# "Building private-by-design IoT systems"

Zavalyshyn, Igor

## ABSTRACT

With the rapid adoption of Internet of Things (IoT) technologies and a growing amount and variety of sensitive data collected by various IoT systems, the mechanisms commonly used to ensure individual privacy and security are still insufficient. Numerous security breaches and sensitive data leaks have become a commonplace. This is mainly due to the fact that traditional security mechanisms can only restrict access to a given IoT data source, but not what can be done with that data after the access has been granted. In this thesis, we reimagine the concept of IoT systems design which aims to give users full control of sensor data generated by their devices, and to provide mechanisms for users to specify and enforce their privacy and security preferences regarding sensor data collection, processing and sharing. To achieve these goals, we propose several novel systems that collectively span across several domains: local, cloud and mobile. For the local domain, we present HomePad, a privacy-aware smart hub for home environment which allows users to determine how various IoT applications (apps) access and process sensitive data collected by smart devices, and to block those apps that violate the privacy preferences specified by the users. To this end, HomePad introduces two key design concepts: (1) a novel dataflow programming model which makes sensitive data flows within apps explicit, and (2) an element-based app structure which allows to model any smart home app as a directed graph and automatically verify its data flows against user-defined privacy policies using Prolog pre...

## CITE THIS VERSION

Zavalyshyn, Igor. *Building private-by-design IoT systems.* Prom. : Sadre, Ramin http://hdl.handle.net/2078.1/254399

UNIVERSIDADE DE LISBOA
INSTITUTO SUPERIOR TÉCNICO

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
(UCLouvain)

UNDER ERASMUS MUNDUS PROGRAMME

# Building Private-by-Design IoT Systems

**Igor Zavalyshyn**

**Supervisor**: Doctor Nuno Miguel Carvalho dos Santos
**Co-Supervisor**: Doctor Ramin Sadre

**Thesis specifically prepared to obtain the PhD Degree in Information Systems and Computer Engineering**

**Jury**

Doctor Ramin Sadre, UCLouvain, Belgium

Doctor Charles Pecheur, UCLouvain, Belgium

Doctor Axel Legay, UCLouvain, Belgium

Doctor Hamed Haddadi, Imperial College London, UK

Doctor Henrique Domingos, NOVA University, Portugal

**2021**

To my family.

# Abstract

With the rapid adoption of Internet of Things (IoT) technologies and a growing amount and variety of sensitive data collected by various IoT systems, the mechanisms commonly used to ensure individual privacy and security are still insufficient. Numerous security breaches and sensitive data leaks have become a commonplace. This is mainly due to the fact that traditional security mechanisms can only restrict access to a given IoT data source, but not what can be done with that data after the access has been granted. In this thesis, we reimagine the concept of IoT systems design which aims to give users full control of sensor data generated by their devices, and to provide mechanisms for users to specify and enforce their privacy and security preferences regarding sensor data collection, processing and sharing.

To achieve these goals, we propose several novel systems that collectively span across several domains: local, cloud and mobile. For the local domain, we present HomePad, a privacy-aware smart hub for home environment which allows users to determine how various IoT applications (*apps*) access and process sensitive data collected by smart devices, and to block those apps that violate the privacy preferences specified by the users. To this end, HomePad introduces two key design concepts: (1) a novel dataflow programming model which makes sensitive data flows within apps explicit, and (2) an element-based app structure which allows to model any smart home app as a directed graph and automatically verify its data flows against user-defined privacy policies using Prolog predicates. For the cloud domain, we propose PatrIoT, a private-by-design IoT platform that extends HomePad's dataflow programming model to the cloud. It leverages Intel SGX to prevent unauthorized access to the sensor data by untrusted cloud providers, and offers homeowners an intuitive security abstraction named *flowwall* which allows them to specify easy-to-use policies for controlling sensitive sensor data flows within the apps they install. Finally, for the mobile domain, we propose Flowverine, a system for building privacy-aware mobile apps handling sensitive IoT data on unmodified Android platforms. Flowverine adapts dataflow programming model to a much more complex Android programming and runtime environment, and uses aspect-oriented programming (AOP) for dynamic taint analysis.

Complementary to these three systems, this thesis also proposes additional techniques for enhancing the security, fault tolerance and reliability of IoT systems based on N-version programming and software hardening.

