as temperature sensors, there is a large range of potential temperatures that would make model checking infeasible without using symbolic techniques. IoTCheck thus supports a set of potential temperature readings (*e.g.*, a hot reading and a cold reading), which is practical given the nature of many smart home apps. IoTCheck does not currently model physical interactions between devices (other than to flag that they could potentially interact); this remains future work.

Finally, IoTCheck model-checks the generated bytecode using the JPF model checker. We developed IoTCheck monitor as a JPF listener that performs conflict analysis while JPF is executing the bytecode. When a conflict is detected, the listener halts JPF and immediately reports the conflict. Otherwise, JPF finishes its execution and the listener reports that there is no conflict.

$$S(n) = \bigcup_{\epsilon \in \text{in}(n)} \phi(A_e, \text{ismanual}(\epsilon), S(\text{in}(\epsilon))\phi(\emptyset, \lambda, S) = S$$

$$\phi(A; \text{write}(\alpha,\, d,\, \tau,\, r,\, v), \lambda, S) = \begin{cases} \text{conflict}, & \textbf{if}(\exists a \in \text{app}(S, d, r).a \neq \alpha \wedge \text{value}(S, d, r) \neq \\ & v \wedge \neg\lambda) \\ \text{update}(\phi(A, \lambda, S), \mathit{write}(\alpha, d, \tau, r, v)) & \textbf{otherwise} \end{cases}$$

$$\phi(A; \text{modewrite}(\alpha,\, \mu), \lambda, S) = \begin{cases} \text{conflict}, & \textbf{if}(\exists a \in \text{modeapp}(S).a \neq \alpha \wedge \text{modevalue}(S) \neq \\ & \mu \wedge \neg\lambda) \\ \text{update}(\phi(A, \lambda, S), \text{modewrite}(\alpha,\, \mu)) & \textbf{otherwise} \end{cases}$$

$$\text{update}(S, \mathcal{A}) = \{\mathcal{A}' \in S \mid \neg\mathcal{A} \triangleq \mathcal{A}'\} \cup \{\mathcal{A}\}$$

$$(\text{modewrite}(\alpha,\, \mu) \triangleq \text{write}(\alpha,\, d,\, \tau,\, r,\, v)) := \mathit{false}$$

$$(\text{write}(\alpha,\, d,\, \tau,\, r,\, v) \triangleq \text{write}(\alpha',\, d',\, \tau',\, r',\, v')) := (d = d') \wedge (r = r')$$

$$(\text{modewrite}(\alpha,\, \mu) \triangleq \text{modewrite}(\alpha',\, \mu')) := \mathit{true}$$

Figure 4.3: Conflict Analysis

Challenges. There are 3 challenges in extending JPF for IoTCheck:

(1) JPF does not provide out-of-the-box support for checking Groovy code. One challenge is that the Groovy runtime system keeps its own internal state that thwarts JPF's state matching algorithm; this often prevents even very simple Groovy programs from model-checking. IoTCheck extends JPF to consider only the state of the virtual smart home devices and the apps when matching states—it ignores state changes that are internal to the Groovy runtime library and do not affect the behavior of apps. This creates a second issue—JPF generates state matching points at many execution points. After eliminating Groovy runtime state from state matching, there can be spurious state matches terminating JPF before the state space is fully explored. To solve this problem, IoTCheck extends JPF to only match states right before generating a new event.

(2) Groovy is a dynamic language. Thus, method calls are resolved at runtime via Java Reflection—JPF was missing this feature and we had to extend it. Furthermore, the same call stack from the perspective of the program can be implemented by many different bytecode-level call stacks due to Groovy's method lookup and caching mechanisms. Since the call stack is considered by JPF's state matching algorithm, this can cause the algorithm to fail to match conceptually identical states and increase the state space to be explored. IoTCheck extends JPF's state matching algorithm to match conceptually identical call stacks with different bytecode-level stacks.

(3) Scalability is a challenge for JPF as an explicit-state model checker. IoTCheck initially exhaustively model-checks a app pair for up to 30 minutes. If it either a detects a conflict or completes, IoTCheck outputs the result and finishes. Otherwise, IoTCheck falls back on JPF's heuristic search and performs it for an extended 30-minute period. If no conflict is detected during this period or the tool runs out memory (usually caused by bigger apps that have tens of events), IoTCheck reports that the result is inconclusive. Future work can employ techniques such as partial order reduction to further improve IoTCheck's performance.

Detection. Conflicts cannot be directly checked on the executions JPF explores because state-based model checking is only guaranteed to explore all program states and transitions and not all possible paths through the state machine. Consider apps α_1 and α_2 where α_1 only turns the light on and α_2 can turn the light on and off. A conflict only occurs when α_1

turns the light on followed by α_2 turning the light off. However, all states and transitions can be reached without exploring this execution path. Thus, we must analyze the state machine and search for any conflicting path.

IoTCheck's conflict analysis is an online analysis of the state machine that JPF explores. Our analysis is similar to a standard dataflow compiler analysis with the exception that in our context nodes represent states and edges represent transitions. IoTCheck updates its analysis results as JPF explores new states and halts the exploration process when a conflict is detected. We abstract state machine as a set of nodes $n \in \mathcal{N}$ that represent the JPF states, and edges $e \in \mathcal{E}$ that represent transitions between JPF states. We denote sequences of actions using A (see execution trace definitions in Section 4.1.1). Each transition e has a corresponding sequence of actions A_e. The relevant actions are write(α, d, τ, r, v) and modewrite(α, μ). We define in(n) to be the set of incoming edges to n and src(e) to be the source node of the edge e. The analysis computes the set $S(n)$ of the most recent updates to each device feature and mode at node n. We define app(S, d, r) to be the set of apps that have most recently updated r on d and value(S, d, r) to be the value of that update. We define modeapp(S) to return the set of apps that have most recently updated the mode and modevalue(S) to return the values of the most recent update to the mode set.

Figure 4.3 presents equations that formalize our analysis. These equations are evaluated using a standard fixed point algorithm whenever JPF explores a new transition to either an existing state or a new state. Function ϕ applies the sequence of actions in transition to the set S for the previous node to compute the transition's contributions to set S for the destination node. The function update applies an action to set S.

Table 4.6: Comparison between manual study and IoTCheck.

Interaction	**IoTCheck**	**Manual Study**	
		Conflict	**No conflict**
Device	**Conflict**	679	38
	No conflict	33	101
	Not terminated	16	396
	Excluded	100	75
Global-Variable	**Conflict**	98	16
	No conflict	0	318
	Not terminated	0	388
	Excluded	26	235

4.5.2 Results

We repeated the same set of evaluations, but using IoTCheck to check for conflicts instead of manual inspection. Table 4.6 compares IoTCheck's results with those from the manual study. We did not use IoTCheck to detect conflicts in *physical-medium* interactions since these conflicts depend on physical factors.

For the *device* interaction, we initially found 829 conflicting pairs through manual study: 632 pairs with *feature conflict*, 76 pairs with *invalid-local-state* conflicts, and 121 pairs with *dropped-update* conflicts (see Table 4.3). From the 829 pairs, we had to exclude 100 conflicting pairs because of the 8 apps that we could not run on IoTCheck: 5 apps use third-party features and 3 apps have serious bugs. Because of these 8 apps, we also had to exclude 75 non-conflicting pairs. Overall, IoTCheck was able to find conflicts in 679 pairs but failed to detect conflicts in 33 pairs—a thorough manual inspection confirmed that 8 pairs are indeed non-conflicting (*i.e.*, mistakes in our manual study), while other conflicts were not detected due to IoTCheck's limitations (*e.g.*, in our modeling of time). It also did not terminate for 16 pairs labeled as conflicting in the manual study, but 4 of them are indeed non-conflicting. Surprisingly, IoTCheck found 38 *new* conflicting pairs that were overlooked in our manual study and labeled as non-conflicting. Thus, in total IoTCheck found 717 conflicting pairs.

For the 497 pairs labeled as non-conflicting in the manual study, IoTCheck confirms that 101 pairs are indeed non-conflicting, whereas it did not terminate for 396 of them.

For the *global-variable* interaction, our manual study found 124 pairs of conflicting apps: 44 pairs with *app-app write-write* conflicts and 80 pairs with *app-direct write-write* conflicts (see Table 4.5). With IoTCheck, we were able to find conflicts in 98 of the 124 pairs. We had to exclude 26 of the pairs with conflicts because of 6 apps that we could not run on IoTCheck: 5 apps use third-party features and 1 app has serious bugs. Additionally, IoTCheck found 16 pairs with a conflict that was initially labeled as a non-conflicting pair. Because we excluded 6 apps, we had to exclude 235 non-conflicting pairs initially observed in the manual study. Among the 706 non-conflicting pairs labeled in the manual study, IoTCheck was able to complete its check and found no conflicts in 318 of them. IoTCheck did not terminate for 388 of them. For the *physical-medium* interaction, IoTCheck generates a warning if one app uses a device that could be the physical input of a device used by the other app.

Statistics. The average runtime for IoTCheck to find conflicts is 27 seconds for the *device* interaction, and 11 seconds for the *global-variable* interaction. These suggest that conflicts are found quickly: the 30-minute time limit is enough to perform an exhaustive model checking in general. Thus, classifying non-terminating runs as non-conflict gives IoTCheck a precision of 100% and a specificity of 100%. The recall is 95.1% for the *device* interaction pairs and 100% for the *global-variable* interaction pairs. The overall recall of the two categories is 95.7%.

False Positives. The false positives/negatives in our manual study were typically due to subtle issues involving complex logic that had several conditions for generating commands or subtle concurrent executions. Due to page limits, we do not have space to provide a full accounting of these errors. We advise the interested reader to obtain IoTCheck and our dataset that provide greater detail of our results for both the manual study and automated conflict detection using IoTCheck [192, 190, 191].

Chapter 5

Related Work

5.1 Network Traffic Analysis and Defenses

Network Signatures for IoT devices. The research community has been actively looking into network traffic analysis techniques and network signatures. Work in this area was pioneered, among others, by Honeycomb [117]. Honeycomb is a system that generates signatures for malicious network traffic automatically. The system uses honeypots, namely decoy computer resources set up to monitor and log the activities of entities that attempt to compromise them. Honeycomb then deploys pattern-detection techniques and packet header conformance tests on traffic captured on the honeypots to identify malicious traffic and generate signatures for this traffic for future detections.

A body of work presented in a series of papers by Apthorpe *et al.* is the closest to our work on packet-level signatures. In their work, they have reported that a set of traffic features is useful to infer device type and activity: MAC addresses, DNS queries, and network traffic shape [40, 41, 42, 43]. They reported that MAC addresses and DNS queries are useful to determine the type of the device. The first three bytes of a MAC address is the organizational unique identifier that uniquely corresponds to a device manufacturer. Many of the DNS queries can also be mapped to a specific device or manufacturer/vendor: the queries oftentimes contain the name of the manufacturer/vendor and they are easily identifiable since a lot of the devices from the same manufacturer/vendor typically communicate with the same set of cloud servers.

Further, they also used network traffic shape (volume-based) signatures to infer the presence of IoT device activity. They discovered that network traffic fluctuations can be correlated with the activity that is occurring on the device. For example, they found a volume increase in the traffic generated by a WeMo plug whenever the plug is toggled ON or OFF. For a Nest camera, the traffic volume increases whenever a livestreaming session is happening. Unfortunately, while their volume-based approach can only infer the occurrence of *some* event, it cannot be used to precisely predict the type of the event. Nevertheless, they claimed that when a device only have a pair of event types, one could hypothetically associate a traffic volume increase with one type of the events and assume that the next volume increase correlates with the other event type: if a WeMo plug traffic volume increase is correlated with a “toggle ON“, then the next one would be a “toggle OFF“. Our packet-level signatures do not share the same weakness—one can infer the exact type of the device and event. Our signatures leverage packet lengths and directions that are unique for each device type and event, even for devices from the same vendor.

Furthermore, their signatures (that correspond to different traffic shapes) are intuitive, but not automatically extracted: the authors did not release any automated traffic analysis software tool. They collected traffic from a number of devices they experimented with. Then, they manually processed the traffic: they separated the traffic into different flows and divided these flows into time series vectors that are further divided into w-second windows. They further extracted 2-element feature vectors containing the mean and standard deviation of the traffic amounts in the samples of each window, and used a 3-nearest-neighbors classifier trained on these feature vectors. This allows them to determine the device type: they discovered that the mean traffic volumes of the Nest and Amcrest security cameras differed by almost an order of magnitude over the data collection period. For device activity, they had to manually observe the volume changes and correlate them with the occurrence of the device activity.

The authors proposed stochastic traffic padding (STP) to mitigate volume-based inference attacks. They also discussed two possible implementations of STP: a VPN-based approach that only defends against WAN sniffers and a device-based approach that defends against both Wi-Fi and WAN sniffers. Our preliminary results show that our packet-level signatures might potentially survive the proposed VPN-based STP implementation. In our experiments, it was revealed that OpenVPN consistently adds a 52-byte header and a 49-byte header for client-to-server and server-to-client packets respectively. Thus it does not completely hide packet lengths and directions. Nevertheless, these results are very limited because we only experimented with our VPN-based STP implementation with fixed parameters because we could not obtain the source code of their actual STP implementation.

HomeSnitch [152] by OConnor *et al.* is another related work closest to ours. HomeSnitch identifies IoT activity using a key observation that is similar to ours, *i.e.*, the client (the IoT device) and the server take turns in a request-reply communication style. HomeSnitch was mainly intended to enhance smart home transparency and control by classifying IoT device communication by semantic *behavior*: we call it *device event* in this work. HomeSnitch consists of two main parts: (1) behavior classification that classifies flows into known behaviors using supervised machine learning, and (2) policy enforcer that translates policy into network rules: it uses OpenFlow to send network flow information to a SDN controller application. HomeSnitch uses statistics derived from the entire client-server dialog traffic, namely from the features derived from *application data unit* (ADU) in four categories: throughput, burstiness, synchronicity, and duration. The authors selected thirteen features from transport layer headers that describe the client/server dialogues of smart home IoT devices. Then, they calculated the feature importance, namely a measurement of the predictive importance of each feature variable: each feature has a different level of contribution to the entire classification process. Finally, they used the Random Forest algorithm to perform the device behavior classification. The classification result is then used by HomeSnitch to enforce policy rules that prevent unwanted device behaviors and report malicious device behaviors.

HomeSnitch and PingPong both exclude IP addresses, port numbers, and DNS information from their event inference methodologies. However, they differ in terms of the granularity of the features they use. HomeSnitch uses aggregate network traffic statistics derived from the thirteen features generated by the ADU extraction from the entire client-server dialog. On the other hand, PingPong considers the direction and length of each individual packet. Interestingly, the most important feature used in HomeSnitch is the average number of bytes sent from the IoT device to the server per turn. This aligns with the main observation of this paper: packet lengths of individual requests (and replies) uniquely identify device events.

Measurement Studies. A recent paper by Ren *et al.* [166] presents a large-scale measurement study of IoT devices and reveals how these devices operate differently in the US and the UK with respect to Internet endpoints contacted, exposure of private information, *etc.* Their work attempts to address six research questions as their main objectives: (1) the destination of network traffic, (2) the extent of traffic encryption, (3) the data sent in plaintext, (4) the content sent using encryption, (5) the information a device exposes unexpectedly, and (6) the impact of the device's location on information exposure. Using 34,586 controlled experiments, they reported that a lot of devices: (1) have one or more destinations that is not a first party, (2) contact destinations outside their region, (3) expose information to eavesdroppers via plaintext flow, and (4) leak information about device behavior. The paper reported information exposure analyses for a total of 81 devices. However, they actually only used 55 distinct devices across their US and UK experimental setups.

We use their dataset to evaluate PingPong in Section 2.4.6. Surprisingly, we found that their dataset is suitable for our methodology. To answer the fifth research question: the information a device exposes unexpectedly, they built a classifier that can infer event types spanning many device categories despite having moderate effectiveness—they achieved F1 score as low as 0.75. For this experiment, they collected network traffic for the 55 devices labeled with the timestamps of each generated device event. We, however, do not compare PingPong with their classifier because it is not the focus of their work.

Aside from [166], we also used the datasets from other measurement studies to evaluate PingPong. Other well-known measurement studies and publicly available IoT network traffic datasets include YourThings [31, 32] and [180], which we use in our evaluation in Section 2.4.3. Despite having massive volume of IoT network traffic, including traffic related to device events, these other datasets are not labeled with timestamps. Therefore, unlike the dataset from [166], these other datasets cannot be used to verify PingPong methodology.

Other Protocols. Other papers consider specific types of devices or protocols. Copos *et al.* [66] analyze network traffic to infer the activity specifically for the Nest Thermostat and Nest Protect (only), and show that the thermostat's transitions between the *Home* and *Auto Away* modes can be inferred with 88% and 67% accuracy for each direction (*Home* to *Auto Away* and *Auto Away* to *Home* respectively). Thus, this reveals whether the home is occupied or not. Other work [28, 222] focus on Zigbee/Z-Wave devices and leverage specialized Zigbee/Z-Wave sniffers to collect traffic from these protocols. Since the Zigbee protocol for communicating with smart home devices is well documented, it makes the creation of the signatures easier than in our case. For instance, in [222] the authors used the application source code to correlate Zigbee traffic and device events. The byte-level information of a particular Zigbee command is typically hard-coded in the source code of the Zigbee device drivers.

Limitations of Other Techniques. Most event inference techniques rely on machine learning [132, 180, 179] or statistical analysis of traffic time series [66, 28, 152, 166]. Limitations of these approaches include: (1) the inability to differentiate event *types* [132, 180, 179] (*e.g.*, distinguishing ON from OFF), and (2) lack of resistance to traffic shaping techniques [66, 28, 152, 166] such as [40]. For example, in traffic volume signatures [40, 41, 42, 43], there is no obvious correlation between a certain traffic volume with a certain type of device event. However, the authors highlighted that this correlation is possible for device type inference. These approaches also rely significantly on the statistical analysis of the TCP/IP network traffic time series (*e.g.*, mean packet length, inter-arrival time, standard deviation in packet lengths, *etc.*). Thus, these signatures can easily be obfuscated by traffic shaping techniques, *e.g.*, once the traffic is shaped, traffic volume changes are not apparent anymore.

On the other hand, our work identifies simple packet exchange(s) between the device/smartphone and the cloud that uniquely identify event types. PingPong's classification performance (recall of more than 97%) is better than most statistical approaches: [28] reported 90% accuracy, [66] reported 88% and 67% accuracy, and [166] reported some F1 scores as low as 0.75. Unsupervised learning techniques may be hard to interpret, especially for large feature sets (*e.g.*, 197 features in [28]). PingPong also uses clustering to identify reoccurring packet pairs, but provides an intuitive interpretation of those pairs: they correspond to a request and the subsequent reply.

Network Traffic Analysis beyond IoT. There is a large body of work in the network measurement community that uses traffic analysis to classify applications and identify anomalies [150, 116, 114], attacks [74], or malware [34, 159]. There has also been a significant amount of work on fingerprinting techniques in the presence of encryption for web browsing [51, 102, 126, 63, 134, 158, 77, 55, 206, 157, 100], and variable bit-rate encodings for communication [214, 213] and movies [174]. The problems of using traffic analysis to fingerprint web browsing sessions or variable bit-rate encodings are quite different from extracting events from IoT device traffic. First, the fundamental structures of the communication protocols are different. For instance, IoT device traffic typically involves a fixed set of endpoints and predictable packet exchanges; hence, our discovery of packet-level signatures. On the other hand, web browsing sessions typically involve communications with a lot more variety of endpoints and significantly more complex packet exchanges patterns. Thus, instead of exploring the potential of minimal set of traffic features (*e.g.*, packet lengths and directions), a lot of web browsing fingerprinting techniques leverage statistical based analyses, *e.g.*, aggregate traffic volume, inter-arrival time, *etc.* Second, the volumes of data also significantly differ. While IoT devices have idle periods and typically generate less amount of traffic, web browsing sessions that occur on more powerful computing machines typically generate significantly larger amount of traffic in a more regular fashion. Thus, these problems require different analysis approaches that consider different features and use different techniques. Moreover, for these examples the underlying protocols are well understood, while PingPong can work with (and is agnostic to) any arbitrary, even proprietary, application-layer protocol.

Defenses against Packet-level Signatures. Related to profiling and fingerprinting is also the body of work on defenses that obfuscate traffic signatures. Examples include [158, 136] that use packet padding and traffic injection techniques to prevent website fingerprinting. In Section 2.5, we discuss two general defense approaches: (1) *traffic shaping* that refers broadly to changing the shape of traffic over time; and (2) *VPN* that brings multiple benefits such as encryption (that our signatures survive), and multiplexing of several flows. We partly evaluate these defenses—we obtained some preliminary results by implementing our version of VPN-based STP with fixed parameters to evaluate the effectiveness of packet-level signatures (see Section 2.5.5). A VPN also provides a natural place to implement additional defenses (*e.g.*, packet padding).

Packet Padding. Packet padding has already been studied as a countermeasure for website fingerprinting [126, 77, 53, 54]. Liberatore and Levine [126] showed that padding to MTU drastically reduces the accuracy of a Jaccard coefficient based classifier and a naive Bayes classifier, both of which use a feature set very similar to packet-level signatures: a vector of <direction, packet length> tuples. Dyer *et al.* [77] later showed that such padding is less successful against more advanced classifiers, such as the support vector machine proposed by Panchenko *et al.* [158] that also considers coarse-grained features such as total traffic volume. Cai *et al.* [53, 54] improved [77] by providing a strategy to control traffic flow that better obfuscates the traffic volume as a result. Although applied in a different context, these prior works indicate that packet padding should successfully guard against a packet-level signature attack. The question then becomes where and how to implement the padding mechanism.

PingPong in Perspective. Our work can be categorized within this broader area of network signatures for IoT devices, and as such our packet-level signatures have the following advantages over the other approaches: it (1) operates on possibly encrypted network traffic, (2) does not rely on application code or deep packet inspection, and (3) is potentially generally applicable across IoT devices.

Table 5.1 summarizes the properties of PingPong and compares it to other IoT traffic analysis approaches. PingPong combines *all* the following desired features: (1) unique signatures that can detect not only the occurrence of an event, but also the exact type of the event; (2) applicability across a broad range of devices, agnostic to their details; (3) applicability to two distinct adversaries (*i.e.*, a WAN sniffer observing IP traffic upstream from the wireless router, and a Wi-Fi sniffer observing encrypted Wi-Fi traffic on the local Wi-Fi network); (4) new signatures (consisting of minimal information of packet length and direction) that (i) are intuitive, *i.e.*, capture the request and reply associated with events, (ii) can be automatically extracted from training datasets, and (iii) are simpler than previously known signatures, allowing for more lightweight detection; (5) seemingly resilient to state-of-the-art defenses

Table 5.1: PingPong's properties vs. alternative approaches (✓= Yes; × = No).

	Approaches for IoT Network Traffic Signatures						
	Volume + DNS based [42, 43, 41, 40]	Nest device [66]	Machine Learning			Zigbee/ Z-Wave device [222]	PingPong
			[152]	[28]	[179] [180]		
(1) Signature can detect							
Device type	✓	✓	✓	✓	✓	✓	✓
Event type	×	✓	✓	✓	×	✓	✓
(2) Applicability to devices							
> 15 Models	×	×	✓	✓	✓	×	✓
(3) Observation points/threat models							
LAN	×	✓	✓	✓	✓	N/A	✓
WAN	✓	×	×	×	×	N/A	✓
Wi-Fi	✓	×	×	✓	×	N/A	✓
(4) Signature characteristics							
Features	Traffic volume, DNS	TCP connection size, protocol	13, ADU	(795) 197	12	Packet length & direction	Packet length & direction
Interpretable	✓	×	✓	×	×	✓	✓
Automated Extraction	×	×	✓	✓	✓	✓	✓
(5) Resilient against defenses							
VPN	×	×	×	×	×	N/A	✓
Traffic shaping	×	×	×	×	×	N/A	✓

such as VPN encryption and traffic shaping. Limitations and further discussion of our approach can be found in Chapter 6.

5.2 Smart Home and IoT Security

Devices and Protocols. Denning *et al.* [70] made the first attempt to identify emergent threats to smart homes due to the use of IoT devices. A recent work from [86] discusses these threats in more concrete contexts, for example, by demonstrating scenarios in which hackers can weaken home security through compromising these devices. The authors investigated the SmartThings smart home platform for security and vulnerability issues. For instance, these devices do not have sufficient sensitive event data protection. Events could leak sensitive information through capability-based access (*e.g.*, battery status events could leak pin code) or device identifier. The environment is also prone to event spoofing, *e.g.*, the authors experimented with generating fake smoke detection events. Furthermore, the environment has insecure third-party integration: HTTP endpoints are used for third-party access. Although OAuth-based authentication service is used [151], the authors found that there are apps with client ID and secret hard-coded in bytecode. On top of these issues, the programming environment also exhibits an unsafe use of dynamic method invocation. For instance, a method `foo()` can be called using the string `"foo"`. This string is often sent via the HTTP protocol in clear text, which makes it vulnerable to command injection attacks.

The research community is also actively looking into how the improperly secured IoT devices can allow them to be used to mount large-scale attacks. Recent work shows that if an attacker can control enough high-wattage IoT devices, the attacker can cause power grid failures [183, 142, 76, 67]. A recent study by Antonakakis *et al.* provides a comprehensive insight into the recently occurred Mirai botnet attack. In the attack, approximately 64,500 devices were infected by Mirai botnet in just 20 hours—approximately 600,000 devices were infected during the peak of the attack in November 2016. With enough devices being infected, the attackers performed a massive distributed DoS attack that crippled a few cloud services providers, such as Krebs on Security, OVH, and Dyn. This in turn brings down a number of well-known websites, such as Amazon, Spotify, *etc.* Antonakakis *et al.* discovered that a bot would scan the legacy telnet port (port 23) and port 2323 of the IoT device that is potential for the attack. If any of the ports is open, the bot would attempt to login by brute force. The attack was successful because a lot of these devices used default telnet passwords for login—the Mirai source code release included 46 unique passwords, some of which were even traceable to a device type and vendor.

There are two main categories of work in current smart home security research, focused primarily on devices and protocols, respectively. Work on devices includes the MyQ garage system that can be used as a surveillance tool to inform burglars when the house is possibly empty, the Honeywell Tuxedo touch controller that has authentication bypass bugs and cross-site request forgery flaws [89, 103], as well as compact florescent lamps that can induce seizures in epileptic users [153]. A study by Ur *et al.* [200] on the access control of the Philips Hue lighting system and the Kwikset door lock found that each system provides a siloed access control system that fails to enable essential use cases such as sharing smart devices with other users like children and temporary workers. On protocols, studies found various flaws in the ZigBee and Z-Wave protocol implementations [93, 131] as well as design flaws in their programming frameworks [86]. Bluetooth devices face similar issues to Zigbee devices regarding access control. Previous work [121] has explored access control for Bluetooth devices. Low-level protocol differences mean that the solution for Bluetooth devices does not solve the problem for Zigbee. A study from Veracode [202] on several smart home hubs found that the SmartThings hub had an open telnet debugging interface that can be exploited, while Fernandes *et al.* [86] discovered framework design flaws in the SmartThings platform.

Overprivilege and Applications. The research community has also recently looked into smart home apps [85, 87, 61, 60, 49, 149, 222, 188, 30]. Fernandes *et al.* present a thorough study on the SmartThings environment [85]. They pointed out underlying security issues and a simple program analysis to detect the *overprivilege* issue in the app source code. They also found that IoT devices are also overprivileged due to the framework design itself. In [87], Fernandes *et al.* investigated and discovered that applications can leak confidential information.

Since many IoT security problems arise due to overprivileged network accesses, much work has been done to limit the privilege of a networked system. However, this is difficult to achieve due to the lack of programming language and system support. For example, Felt *et al.* [83] analyzed many Android apps on the privilege they were given. Using Stowaway, an automated tool that they developed in their study, they discovered that 323 out of 940 Android apps (approximately 35.8%) are overprivileged. They also discovered that more than half of these apps have 1 extra permission and 6% request more than 4 unnecessary permissions. Au *et al.* [44] developed PScout, a static analysis framework for Android source code to produce complete permission specifications for different Android versions. The authors discovered that Stowaway executes APIs in apps and, thus, it does not analyze the apps comprehensively. PScout, instead, performs a static analysis on the Android app source code and extracts specifications from the API source code to provide a more comprehensive analysis. They specifically highlight that although there is a small percentage that permissions could be redundant and undocumented APIs are used, most APIs only require a simple permission scheme: more than 80% APIs only need 1 permission.

Roesner *et al.* [170, 169] introduced User-Driven Access Control, which involves the user at the moment an app uses a sensitive resource. For instance, a remote control door lock app should only be able to control a door lock in response to user action. However, certain devices and apps are better suited to install-time permissions. Felt *et al.* introduced guidelines on when to use different types of permissions [84].

Fernandes *et al.* [86] conducted a similar analysis on smart home IoT devices and found that overprivilege also exists in IoT devices and much of the overprivilege is due to the framework design itself. They investigated 499 apps and 132 device handlers using static analysis: they found that 55% do not entirely use the requested capabilities and 42% are granted the capabilities that are never requested. They discovered that the problem is twofold: (1) coarse-grained capabilities, and (2) coarse binding between app and device. HomeOS supports smart home devices using a PC-like abstraction [73]. HomeOS only provides support for placing restrictions on modules running on the PC—other devices on the network are free to attack smart home devices. Vigilia provides much stronger security guarantees—it can defend against attacks from any arbitrary components on the home network.

FlowFence [87] provides security by requiring consumers of sensitive applications to declare their data flow patterns. It provides IoT applications with the ability to perform computation on the sensitive IoT data, while mitigating data abuse at the same time. To achieve this, the authors devised a technique called opacified computational model that consists of two parts: (1) Quarantined Modules, (2) opaque handles. A Quarantine Module is a code module written by developers. This module performs computation on sensitive data, which is assigned a taint label at the data source—the module runs in a sandbox that is provided by the system. An opaque handle is an interface with the non-sensitive side. It does not reveal any information about the data value, data type, data size, taint size, or exceptions that could occur to the non-sensitive code. This mechanism provides a means to track sensitive data and information flow. It also prevents information leak from the sensitive data as this sensitive data computation always occurs in a Quarantine Module. Quarantine Modules and opaque handles are associated with a set of taint labels that indicates data source and helps track information flows. FlowFence has the authority to make security decisions over sensitive data: all declassification of sensitive data can only occur through trusted APIs provided in the framework. Thus, this framework uses a similar concept that has been explored by the research community in the area of information flow control (see Section 5.4).

ContexIoT [113] is a context-based permission system provides contextual integrity by supporting fine-grained context identification for sensitive actions. Thus, ContexIoT is orthogonal to the technique presented in FlowFence. context. ContexIoT tries to provide contextual integrity—sensitive action has to match trigger time and user's intention. ContexIoT patches the app code with additional code that will ask for user's permission if the context and the request appear to breach the security requirements, *e.g.*, an app that requests to open the bedroom window when the temperature is high and the user is sleeping. ContexIoT consists of two main parts: (1) static analysis (intra-procedural context analysis and code patching), and (2) runtime logging for dynamic/runtime values.

Network and Routing Policies. Network-based policy checking is not a new idea and it has been studied in the *software-defined networking* (SDN) community [94, 59, 58]. SANE [59] and Ethane [58], a successor of SANE, are systems developed to manage policies in enterprise networks. With ACLs, packet filters, and other security measures that were intended to secure network perimeters, enterprise networks become too inflexible and very hard to maintain. Administrators have to manage a large set of policies that easily need changing as the network devices change. The SANE creators intend to mitigate the burden and move this responsibility to the network itself. Based on capabilities that are enforced at each switch, the network is able to make routing decisions. These are done automatically, without compromising the security, and involving any human intervention. For example, Ethane [58] requires each application to specify a manifest of its required communication and then checks packets against security rules and installs forwarding rules as required. While Ethane is applicable in our setting, it is designed primarily for enterprise networks that have a large number of switches; it requires a sophisticated controller that performs authentication, registration, and checking. It also requires expert administrators to develop routing policies—a task that is beyond the abilities of most end users. Moreover, IoT devices typically communicate via Wi-Fi and often do not support the enterprise security modes. Thus, malicious devices can masquerade as other devices to bypass the SDN protections. HanGuard [69] also uses SDN-like techniques to learn the normal traffic between smartphone apps and their respective smart home devices. HanGuard has a Monitor app runs on the phone to identify any attacks. It then informs the router about the attacks through the system's control plane. The router subsequently enforces policies in the data plane after verifying the party that attempts to access the device—if the attempt is identified as an attack (or as an activity that is malicious), it will then be rejected by HanGuard.

Many other projects have made a similar observation that IoT devices have highly structured communication patterns. The research community, thus, developed toolkits for managing firewalls to manage the communications between these devices: IoTSec and IDIoT [184, 46] are the closest to Vigilia. They use a similar concept to Vigilia's that firewalls are useful to constrain accesses and communications that occur through a router between IoT devices. However, their policy management appears to require users to specify allowed and rejected network properties at a very low level—Vigilia, instead, derives the policy and, thus, firewalls directly from the application (source code) level. IoTSec has two phases: profiling and deployment. During profiling, it attempts to learn the normal traffic of devices, *e.g.*, legitimate source and destination IP addresses, port numbers, protocols, *etc.* Then, a set of firewall rules will be generated and can be deployed on the router. Similarly to Vigilia, IoTSec reduces the attack surface with firewall while trying to maintain full functionality of devices.

The Firmato [47] toolkit allows administrators to specify rules in terms of a higher-level model. This model is specified by the administrator and thus has no direct relationship to code— errors in specifying the model can either open the system to attacks or block desired communications. Moreover, it is likely to not be reasonable to expect end users to develop such models for their home networks.

The Bark policy language uses manually created policies [107]. This policy language provides five types, *i.e.*, *who*, *what*, *where*, *when*, and *how* to capture the high level information (*e.g.*, devices, apps, types of service, etc.) needed to construct network level policies. Other approaches, such as IoT Sentinel, propose learning policies [219, 140]. IoT Sentinel [140] provides mitigation for devices with vulnerabilities. It identifies vulnerable devices in the system, controls traffic flows, and constrains communication for such devices. It consists of a security gateway (router) and a security service. The security gateway performs traffic monitoring and control, and device fingerprinting. The security service classifies device type, performs machine-learning on the collected fingerprints, and assesses vulnerability. The vulnerability assessment will categorize vulnerable devices in different isolation levels. IoT Sentinel specifies three levels for isolation: strict, restricted, and trusted.

Firmato, Bark, and the other approaches suffer from similar challenges to IoTSec, in that they can generate overly relaxed policies that allow attacks or overly restrictive policies that break applications. Moreover, compromised devices can easily bypass the policies by masquerading as other devices. For simple IoT control rules of the form used by IFTTT, automated analysis can generate rich policies that only grant permissions under specific criteria (e.g., one can only turn on the heat if it is cold) [88].

SIFT [125] is a related work that is orthogonal to the other related work in rules and policy generation. The authors made an observation that IoT applications (in the form of rules and policies) can introduce conflict and safety issues. SIFT checks app rules and policies for conflicts. It first transforms rules into intermediate representation that becomes an input to the SMT solver. The process also combines model checking with symbolic execution to find specific instances of environmental inputs that can trigger conflicts.

Automatic routing policy derivation and deployment have also been a critical goal of Vigilia. To avoid mistakes in manual setup caused by human errors, Vigilia is designed to be able to derive these policy rules based on the network's security invariants, and the specifications of sets and relations of devices. While SANE or Ethane derives its routing policies dynamically based on the network conditions and actual packets at runtime, Vigilia statically derives all the security policies from sets/relations and information stored in a database. This approach is suitable for IoT devices as each device serves a single purpose and thus generates monotonous traffic.

5.3 Interactions of Smart Home Applications

Data Races. Interactions of smart home apps may appear similar to those of concurrent programs, including data races [175, 65, 64, 80, 127] and atomicity violations [135, 217, 91]. Earliest work in race detections is based on Lamport's concept of *happens-before* relation [118]. Eraser [175] is among the pioneering work in the area of data race detection. Eraser dynamically tracks the set of locks assigned and held at runtime during program execution. It computes the intersection of all locks held when involved threads access a shared memory location. Thus, when a shared location does not have any intersection, it is flagged as not being properly protected. Intersection set is commutative, so Eraser can flag errors regardless of the actual interleavings between the involved threads. Eraser has been improved by Choi *et al.* by incorporating static analysis to remove unnecessary checks during runtime analysis [65]. This technique tremendously reduced the overhead of Eraser to at most around 40%—it was originally a factor of 10.

Engler *et al.* took a different approach and developed RacerX, a race detection tool that is based on static analysis. RacerX is a static tool that detects race conditions and deadlocks using flow-sensitive, interprocedural analysis. In general, RacerX traces locks based on the operations they protect, code contexts that are multithreaded, and shared accesses that are dangerous. It also tracks code features that are useful to sort errors based on their severity. As counter analysis to avoid mistakes, RacerX uses novel techniques: (1) result selection based on the most trustworthy paths, (2) decision cross-check in different ways, and (3) semaphore inference to differentiate between mutual exclusion and unilateral synchronization.

The research community has also looked into race detection of other programming languages and built detection tools—data race detection tools are mostly developed for C/C++. Cheng *et al.* developed a race detector for Cilk programs [64]. Flanagan *et al.* developed a race detector for Java programs [90].

Atomicity Violations. Atomicity violations could be another problem that occurs although a critical region has been properly protected using locks. Locks can guarantee that a shared memory location is not accessed by multiple threads at the same time. However, the interference between concurrent threads can still cause a problem. Different threads may access this shared location at different times (race condition free), but they could still interfere with the same shared location through other functions and procedures in the program; this may leave a shared location with a random value that could cause the program to throw an exception.

Flanagan *et al.* developed Atomizer [91], a dynamic analysis tool that detects atomicity violations by verifying that every execution of a code block annotated as being atomic is not affected and does not interfere with other running threads. Atomizer takes an aggressive approach: instead of waiting for the signs of erroneous program behaviors, it is preemptively looking for evidence of atomicity violations with the potential of causing future errors.

Lu *et al.* studied and developed a methodology to test the interleaving space and expose atomicity-violation bugs [135]. The authors designed and evaluated a number of interleaving coverage criteria through the use of real-world concurrency bugs. They formulated a coverage criterion, called *unserializable interleaving coverage*, that they used to study the effectiveness of stress testing. Finally, they designed CTrigger, a tool that deploys the unserializable interleaving coverage criterion.

Yoga *et al.* proposed a dynamic analysis technique to detect atomicity violations in task parallel programs [217]. Similarly to thread-based programs, interference between two tasks that execute in parallel can also cause data races and atomicity violations. In this case, the two tasks can logically execute in parallel, access the same location, and at least one of them performs write operation. Their proposed technique detects atomicity violations through appropriate metadata and dynamic execution structure of task parallel executions. The technique maintains a history of multiple tasks that dynamically access a shared memory location. Using the access history, it determines if memory accesses that are performed by parallel tasks are conflict serializable.

Data races can be resolved by acquiring locks appropriately, while atomicity violations can be resolved by ensuring that locks are held long enough to guarantee that multiple threads do not interfere with one another when accessing the critical region. These techniques, however, cannot resolve the above-mentioned conflict of smart home apps. They could, instead, disable the desirable functionality of the apps that attempt to access a shared memory location and render the apps useless.

Feature Interaction. Feature interaction focuses on the interactions between different software features [56, 37, 112, 156, 39, 38]. In this case, these different software features can have negative interactions.

In their position paper, Apel *et al.* [37] explored the nature of feature interactions systematically. The authors comprehensively classified feature interactions in terms of order and visibility. In particular, they attempted to understand the efficiency of interaction-detection and performance-prediction techniques. They presented a set of preliminary results. They also presented a discussion of possible experimental setups.

Jackson *et al.* [112] reported their findings on feature interactions in telecommunications services. They presented Distributed Feature Composition (DFC) as a new technology for feature specification and composition. In the DFC architecture, calls from customers are processed dynamically through assembling configurations of filter-like components. Each component implements an applicable feature and communicates with other components through internal calls.

Oster *et al.* [156] explored a similar feature interaction problem in the domain of *Software Product Lines* (SPLs). In this domain, features interact, exchange information with other features, or influence one another. Thus, an adequate test criterion is necessary to increase coverage for various interactions between different features. Their technique combines a combinatorial design algorithm for pairwise feature generation with model-based testing.

The purpose is to reduce the size of the SPL and to achieve a comprehensive coverage for feature interactions.

Apel *et al.* [39] investigated whether feature-based specifications can be used to detect feature interactions. Feature-based specifications aim to achieve modularity in feature-oriented systems. The authors addressed the problem of how much the modularity of specifications can affect the detection of feature interactions. However, some of these specifications required workarounds as they were not properly modularized.

In [38], the authors proposed a novel software design paradigm, namely *feature-oriented design*. *Feature-oriented software development* (FOSD) considers a systematized feature mapping, and the tendency of features to have unpredictable interactions. The authors created FeatureAlloy, an extension to the lightweight modeling language Alloy, that has a support for feature-oriented design. They demonstrated that the feature-oriented design techniques deployed in FeatureAlloy facilitate separation of concerns, variability, and reuse of models of individual features. The techniques also help with the definition and detection of semantic dependence and interactions between features.

Compared to related work in feature interactions in these areas, the smart home setting is different and unique. Smart home apps are developed independently by different app developers. They are also composed separately by end users. In the context of the SmartThings platform, the smart home apps are distributed through many different channels; these include pay-for-source channels. In other words, the community (developers and end users) does not have a means to detect, resolve, and avoid conflicts during the development phase of these apps.

In the context of smart home, feature interactions have also been studied [120, 210, 163]. Nakamura *et al.* [120] used an object-oriented approach, in which each networked appliance is modeled as an object. This object consists of properties and methods. The authors define two types of feature interactions: appliance and environment interactions. An appliance interaction occurs on an appliance object when different services try to invoke methods and this results in incompatible updates or references for common properties. An environment interaction occurs when methods of various appliances conflict through environment object—this happens indirectly.

Wilson *et al.* [210] studied the side-effect problem in the interaction between components in environmental modelling. While previous work introduced a device-centric approach that detects undesirable behaviors when there are interactions between appliances, the authors argue that some appliances also affect the environment through their side effects, *e.g.*, an air conditioner that decreases the humidity in addition to cooling the air as its main function.

Rajan *et al.* [163] proposed test oracle specifications. Due the dynamic nature of smart home environments, many of the services inside a smart home may change their configuration at runtime. This presents testing challenges, namely the specification of test oracles. The authors gave the formal specification of test oracles in the JML specification language. In the presence of dynamic reconfigurations in a smart home, the proposed mechanism notifies dynamic changes along with the runtime evaluation of JML specifications.

Although this body of work also has smart home as a context, what they used was early research prototypes with home automation. In these early systems, the richness and complexity of the smart home environment do not compare with those of modern smart home platforms, such as the SmartThings platform. Since they were prototype systems, they presumed much simpler and coarser apps. For instance, a smart home app could be a single app that controls a light; this app does not serve any other functionality. HCI researchers have shown that feature interactions in IoT systems make it difficult for users to understand the systems' behavior [216]. In rule-based smart home systems, researchers have developed tools for repairing incorrect rules [147].

Interactions of Mobile Apps. The research community is actively studying the interactions between Android apps [52, 198, 57, 101, 122, 45, 119]. Bagheri *et al.* [45] presented a technique that enables end-users to safeguard apps installed on their device from attacks. These attacks exploit the vulnerabilities in downloaded and installed apps on Android platforms—it is hard to foresee these attacks as developers would not know which apps will be installed together by end users. Based on these findings, the authors created SEPAR, a static analysis tool that uses formal method techniques to infer security properties from a group of apps automatically. It subsequently uses a constraint solver to predict possible exploits and derive fine-grained security policies as a means for protection for these apps.

A body of work focuses on the analysis of the Android message passing mechanism called Inter-Component Communication (ICC). Inter-app ICCs have the potential to be abused by malwares to launch collusion attacks using multiple apps, but the analysis complexity prohibits concrete security evidence. Bosu *et al.* [52] reported their findings in the first large-scale detection of collusive and vulnerable apps. Their design aims to achieve accuracy, as well as runtime scalability. Still in the domain of multi-app vulnerabilities, IACDroid [57] deploys an inter-application analysis technique to detect sensitive data leakage. IACDroid was tested on DroidBench and IAC Extended DroidBench datasets—it yields high accuracy. LinkFlow [101] leverages taint analysis technique to enumerate ICCs that could lead to privacy leakage in each individual apps. Most ICCs are normal, so they can be excluded in the next step analysis. This allows LinkFlow to perform large-scale leakage detection among a large set of apps, which was still a challenge prior to this work. SEALANT [119] combines static analysis, which discovers vulnerable communication channels, of app code with runtime monitoring of inter-app communication that occurs through those channels. This helps to prevent attacks. ApkCombiner [122] combines different apps into a single apk. With this, existing tools can indirectly perform inter-app analysis. IoTCheck also combines multiple apps into one app before using JPF to perform model checking on the combined app. JITANA [198] analyzes interacting Android apps simultaneously.

These techniques focus primarily on cross-app information flow and taint analysis via ICC/IAC mechanisms in Android (*e.g.*, Intents). Their settings are different from the setting for smart home apps. The smart home apps problem requires checking execution trace (representing the progress of program states) and its properties (*e.g.*, the changes of shared memory locations). Hence, the techniques developed for Android ICC/IAC analysis would not be effective to handle these specific requirements.

Event-Based Races: Mobile Apps. The interactions of smart home apps also has some similarity to event-based races in mobile apps [108, 165, 138, 109, 50]. Bielik *et al.* [50] presented a complete dynamic analysis system that is capable of finding data races in real-world Android applications in just minutes. The system the authors created was based on three key concepts: (1) a thorough *happens-before* model of Android-specific concurrency, (2) an analysis algorithm that is scalable for efficiently constructing the *happens-before* graph, and (3) a domain-specific filter that effectively reduces the number of reported data races (by several orders of magnitude).

Hu *et al.* [109] introduced SIERRA, a static analysis tool with precision and scalability to detect Android event-based races. SIERRA is based on a novel concept called *concurrency action* that "reifies threads, events, messages, system, and user actions". It then statically derives order in terms of *happens-before* relation for the tested Android apps.

Hsiao *et al.* [108] presented a race detection tool, CAFA, for event-driven mobile applications. CAFA accounts for the causal order due to the event queues. This is not accounted for in past data race detectors. CAFA also overcomes the problem of detecting races based on low-level properties between memory accesses that leads to a large number of false positives by, instead, checking for races between high-level operations.

Raychev *et al.* [165] developed EVENTRACER, a tool built on their novel technique that addresses two problems: (1) large number of false positives and (2) scalability challenge.

The authors introduced *race coverage*, a method that systematically exposes ad hoc synchronization and other potentially harmful races to the user to reduce false positives. They also presented an algorithm to compute race coverage.

Maiya *et al.* [138] implemented DROIDRACER, a tool that tests Android applications and generates execution traces. DROIDRACER detects data races by computing the *happens-before* relation on the traces. The authors also formalized the concurrency semantics of the programming model and defined the *happens-before* relation for Android applications.

Related work on mobile apps typically deals with one app that has multiple events. On the other hand, our work focuses on the interactions between multiple apps. The event and event handlers in smart home apps are developed by different programmers who are uncoordinated. Furthermore, a number of apps also allow the user to generate arbitrary events. Even though the ordering between events in one app can be clearly defined, the ordering between events across multiple apps combined with user-generated events is completely arbitrary. Thus, the techniques presented to synchronize events in individual apps would not be useful in this context.

Concurrency in Smart Home Apps. The research community has been actively looking into security bugs and issues in smart home apps. They conducted studies and developed novel techniques to detect these issues [85, 87, 61, 60, 49, 222, 188, 30, 130]. Unfortunately, interactions and conflicts between multiple apps are not the focus of this body of work.

Researchers have presented new techniques to model-check and analyze confidential information leakage in smart home applications. The techniques presented in [149, 148] require translating the apps to perform the model checking using SPIN [106]. The limitation is that the expressiveness of app features could be lost in translation: with just 3 apps the authors found 1 feature that their system could not express concisely [148]. Other work [61, 60, 49] ignores internal application state, and thus admits executions that cannot happen.

Several of our apps depend on internal state to decide whether to perform an action, and thus they would not be accurately modeled by their techniques. While conflicts between apps are discussed in [61], they considered a much smaller corpus of apps and a number of of them are self-crafted to generate the intended conflicts. Unfortunately, their system is not publicly available.

There have also been efforts to resolve the conflicts between smart home apps from the perspective of dependencies between application components at the system level [204, 212, 203, 143, 215]. This body of work, which was done in the cyber-physical systems community, attempts to identify and resolve conflicts between smart home apps at the system level, viewing apps as *black boxes* Several systems [204, 203, 212] provide frameworks for programming networks of sensors and actuators. DepSys [143] provides infrastructure with comprehensive strategies to specify, detect, and resolve conflicts through the use of user-specified metadata. Kripke [215] performs conflict detection through the use of model checking.

While such techniques are useful in certain simple scenarios, they are still semantics-agnostic and do not work even for the above-mentioned conflicts. Understanding the semantics of the apps (*e.g.*, scheduling of events) is key to build a tool that can automatically detect the conflicts—they can typically be exposed only when a specific set of events occur in a certain order. Our work is orthogonal to this body of work that attempts to deal with conflicts between apps at the system level, by viewing apps as black boxes. Our work, on the contrary, studies how apps interact and what can be done at the source code level to mitigate conflicts.

IA-Graph [123, 124] studies smart-home app conflicts and proposes a lightweight approach to check for conflicts. This work extracts an SMT formula that describes the legal transitions for an app and then uses an SMT solver to detect whether a set of apps has conflicting transitions. As acknowledged in the IA-Graph paper, IA-Graph "ignores complicated computations in the app code"—they are, in fact, used either (1) in condition statements, or (2) to update the device state—and hence the patterns it finds are limited. In addition, not all transitions in an app can be expressed in SMT, further limiting the kinds of conflicts IA-Graph can detect. Another important drawback is IA-Graph does not check whether a conflicting transition is reachable in an execution and hence can produce many false positives—it may incorrectly label non-conflicting apps as conflicting. Understanding the impact of these issues is quite difficult without accessing their implementation. Unfortunately, the authors did not release any software—thus an empirical evaluation is not possible. Furthermore, they did not perform any wide-scale study.

5.4 Other Related Work

Capability-based Object Model. The concept of capability-based object model is used to limit accesses to a certain object based on the set of capabilities it provides. It was first coined in 1959 by Dennis *et al.* [71] in which a hierarchical structure of object names is presented in programming semantics for multi-programmed computers. Miller *et al.*, [141] compared access control list (ACL) with capability-based model. In [137, 136] capability-based model has been used to secure web browsers, while [172] compares features and capability based model.

Conceptually, in many of those systems, capabilities are resolved dynamically as needs arise. Joe-E [139] demonstrates how it is possible to achieve the strong security properties of an object-capability language while retaining the features and feel of a mainstream object-oriented language. Unix based operating systems, such as Linux, have fundamentally implemented ACL that is orthogonal to capability-based object model. SELinux [133] adds support for mandatory access control policies to Linux. Policies are specified by an administrator and they capture the behaviors a program is allowed to perform. Errors in policies can cause the operating system to block operations, wasting user time debugging issues and in many cases resulting in SELinux ultimately being disabled.

Vigilia provides in-depth security guarantees through capability-based object model. Vigilia RMI compiler generates stubs and skeletons for each object based on the capabilities that the stub can access on the object that controls a certain device. Although it is generated statically when the stub and skeleton are created, the filter checks method invocations dynamically at runtime.

Type System and Information Flow Control. Vigilia deploys a type system in its capability-based RMI mechanisms that set up constraints for communication between components. The stub and skeleton code generated by Vigilia use a filter to verify that the remote method invocations come from components that have been granted accesses to the appropriate capabilities. Type systems have been used to control information flow and avoid vulnerabilities in several previous works. JFlow [144], is a language that is based on Java and it analyzes information flow before it translates its source code into Java code. A programmer needs to use labels and write policies on variables, methods, classes, *etc.* The Java Checker Framework [72] is another work that provides a framework to write type checkers for specific needs in Java, *e.g.*, null dereferences, equality operators, reference/object immutability, *etc.* Basically, Vigilia 's type system checks and forbids remote object passing from one process to another through its RMI compiler that generates two interfaces.

Operating systems [220] and distributed systems [221, 129] have used information flow to enforce similar properties. Fabric [129] extends the JiF language with support for transactions and remote calls and can be used to enforce information flow security on distributed systems. The techniques developed by Vigilia are largely orthogonal to those in Fabric: Vigilia assumes that component implementations may have bugs and thus focuses on constraining communication between components while Fabric relies on the correctness of components and instead focuses on information flow properties. This body of work is largely orthogonal to Vigilia.

Chapter 6

Conclusions

6.1 Summary

In this dissertation, we present a passive inference attack we discovered and studied on smart home devices: packet-level signatures. We designed, implemented, and evaluated PingPong, a methodology for automatically extracting packet-level signatures for smart home device events from network traffic. Notably, traffic can be encrypted or generated by proprietary or unknown protocols. This work advances the state-of-the-art by: (1) identifying simple packet-level signatures that were not previously known; (2) proposing an automated methodology for extracting these signatures from training datasets; and (3) showing that they are effective in inferring events across a wide range of devices, event types, traces, and attack models (WAN sniffer and Wi-Fi sniffer). We have made PingPong (software and datasets) publicly available at [194]. We note that the new packet-level signatures can be used for several applications, including launching a passive inference attack, anomaly detection, *etc.* To deal with such attacks, we outlined a simple defense based on packet padding.

Based on the knowledge that smart home devices are vulnerable to network attacks, we developed techniques to secure smart home systems that, in turn, also secure smart home devices. We present an approach for building secure smart home systems out of insecure components in Vigilia. Our approach moves the burden of securing the system from the device manufacturers to the platform, reducing concerns about the long-term availability of security patches.

We have implemented 4 applications in Vigilia using commercially available IoT devices. The intended deployment of the IoT devices used by each application had least one vulnerability. Vigilia successfully defended all our applications against all attacks.

In addition to smart home devices and systems, we also looked into smart home apps. In this dissertation, we also present a comprehensive study of interactions and conflicts between smart home apps, as well as an automated tool for finding conflicts. We studied the SmartThings framework: we collected and studied 198 official and 69 third-party apps. We formed pairs from the apps and we categorized their interaction patterns (*i.e.*, *interacts-with* relation) between apps into the following 3 major categories: (1) *device*, (2) *physical-medium*, and (3) *global-variable* relations.

Our results show conflicts in close to 60% of app pairs in the *device* relation category, more than 90% of app pairs in the *physical-medium* relation category, and around 11% of app pairs in the *global-variable* relation category.

Based on these results, we developed a tool, IoTCheck, that uses model checking to automatically detect and report conflicts. IoTCheck successfully detects around 96% of the conflicts in app pairs in the *device* and *physical-medium* relation categories.

6.2 Limitations and Future Directions

PingPong has its limitations and can be extended in several directions. First, in order to extract the signature of a new device, one must first acquire the device and apply PingPong to train and extract the corresponding packet-level signatures. This is actually realistic for an attacker with minimal side information, *i.e.*, one who knows what device they want to attack or who wants to distinguish between two different types of devices. One direction for future work is to extend PingPong by finding "similar" known behaviors for a new device, *e.g.*, via relaxed matching of known and unknown signatures.

Second, a signature may evolve over time, *e.g.*, when a software/firmware update changes a device's communication protocol. Whoever maintains the signature (*e.g.*, the attacker) needs to retrain and update the signature. We observed this phenomenon, for example, for the TP-Link plug. This can be handled by relaxed matching since the packet sequences tend to be mostly stable and only evolve by a few bytes (see Section 2.4.6).

Third, there may be inherent variability in some signatures due to configuration parameters (*e.g.*, credentials and device IDs) that are sent to the cloud and may lead to slightly different packet lengths. In Section 2.4.6, we saw that this variability is small: from a few to tens of bytes difference and only for some individual packets within a longer sequence. An attacker could train with different configuration parameters and apply relaxed matching when necessary (only on packets with length variations).

Other possible improvements include: profiling and subtracting background/periodic traffic during signature creation, and unifying the way we account for small variation in the signatures in the training and detection—PingPong currently supports range-based matching (see Section 2.3.2) and relaxed matching as separate features. Another limitation is that our methodology currently applies only to TCP—not to UDP-based devices that do not follow the request-reply pattern.

The techniques used in Vigilia, *i.e.*, static checking, router policy enforcement, process sandboxing, and capability-based RMI (see Section 3.6) along with Wi-Fi network filtering and Zigbee firewall (see Section 3.7) could also be deployed in existing systems, *e.g.*, Smart-

Things. The major issue with directly implementing our approach on SmartThings is that the SmartApps run on their cloud servers and none of the source code for their software infrastructure for running SmartApps is available. Nevertheless, assuming that we had access to their software infrastructure and extended it to execute applications and device drivers on the local network, it would be straightforward to deploy the static checking, router policy enforcement, process sandboxing, and capability-based RMI. The techniques to secure the Wi-Fi network against snooping, ARP-spoofing, and MAC-spoofing could be applied directly to the router, while the Zigbee firewall could also be integrated fairly easily into the smart hub.

The study of interactions between smart home apps and automated conflict detection using IoTCheck have a few alternatives for future directions. IoTCheck model-checks smart home apps to find *device* and *global-variable* conflicts based on the definition given in Definition 4.1. This definition works for most of the cases we have found in our manual study. However, we also found an exception of a pair of apps that do not actually conflict (see Section 4.4.1), but will be determined as a conflicting pair by IoTCheck. Thus, future work includes improving our definitions of conflict. Another future direction is to study and measure the criticality of conflicts between smart home apps—this is not a trivial problem because it depends on the specific deployment of the apps (see Section 4.1.3). Finally, conflict resolution is also a potential future direction. For example, our preliminary results show that a large number of the conflicts we detected can be resolved by redesigning APIs to (1) enable developers to understand and reason about potential conflicts during app development and (2) allow the platform to centrally control various app requests. Nevertheless, further study, experiment, implementation, and evaluation remain as future work.

Bibliography

[1] Forgiving security. https://github.com/imbrianj/forgiving_security/blob/master/forgiving_security.groovy, 2013.

[2] Auto humidity vent. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/jonathan-a/auto-humidity-vent.src/auto-humidity-vent.groovy, 2014.

[3] Close the valve. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/close-the-valve.src/close-the-valve.groovy, 2014.

[4] Hello, home phrase director. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/tslagle13/hello-home-phrase-director.src/hello-home-phrase-director.groovy, 2014.

[5] Lock it at a specific time. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/user8798/lock-it-at-a-specific-time.src/lock-it-at-a-specific-time.groovy, 2014.

[6] Sprayer controller 2. https://github.com/erocm123/SmartThingsPublic-1/blob/master/smartapps/sprayercontroller/sprayer-controller-2.src/sprayer-controller-2.groovy, 2014.

[7] Big turn off. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/big-turn-off.src/big-turn-off.groovy, 2015.

[8] Big turn on. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/big-turn-on.src/big-turn-on.groovy, 2015.

[9] Bon voyage. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/bon-voyage.src/bon-voyage.groovy, 2015.

[10] Good night. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/good-night.src/good-night.groovy, 2015.

[11] Greetings earthling. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/greetings-earthling.src/greetings-earthling.groovy, 2015.

[12] Keep me cozy. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/keep-me-cozy.src/keep-me-cozy.groovy, 2015.

[13] Light up the night. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/light-up-the-night.src/light-up-the-night.groovy, 2015.

[14] Lock it when i leave. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/lock-it-when-i-leave.src/lock-it-when-i-leave.groovy, 2015.

[15] Samsung smart fridge leaves Gmail logins open to attack. http://www.theregister.co.uk/2015/08/24/smart_fridge_security_fubar/, August 2015.

[16] Smart security. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/smartthings/smart-security.src/smart-security.groovy, 2015.

[17] Turn on at sunset. https://github.com/SmartThingsCommunity/Code/blob/master/smartapps/sunrise-sunset/turn-on-at-sunset.groovy, 2015.

[18] Initial state event streamer. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/master/smartapps/initialstate-events/initial-state-event-streamer.src/initial-state-event-streamer.groovy, 2016.

[19] Lawn watering tips - best times & schedules. http://www.scotts.com/smg/goART3/Howto/lawn-watering-tips/33800022/12400007/32000006/18800019, April 2016.

[20] Understanding illuminance: What's in a lux? https://www.allaboutcircuits.com/technical-articles/understanding-illuminance-whats-in-a-lux/, January 2016.

[21] Influxdb logger. https://github.com/codersaur/SmartThings/blob/master/smartapps/influxdb-logger/influxdb-logger.groovy, 2017.

[22] Thermostats. https://github.com/SmartThingsCommunity/SmartThingsPublic/blob/61b864535321a6f61cf5a77216f1e779bde68bd5/smartapps/smartthings/thermostats.src/thermostats.groovy, 2017.

[23] Ifttt. https://www.ifttt.com/, September 2018.

[24] Initial state. https://www.initialstate.com/, September 2018.

[25] Neato (connect). https://github.com/alyc100/SmartThingsPublic/blob/master/smartapps/alyc100/neato-connect.src/neato-connect.groovy, 2018.

[26] Smartthings groovy ide. https://graph.api.smartthings.com/, 2019.

[27] If motion detected by D-Link motion sensor, then turn on D-Link smart plug. https://ifttt.com/applets/393508p-if-motion-detected-by-d-link-motion-sensor-then-turn-on-d-link-smart-plug, January 2020.

[28] A. Acar, H. Fereidooni, T. Abera, A. K. Sikder, M. Miettinen, H. Aksu, M. Conti, A.-R. Sadeghi, and A. S. Uluagac. Peek-a-Boo: I see your smart home activities, even encrypted! *arXiv preprint arXiv:1808.02741*, 2018.

[29] Alexa. Top sites in United States. https://www.alexa.com/topsites/countries/US, November 2018.

[30] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security evaluation of home-based IoT deployments. In *Proceedings of the IEEE Symposium on Security and Privacy*, page 0. IEEE.

[31] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. SoK: Security evaluation of home-based IoT deployments. In *2019 2019 IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 208–226.

[32] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose. Yourthings scorecard. https://yourthings.info/, 2019.

[33] Amazon. https://www.amazon.com/smart-home/b/?ie=UTF8&node=6563140011&ref_=sv_hg_7, March 2019.

[34] B. Anderson and D. McGrew. Identifying encrypted malware traffic with contextual flow data. In *Proceedings of the 2016 ACM Workshop on Artificial Intelligence and Security*, AISec '16, pages 35–46, New York, NY, USA, 2016. ACM.

[35] Android.com. Android debug bridge (adb). https://developer.android.com/studio/command-line/adb, November 2018.

[36] M. Antonakakis, T. April, M. Bailey, M. Bernhard, E. Bursztein, J. Cochran, Z. Durumeric, J. A. Halderman, L. Invernizzi, M. Kallitsis, D. Kumar, C. Lever, Z. Ma, J. Mason, D. Menscher, C. Seaman, N. Sullivan, K. Thomas, and Y. Zhou. Understanding the mirai botnet. In *Proceedings of the 26th USENIX Security Symposium*, 2017.

[37] S. Apel, S. Kolesnikov, N. Siegmund, C. Kästner, and B. Garvin. Exploring feature interactions in the wild: The new feature-interaction challenge. In *Proceedings of the 5th International Workshop on Feature-Oriented Software Development (FOSD)*, pages 1–8, 2013.

[38] S. Apel, W. Scholz, C. Lengauer, and C. Kästner. Detecting dependences and interactions in feature-oriented design. In *IEEE 21st International Symposium on Software Reliability Engineering (ISSRE)*, pages 161–170, 2010.

[39] S. Apel, A. Von Rhein, T. ThüM, and C. KäStner. Feature-interaction detection based on feature-based specifications. *Computer Networks: The International Journal of Computer and Telecommunications Networking*, 57(12):2399–2409, August 2013.

[40] N. Apthorpe, D. Y. Huang, D. Reisman, A. Narayanan, and N. Feamster. Keeping the smart home private with smart(er) IoT traffic shaping. *Proceedings on Privacy Enhancing Technologies*, 2019(3), 2019.

[41] N. Apthorpe, D. Reisman, and N. Feamster. Closing the blinds: Four strategies for protecting smart home privacy from network observers. *CoRR*, abs/1705.06809, 2017.

[42] N. Apthorpe, D. Reisman, and N. Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted IoT traffic. *CoRR*, abs/1705.06805, 2017.

[43] N. Apthorpe, D. Reisman, S. Sundaresan, A. Narayanan, and N. Feamster. Spying on the smart home: Privacy attacks and defenses on encrypted IoT traffic. *CoRR*, abs/1708.05044, 2017.

[44] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie. PScout: Analyzing the android permission specification. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security, Raleigh, NC, USA, October 2012 (CCS '12)*, pages 217–228. ACM, 2012.

[45] H. Bagheri, A. Sadeghi, R. Jabbarvand, and S. Malek. Practical, formal synthesis and automatic enforcement of security policies for android. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 514–525. IEEE, 2016.

[46] D. Barrera, I. Molloy, and H. Huang. Idiot: Securing the internet of things like it's 1994. *CoRR*, abs/1712.03623, 2017.

[47] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Transactions on Computer Systems*, 22(4):381–420, November 2004.

[48] M. Bergin. Unplugging an IoT device from the cloud. `https://blog.korelogic.com/blog/2015/12/11/unplugging_iot_from_the_cloud`, December 2015.

[49] Z. Berkay Celik, E. Fernandes, E. Pauley, G. Tan, and P. McDaniel. Program analysis of commodity IoT applications for security and privacy: Challenges and opportunities. *arXiv preprint arXiv:1809.06962*, 2018.

[50] P. Bielik, V. Raychev, and M. Vechev. Scalable race detection for Android applications. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2015, pages 332–348, New York, NY, USA, 2015. ACM.

[51] G. D. Bissias, M. Liberatore, D. Jensen, and B. N. Levine. Privacy vulnerabilities in encrypted HTTP streams. In *Proceedings of the 5th International Conference on Privacy Enhancing Technologies*, PET'05, pages 1–11, Berlin, Heidelberg, 2006. Springer-Verlag.

[52] A. Bosu, F. Liu, D. D. Yao, and G. Wang. Collusive data leak and more: Large-scale threat analysis of inter-app communications. In *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*, pages 71–85. ACM, 2017.

[53] X. Cai, R. Nithyanand, and R. Johnson. CS-BuFLO: A congestion sensitive website fingerprinting defense. In *Proceedings of the 13th Workshop on Privacy in the Electronic Society*, pages 121–130. ACM, 2014.

[54] X. Cai, R. Nithyanand, T. Wang, R. Johnson, and I. Goldberg. A systematic approach to developing and evaluating website fingerprinting defenses. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 227–238. ACM, 2014.

[55] X. Cai, X. C. Zhang, B. Joshi, and R. Johnson. Touching from a distance: Website fingerprinting attacks and defenses. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 605–616, New York, NY, USA, 2012. ACM.

[56] M. Calder, M. Kolberg, E. H. Magill, and S. Reiff-Marganiec. Feature interaction: a critical review and considered forecast. *Computer Networks*, 41(1):115–141, 2003.

[57] N. T. Cam, P. Van Hau, and T. Nguyen. Android security analysis based on inter-application relationships. In *Information Science and Applications (ICISA) 2016*, pages 689–700. Springer, 2016.

[58] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. *SIGCOMM Comput. Commun. Rev.*, 37(4):1–12, Aug. 2007.

[59] M. Casado, T. Garfinkel, M. Freedman, A. Akella, D. Boneh, N. McKeowon, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *Proc. Usenix Security Symposium*, August 2006.

[60] Z. B. Celik, L. Babun, A. K. Sikder, H. Aksu, G. Tan, P. McDaniel, and A. S. Uluagac. Sensitive information tracking in commodity iot. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association.

[61] Z. B. Celik, P. McDaniel, and G. Tan. Soteria: Automated IoT safety and security analysis. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, 2018.

[62] A.-M. Chang, F. A. J. L. Scheer, and C. A. Czeisler. The human circadian system adapts to prior photic history. *The Journal of Physiology*, 589(5):1095–1102, March 2011.

[63] S. Chen, R. Wang, X. Wang, and K. Zhang. Side-channel leaks in web applications: A reality today, a challenge tomorrow. In *2010 IEEE Symposium on Security and Privacy*, pages 191–206. IEEE, 2010.

[64] G.-I. Cheng, M. Feng, C. E. Leiserson, K. H. Randall, and A. F. Stark. Detecting data races in Cilk programs that use locks. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 298–309, New York, NY, USA, 1998. ACM.

[65] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented programs. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 258–269, 2002.

[66] B. Copos, K. Levitt, M. Bishop, and J. Rowe. Is anybody home? Inferring activity from smart home network traffic. In *Security and Privacy Workshops (SPW), 2016 IEEE*, pages 245–251. IEEE, 2016.

[67] A. Dabrowski, J. Ullrich, and E. R. Weippl. Grid shock: Coordinated load-changing attacks on power grids: The non-smart power grid is vulnerable to cyber attacks as well. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 303–314, New York, NY, USA, 2017. ACM.

[68] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. Gunter, X. Zhou, and M. Grace. Guardian of the HAN: Thwarting mobile attacks on smart-home devices using OS-level situation awareness. https://arxiv.org/abs/1703.01537, 2017.

[69] S. Demetriou, N. Zhang, Y. Lee, X. Wang, C. Gunter, X. Zhou, and M. Grace. Guardian of the HAN: Thwarting mobile attacks on smart-home devices using OS-level situation awareness. https://arxiv.org/abs/1703.01537, 2017.

[70] T. Denning, T. Kohno, and H. M. Levy. Computer security and the modern home. *Commun. ACM*, 56(1):94–103, Jan. 2013.

[71] J. B. Dennis and E. C. Van Horn. Programming semantics for multiprogrammed computations. *Commun. ACM*, 9(3):143–155, Mar. 1966.

[72] W. Dietl, S. Dietzel, M. D. Ernst, K. Muşlu, and T. Schiller. Building and using pluggable type-checkers. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 681–690, 2011.

[73] C. Dixon, R. Mahajan, S. Agarwal, A. J. Brush, B. Lee, S. Saroiu, and P. Bahl. An operating system for the home. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, 2012.

[74] R. Doshi, N. Apthorpe, and N. Feamster. Machine learning DDoS detection for consumer internet of things devices. *CoRR*, abs/1804.04159, 2018.

[75] L. DROLEZ. Wanscam JW0004 IP Webcam hacking. `http://www.drolez.com/blog/?category=Hardware&post=jw0004-webcam`, July 2015.

[76] Y. Dvorkin and S. Garg. Iot-enabled distributed cyber-attacks on transmission and distribution grids. *CoRR*, abs/1706.07485, 2017.

[77] K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, i still see you: Why efficient traffic analysis countermeasures fail. In *2012 IEEE symposium on security and privacy*, pages 332–346. IEEE, 2012.

[78] M. Eddy, V. Song, and J. R. Delaney. Bitdefender box 2. `https://www.pcmag.com/review/357433/bitdefender-box-2`, November 2017.

[79] M. Eddy, V. Song, and J. R. Delaney. Symantec norton core router. `https://www.pcmag.com/review/355417/symantec-norton-core-router`, September 2017.

[80] D. Engler and K. Ashcraft. RacerX: Effective, static detection of race conditions and deadlocks. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 237–252, New York, NY, USA, 2003. ACM.

[81] H. P. Enterprise. Internet of things research study: 2015 report. `http://h20195.www2.hp.com/V2/GetDocument.aspx?docname=4AA5-4759ENW&cc=us&lc=en`, 2015.

[82] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[83] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proceedings of the 18th ACM Conference on Computer and Communications Security, Chicago, IL, USA, October 2011 (CCS '11).*

[84] A. P. Felt, S. Egelman, M. Finifter, D. Akhawe, and D. Wagner. How to ask for permission. In *Proceedings of the 7th USENIX Workshop on Hot Topics in Security, Bellevue, WA, August 2012 (HotSec '12).* USENIX, 2012.

[85] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 636–654. IEEE, 2016.

[86] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP), Oakland, CA, USA, May 2016 (Oakload '16)*, pages 636–654, May 2016.

[87] E. Fernandes, J. Paupore, A. Rahmati, D. Simionato, M. Conti, and A. Prakash. FlowFence: Practical data protection for emerging IoT application frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 531–548, Austin, TX, 2016. USENIX Association.

[88] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized Action Integrity for Trigger-Action Platforms. In *22nd Network and Distributed Security Symposium (NDSS 2018)*, Feb. 2018.

[89] D. Fisher. Pair of bugs open honeywell home controllers up to easy hacks. `https://threatpost.com/pair-of-bugs-open-honeywell-home-controllers-up-to-easy-hacks/`, 2015.

[90] C. Flanagan and S. N. Freund. Type-based race detection for java. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 219–232, 2000.

[91] C. Flanagan and S. N. Freund. Atomizer: A dynamic atomicity checker for multithreaded programs. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 256–267, New York, NY, USA, 2004. ACM.

[92] A. Forum. Mirai infection. `https://amcrest.com/forum/technical-discussion-f3/mirai-infection-t3686.html`, October 2017.

[93] B. Fouladi and S. Ghanoun. Honey, i'm home!!, hacking zwave home automation system. In *Black Hat USA*, 2013.

[94] O. N. Foundation. Software-defined networking (sdn) definition. `https://www.opennetworking.org/sdn-resources/sdn-definition`, 2017.

[95] T. A. S. Foundation. The apache groovy programming language. `http://groovy-lang.org/`, 2003-2018.

[96] M. Ghiglieri and E. Tews. A privacy protection system for HbbTV in Smart TVs. In *2014 IEEE 11th Consumer Communications and Networking Conference (CCNC)*, pages 357–362, Jan 2014.

[97] Google. Android things website. `https://developer.android.com/things/`, 2018.

[98] J. J. Gooley, K. Chamberlain, K. A. Smith, S. B. S. Khalsa, S. M. W. Rajaratnam, E. V. Reen, J. M. Zeitzer, C. A. Czeisler, and S. W. Lockley. Exposure to room light before bedtime suppresses melatonin onset and shortens melatonin duration in humans. *Journal of Clinical Endocrinology & Metabolism*, 96(3):E463–E472, March 2011.

[99] J. Hartin, P. M. Geisel, and C. L. Unruh. Lawn watering guide for california. Technical Report ANR 8044, University of California – Agriculture and Natural Resources, `http://anrcatalog.ucanr.edu/pdf/8044.pdf`, 2001.

[100] J. Hayes and G. Danezis. K-fingerprinting: A robust scalable website fingerprinting technique. In *Proceedings of the 25th USENIX Conference on Security Symposium*, SEC'16, pages 1187–1203, Berkeley, CA, USA, 2016. USENIX Association.

[101] Y. He, Q. Li, and K. Sun. Linkflow: Efficient large-scale inter-app privacy leakage detection. In *International Conference on Security and Privacy in Communication Systems*, pages 291–311. Springer, 2017.

[102] D. Herrmann, R. Wendolsky, and H. Federrath. Website fingerprinting: Attacking popular privacy enhancing technologies with the multinomial naïve-bayes classifier. In *Proceedings of the 2009 ACM workshop on Cloud computing security*, pages 31–42. ACM, 2009.

[103] A. Hesseldahl. A hackers-eye view of the internet of things. `http://recode.net/2015/04/07/a-hackers-eye-view-of-the-internet-of-things/`, 2015.

[104] K. Hill. The half-baked security of our 'Internet Of Things'. `http://www.forbes.com/sites/kashmirhill/2014/05/27/article-may-scare-you-away-from-internet-of-things/`.

[105] D. C. Holzman. What's in a color? the unique human health effects of blue light. *Environmental Health Perspectives*, 118(1):A22–A27, January 2010.

[106] G. J. Holzmann. *The SPIN model checker: Primer and reference manual*, volume 1003.

[107] J. Hong, A. Levy, L. Riliskis, and P. Levis. Don't talk unless i say so! securing the internet of things with default-off networking. In *2018 IEEE/ACM Third International Conference on Internet-of-Things Design and Implementation (IoTDI)*, pages 117–128, April 2018.

[108] C.-H. Hsiao, J. Yu, S. Narayanasamy, Z. Kong, C. L. Pereira, G. A. Pokam, P. M. Chen, and J. Flinn. Race detection for event-driven mobile applications. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 326–336, New York, NY, USA, 2014. ACM.

[109] Y. Hu and I. Neamtiu. Static detection of event-based races in Android apps. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 257–270, New York, NY, USA, 2018. ACM.

[110] A. Inc. Airplay. `https://developer.apple.com/airplay/`, 2018.

[111] IOActive. Belkin WeMo home automation vulnerabilities. `http://www.ioactive.com/pdfs/IOActive_Belkin-advisory-lite.pdf`, 2014.

[112] M. Jackson and P. Zave. Distributed feature composition: A virtual architecture for telecommunications services. *IEEE Transactions on Software Engineering*, 24(10):831–847, October 1998.

[113] Y. J. Jia, Q. A. Chen, S. Wang, A. Rahmati, E. Fernandes, Z. M. Mao, and A. Prakash. ContexIoT: Towards providing contextual integrity to appified IoT platforms. In *NDSS*, 2017.

[114] Y. Jin, E. Sharafuddin, and Z.-L. Zhang. Unveiling core network-wide communication patterns through application traffic activity graph decomposition. In *Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '09, pages 49–60, New York, NY, USA, 2009. ACM.

[115] Security of the Local LAN? `https://community.smartthings.com/t/security-on-the-local-lan/41585`, May 2018.

[116] T. Karagiannis, K. Papagiannaki, and M. Faloutsos. Blinc: Multilevel traffic classification in the dark. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 229–240, New York, NY, USA, 2005. ACM.

[117] C. Kreibich and J. Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *ACM SIGCOMM computer communication review*, 34(1):51–56, 2004.

[118] L. Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[119] Y. K. Lee, J. Y. Bang, G. Safi, A. Shahbazian, Y. Zhao, and N. Medvidovic. A sealant for inter-app security holes in android. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 312–323. IEEE, 2017.

[120] P. Leelaprute, T. Matsuo, T. Tsuchiya, and T. Kikuno. Detecting feature interactions in home appliance networks. In *Proceedings of the 2008 Ninth ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD)*, pages 895–903, 2008.

[121] A. Levy, J. Hong, L. Riliskis, P. Levis, and K. Winstein. Beetle: Flexible Communication for Bluetooth Low Energy. In *Proceedings of the 14th International Conference on Mobile Systems, Applications and Services (MobiSys)*, June 2016.

[122] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, and Y. Le Traon. Apkcombiner: Combining multiple android apps to support inter-app analysis. In *IFIP International Information Security and Privacy Conference*, pages 513–527. Springer, 2015.

[123] X. Li, L. Zhang, and X. Shen. IA-graph based inter-app conflicts detection in open IoT systems. In *Proceedings of the 20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 135–147, 2019.

[124] X. Li, L. Zhang, X. Shen, and Y. Qi. A systematic examination of inter-app conflicts detections in open IoT systems. Technical Report TR-2017-1, North Carolina State University, Dept. of Computer Science, 2017.

[125] C.-J. M. Liang, B. F. Karlsson, N. D. Lane, F. Zhao, J. Zhang, Z. Pan, Z. Li, and Y. Yu. Sift: building an internet of safe things. In *Proceedings of the 14th International Conference on Information Processing in Sensor Networks*, pages 298–309, 2015.

[126] M. Liberatore and B. N. Levine. Inferring the source of encrypted http connections. In *Proceedings of the 13th ACM conference on Computer and communications security*, pages 255–263. ACM, 2006.

[127] C. Lidbury and A. F. Donaldson. Dynamic race detection for C++11. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL 2017, pages 443–457, New York, NY, USA, 2017. ACM.

[128] LIFX. Device messages. `https://lan.developer.lifx.com/docs/device-messages`, 2018.

[129] J. Liu, M. D. George, K. Vikram, X. Qi, L. Waye, and A. C. Myers. Fabric: A platform for secure distributed computation and storage. In *Proceedings of the ACM 2009 Symposium on Operating Systems Principles and Implementation*, 2009.

[130] R. Liu, Z. Wang, L. Garcia, and M. Srivastava. RemedioT: Remedial actions for Internet-of-Things conflicts. In *Proceedings of the 6th ACM International Conference on Systems for Energy-Efficient Buildings, Cities, and Transportation*, BuildSys '19, pages 101–110, 2019.

[131] N. Lomas. Critical flaw identified in zigbee smart home devices. `http://techcrunch.com/2015/08/07/critical-flaw-ided-in-zigbee-smart-home-devices/`, 2015.

[132] M. Lopez-Martin, B. Carro, A. Sanchez-Esguevillas, and J. Lloret. Network traffic classifier with convolutional and recurrent neural networks for internet of things. *IEEE Access*, 5:18042–18050, 2017.

[133] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the Linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, 2001.

[134] L. Lu, E.-C. Chang, and M. C. Chan. Website fingerprinting and identification using ordered feature sequences. In *Proceedings of the 15th European Conference on Research in Computer Security*, ESORICS'10, pages 199–214, Berlin, Heidelberg, 2010. Springer-Verlag.

[135] S. Lu, S. Park, and Y. Zhou. Finding atomicity-violation bugs through unserializable interleaving testing. *IEEE Transactions on Software Engineering*, 38(4):844–860, 2012.

[136] T. Luo and W. Du. *Contego: Capability-Based Access Control for Web Browsers*, pages 231–238. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[137] S. Maffeis, J. C. Mitchell, and A. Taly. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*, pages 125–140, 2010.

[138] P. Maiya, A. Kanade, and R. Majumdar. Race detection for Android applications. *SIGPLAN Not.*, 49(6):316–325, June 2014.

[139] A. Mettler, D. Wagner, and T. Close. Joe-E: A security-oriented subset of Java. In *Network and Distributed Systems Symposium.* Internet Society, 2010.

[140] M. Miettinen, S. Marchal, I. Hafeez, T. Frassetto, N. Asokan, A. R. Sadeghi, and S. Tarkoma. IoT Sentinel: Automated device-type identification for security enforcement in IoT. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[141] M. S. Miller, K.-P. Yee, and J. Shapiro. Capability myths demolished, 2003.

[142] A. Mohsenian-Rad and A. Leon-Garcia. Distributed internet-based load altering attacks against smart power grids. *IEEE Transactions on Smart Grid*, 2(4):667–674, Dec 2011.

[143] S. Munir and J. A. Stankovic. Depsys: Dependency aware integration of cyber-physical systems for smart homes. In *ICCPS '14: ACM/IEEE 5th International Conference on Cyber-Physical Systems (with CPS Week 2014)*, ICCPS '14, pages 127–138, Washington, DC, USA, 2014. IEEE Computer Society.

[144] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1999.

[145] B. Nadel. Norton core router review. `https://www.tomsguide.com/us/norton-core-router,review-4827.html`, November 2017.

[146] B. Nadel. Bitdefender box (2018) review: Flexible protection. `https://www.tomsguide.com/us/bitdefender-box,review-3766.html`, January 2018.

[147] C. Nandi and M. D. Ernst. Automatic trigger generation for rule-based smart homes. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security (PLAS)*, pages 97–102, 2016.

[148] J. L. Newcomb, S. Chandra, J.-B. Jeannin, C. Schlesinger, and M. Sridharan. Iota: a calculus for internet of things automation. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 119–133, 2017.

[149] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. M. Colbert, and P. McDaniel. IotSan: Fortifying the safety of systems. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, pages 191–203, New York, NY, USA, 2018. ACM.

[150] T. T. Nguyen and G. Armitage. A survey of techniques for internet traffic classification using machine learning. *Commun. Surveys Tuts.*, 10(4):56–76, Oct. 2008.

[151] Oauth integrations. `https://smartthings.developer.samsung.com/docs/oauth/oauth-integration.html`.

[152] T. OConnor, R. Mohamed, M. Miettinen, W. Enck, B. Reaves, and A.-R. Sadeghi. HomeSnitch: Behavior transparency and control for smart home IoT devices. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, pages 128–138, New York, NY, USA, 2019. ACM.

[153] T. Oluwafemi, T. Kohno, S. Gupta, and S. Patel. Experimental security analyses of non-networked compact fluorescent lamps: A case study of home automation security. In *Proceedings of the LASER 2013, Arlington, VA, USA (LASER 2013)*, pages 13–24. USENIX, 2013.

[154] openHAB. openhab website. `https://www.openhab.org/`, 2018.

[155] OpenWrt. `https://openwrt.org`.

[156] S. Oster, M. Zink, M. Lochau, and M. Grechanik. Pairwise feature-interaction testing for SPLs: Potentials and limitations. In *Proceedings of the 15th International Software Product Line Conference (SPLC)*, pages 6:1–6:8, 2011.

[157] A. Panchenko and F. Lanze. Website fingerprinting at internet scale. In *NDSS*, 2016.

[158] A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the 10th annual ACM workshop on Privacy in the electronic society*, pages 103–114. ACM, 2011.

[159] R. Perdisci, W. Lee, and N. Feamster. Behavioral clustering of http-based malware and signature generation using malicious network traces. In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*, NSDI'10, pages 26–26, Berkeley, CA, USA, 2010. USENIX Association.

[160] A. J. Pinheiro, J. M. Bezerra, and D. R. Campelo. Packet padding for improving privacy in consumer IoT. In *2018 IEEE Symposium on Computers and Communications (ISCC)*, pages 00925–00929, June 2018.

[161] Proxy arp. `http://www.cisco.com/c/en/us/support/docs/ip/dynamic-address-allocation-resolution/13718-5.html`, January 2008.

[162] Y. Racine. Fireco2alarm smartapp. `https://github.com/yracine/device-type.myecobee/blob/master/smartapps/FireCO2Alarm.src/FireCO2Alarm.groovy`, 2014.

[163] A. Rajan, L. du Bousquet, Y. Ledru, G. Vega, and J.-L. Richier. Assertion-based test oracles for home automation systems. In *Proceedings of the 7th International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPRES)*, pages 45–52, 2010.

[164] Rapid. HACKING IoT: A case study on baby monitor exposures and vulnerabilities. `https://www.rapid7.com/docs/Hacking-IoT-A-Case-Study-on-Baby-Monitor-Exposures-and-Vulnerabilities.pdf`, September 2015.

[165] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[166] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi. Information exposure from consumer IoT devices: A multidimensional, network-informed measurement approach. In *Proceedings of the Internet Measurement Conference*, pages 267–279, 2019.

[167] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, RFC Editor, August 2018.

[168] E. Rescorla and T. Dierks. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246, Aug. 2008.

[169] F. Roesner and T. Kohno. Securing embedded user interfaces: Android and beyond. In *Proceedings of the 22nd USENIX Security Symposium, Washington, D.C. (USENIX Security '13)*, pages 97–112. USENIX, 2013.

[170] F. Roesner, T. Kohno, A. Moshchuk, B. Parno, H. J. Wang, and C. Cowan. User-driven access control: Rethinking permission granting in modern operating systems. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy, Oakland, CA, USA, May 2012 (Oakland '12)*, pages 224–238, 2012.

[171] C. Sader. Auto lock door smartapp. `https://github.com/smartthings-users/smartapp.auto-lock-door/blob/master/auto-lock-door.smartapp.groovy`, 2013.

[172] S. Saghafi, K. Fisler, and S. Krishnamurthi. Features and object capabilities: Reconciling two visions of modularity. In *Proceedings of the 11th Annual International Conference on Aspect-oriented Software Development*, AOSD '12, pages 25–34, New York, NY, USA, 2012. ACM.

[173] Samsung SmartCam. `https://www.exploitee.rs/index.php/Samsung_SmartCam%E2%80%8B#Fixing_Password_Reset_.22Pre-Auth.22`, August 2014.

[174] T. S. Saponas, J. Lester, C. Hartung, S. Agarwal, and T. Kohno. Devices that tell on you: Privacy trends in consumer ubiquitous computing. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 5:1–5:16, Berkeley, CA, USA, 2007. USENIX Association.

[175] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems (TOCS)*, 15(4):391–411, 1997.

[176] J. Schwenn. How to protect wordpress from xml-rpc attacks on ubuntu 14.04. https://www.digitalocean.com/community/tutorials/how-to-protect-wordpress-from-xml-rpc-attacks-on-ubuntu-14-04, February 2016.

[177] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani. Toward generating a new intrusion detection dataset and intrusion traffic characterization. 2018.

[178] S. Shekyan and A. Harutyunyan. To watch or to be watched: Turning your surveillance camera against you. https://conference.hitb.org/hitbsecconf2013ams/materials/D2T1%20-%20Sergey%20Shekyan%20and%20Artem%20Harutyunyan%20-%20Turning%20Your%20Surveillance%20Camera%20Against%20You.pdf.

[179] A. Sivanathan, H. H. Gharakheili, A. R. Franco Loi, C. Wijenayake, A. Vishwanath, and V. Sivaraman. Classifying IoT devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing*, (01):1–1.

[180] A. Sivanathan, D. Sherratt, H. H. Gharakheili, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman. Characterizing and classifying IoT traffic in smart cities and campuses. In *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 559–564, May 2017.

[181] SmartThings. Smartthings public github repo. https://github.com/SmartThingsCommunity/SmartThingsPublic, 2018.

[182] S. SmartThings. Samsung smartthings website. http://www.smartthings.com, 2018.

[183] S. Soltan, P. Mittal, and H. V. Poor. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 15–32, Baltimore, MD, 2018. USENIX Association.

[184] D. A. Sorensen, N. Vanggaard, and J. M. Pedersen. IoTsec: Automatic profile-based firewall for IoT devices. http://projekter.aau.dk/projekter/files/260081086/report_print_friendly.pdf, June 2017.

[185] Spruce - the smart irrigation controller. http://www.spruceirrigation.com, April 2016.

[186] Square, Inc. What's going to happen with IFTTT? https://square.github.io/okhttp/, 2019.

[187] Stacey Higginbotham. OkHttp. https://staceyoniot.com/whats-going-to-happen-with-ifttt/, 2019.

[188] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Guo, and P. Tague. Smartauth: User-centered authorization for the internet of things. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, pages 361–378, Berkeley, CA, USA, 2017. USENIX Association.

[189] Tomoyo linux. https://tomoyo.osdn.jp/index.html.en, April 2017.

[190] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, and G. H. Xu. Iotcheck. `http://plrg.ics.uci.edu/iotcheck/`, 2020.

[191] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, and G. H. Xu. Iotcheck and manual study supporting materials. `http://plrg.ics.uci.edu/iotcheck/`, 2020.

[192] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, and G. H. Xu. Iotcheck vagrant package. `http://plrg.ics.uci.edu/iotcheck/`, 2020.

[193] R. Trimananda, S. A. H. Aqajari, J. Chuang, B. Demsky, G. H. Xu, and S. Lu. Understanding and automatically detecting conflicting interactions between smart home IoT applications. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, November 2020.

[194] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky. Pingpong: Packet-level signatures for smart home devices (software and dataset). `http://plrg.ics.uci.edu/pingpong/`.

[195] R. Trimananda, J. Varmarken, A. Markopoulou, and B. Demsky. Packet-Level Signatures for Smart Home Devices. *Proceedings of the 2020 Network and Distributed System Security (NDSS) Symposium*, February 2020.

[196] R. Trimananda, A. Younis, B. Wang, B. Xu, B. Demsky, and G. Xu. Vigilia: Securing smart home edge computing (software). `http://plrg.ics.uci.edu/vigilia/`.

[197] R. Trimananda, A. Younis, B. Wang, B. Xu, B. Demsky, and G. Xu. Vigilia: Securing smart home edge computing. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 74–89. IEEE, 2018.

[198] Y. Tsutano, S. Bachala, W. Srisa-An, G. Rothermel, and J. Dinh. An efficient, robust, and scalable approach for analyzing interacting android apps. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 324–334. IEEE, 2017.

[199] UNSW. List of smart home devices. `https://iotanalytics.unsw.edu.au/resources/List_Of_Devices.txt`, November 2018.

[200] B. Ur, J. Jung, and S. Schechter. The current state of access control for smart devices in homes. In *Proceedings of Workshop on Home Usable Privacy and Security, Newcastle, UK, July 2013 (HUPS)*, 2013.

[201] S. H. USA. What is a smart home? `https://www.smarthomeusa.com/smarthome/`, 2018.

[202] Veracode. The internet of things: Security research study. `https://www.veracode.com/sites/default/files/Resources/Whitepapers/internet-of-things-whitepaper.pdf`, 2015.

[203] P. A. Vicaire, E. Hoque, Z. Xie, and J. A. Stankovic. Bundle: A group-based programming abstraction for cyber-physical systems. *IEEE Transactions on Industrial Informatics*, 8(2):379–392, 2012.

[204] P. A. Vicaire, Z. Xie, E. Hoque, and J. A. Stankovic. Physicalnet: A generic framework for managing and programming across pervasive computing networks. In *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE*, pages 269–278. IEEE, 2010.

[205] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. 10:203–232, April 2003.

[206] T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *23rd {USENIX} Security Symposium ({USENIX} Security 14)*, pages 143–157, 2014.

[207] G. Wassermann. ZyXEL NBG-418N, PMG5318-B20A and P-660HW-T1 routers contain multiple vulnerabilities. `http://www.kb.cert.org/vuls/id/870744`, October 2015.

[208] A. Whitaker and D. Newman. Penetration testing and network defense: Performing host reconnaissance. `http://www.ciscopress.com/articles/article.asp?p=469623&seqNum=3`, June 2018.

[209] Z. Whittaker. Hackers exploiting 'serious' flaw in Netgear routers. `http://www.zdnet.com/article/hackers-exploiting-serious-flaw-in-netgear-routers/`, October 2015.

[210] M. Wilson, M. Kolberg, and E. H. Magill. Considering side effects in service interactions in home automation-an online approach. *Feature Interactions in Software and Communication Systems IX*, pages 172–187, 2008.

[211] Wink hub. `https://www.exploitee.rs/index.php/Wink_Hub%E2%80%8B%E2%80%8B#Wink_Hub_.22.2Fvar.2Fwww.2Fdev_detail.php.22_SQLi_for_root_command_execution`.

[212] A. D. Wood, J. A. Stankovic, G. Virone, L. Selavo, Z. He, Q. Cao, T. Doan, Y. Wu, L. Fang, and R. Stoleru. Context-aware wireless sensor networks for assisted living and residential monitoring. *IEEE network*, 22(4), 2008.

[213] C. V. Wright, L. Ballard, S. E. Coull, F. Monrose, and G. M. Masson. Uncovering spoken phrases in encrypted voice over IP conversations. *ACM Transactions on Information and System Security*, 13(4):35:1–35:30, Dec. 2010.

[214] C. V. Wright, L. Ballard, F. Monrose, and G. M. Masson. Language identification of encrypted VoIP traffic: Alejandra y Roberto or Alice and Bob? In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, SS'07, pages 4:1–4:12, Berkeley, CA, USA, 2007. USENIX Association.

[215] M. Yagita, F. Ishikawa, and S. Honiden. An application conflict detection and resolution system for smart homes. In *Proceedings of the First International Workshop on Software Engineering for Smart Cyber-Physical Systems*, SEsCPS '15, pages 33–39, Piscataway, NJ, USA, 2015. IEEE Press.

[216] S. Yarosh and P. Zave. Locked or not?: Mental models of IoT feature interaction. In *Proceedings of the 2017 Conference on Human Factors in Computing Systems (CHI)*, pages 2993–2997, 2017.

[217] A. Yoga and S. Nagarakatte. Atomicity violation checker for task parallel programs. In *Proceedings of the 2016 International Symposium on Code Generation and Optimization*, CGO '16, pages 239–249, New York, NY, USA, 2016. ACM.

[218] K. York. Dyn statement on 10/21/2016 ddos attack. `http://dyn.com/blog/dyn-statement-on-10212016-ddos-attack/`, October 2016.

[219] T. Yu, V. Sekar, S. Seshan, Y. Agarwal, and C. Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the Internet-of-Things. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks (HotNets)*, 2015.

[220] N. Zeldovich, S. Boyd-Wickizer, E. Kohler, and D. Mazières. Making information flow explicit in HiStar. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, 2006.

[221] N. Zeldovich, S. Boyd-Wickizer, and D. Mazières. Securing distributed systems with information flow control. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, 2008.

[222] W. Zhang, Y. Meng, Y. Liu, X. Zhang, Y. Zhang, and H. Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, pages 1074–1088, New York, NY, USA, 2018. ACM.