# Résumé

Avec l'adoption rapide des technologies de l'Internet des objets (IoT) ainsi qu'une quantité et une variété croissantes de données sensibles collectées par divers systèmes IoT, les mécanismes utilisés pour garantir la confidentialité et la sécurité individuelle sont encore insuffisants. De nombreuses failles de sécurité et fuites de données sensibles sont devenues monnaie courante. Cela est principalement dû au fait que les mécanismes de sécurité traditionnels ne peuvent restreindre l'accès qu'à une source de données IoT spécifique, mais pas à ce qui peut être fait avec ces données une fois que l'accès a été accordé. Dans cette thèse, nous réinventons le principe de conception de systèmes IoT en visant à donner aux utilisateurs un contrôle total sur les données générées par les capteurs de leurs appareils, et à fournir des mécanismes permettant aux utilisateurs de spécifier et d'imposer leurs préférences de confidentialité et de sécurité concernant la collecte, le traitement et le partage des données de capteurs.

Pour atteindre ces objectifs, nous proposons plusieurs systèmes novateurs qui s'étendent collectivement sur plusieurs domaines : local, cloud et mobile. Pour le domaine local, nous proposons HomePad, un hub intelligent, soucieux de la confidentialité, destiné à l'environnement domestique, qui permet aux utilisateurs de déterminer comment les diverses applications IoT accèdent et traitent les données sensibles collectées par les appareils intelligents, et de bloquer les applications qui violent les préférences de confidentialité spécifiées par les utilisateurs. à cette fin, HomePad introduit deux principes de conception clés : (1) un nouveau modèle de programmation de flux de données qui rend explicite les flux de données sensibles au sein des applications, et (2) une structure d'application basée sur des éléments qui permet de modéliser toute application de maison intelligente sous forme d'un graphe dirigé, et vérifie automatiquement ses flux de données par rapport aux politiques de confidentialité définies par l'utilisateur à l'aide de prédicats Prolog. Pour le domaine du cloud, nous proposons PatrIoT, une plateforme IoT privée par conception qui étend au cloud le modèle de programmation de flux de données de HomePad. PatrIoT exploite Intel SGX pour empêcher l'accès non autorisé aux données des capteurs par des fournisseurs de cloud non approuvés et offre aux propriétaires une abstraction de sécurité intuitive appelée flowwall leur permettant de spécifier des politiques faciles à utiliser afin de contrôler les flux de données sensibles des capteurs au sein des applications qu'ils installent. Enfin, pour le domaine mobile, nous proposons Flowverine, un système de création d'applications mobiles respectueuses de la vie privée traitant des données IoT sensibles sur des plates-formes Android non modifiées. Flowverine adapte le modèle de programmation de flux de données à un environnement de programmation et d'exécution Android beaucoup plus complexe, et utilise la programmation orientée aspect (AOP) pour l'analyse dynamique de marquage.

En complément de ces trois systèmes, cette thèse propose également des techniques complémentaires pour améliorer la sécurité, la tolérance aux pannes et la fiabilité des systèmes IoT basés sur la programmation multiversion (NVP) et le durcissement logiciel.

# Resumo

Com a rápida adoção de tecnologias de Internet das Coisas (IoT) acompanhado por uma crescente quantidade e variedade de dados confidenciais recolhidos por vários sistemas de IoT, verifica-se uma clara insuficiência nos mecanismos tipicamente utilizados para garantir a privacidade e segurança individuais dos utilizadores. De facto, inúmeras violações de segurança e roubo de dados confidenciais tornaram-se um lugar-comum. Isto deve-se principalmente ao facto de que os mecanismos de segurança tradicionais apenas restringem o acesso às fontes de dados de IoT, mas não ao que pode ser feito com esses dados após o acesso ter sido concedido. Nesta tese, nós repensamos a arquitectura de sistemas IoT procurando dar aos utilizadores o controlo total dos dados gerados pelos sensores dos seus dispositivos e fornecer mecanismos para que os utilizadores especifiquem as suas preferências de privacidade e segurança em relação à recolha, processamento e partilha de dados dos sensores.

Para atingir esses objetivos, propomos novos sistemas que abarcam vários domínios: local, nuvem e móvel. Para o domínio local, apresentamos o HomePad, um dispositivo inteligente com mecanismos de protecção de privacidade para ambiente doméstico que permite aos utilizadores determinar como várias aplicações de IoT acedem e processam dados confidenciais e bloquear as aplicações que violam as preferências de privacidade especificadas pelos utilizadores. Para este fim, o HomePad apresenta dois conceitos-chave: (1) um novo modelo de programação de fluxo de dados que torna explícito a propagação de dados sensíveis dentro das aplicações (2) uma estrutura aplicacional baseada em elementos que permite modelar as aplicações domésticas inteligentes sob a forma de um gráfico direcionado e verificar automaticamente os seus fluxos de dados em relação às políticas de privacidade definidas pelo utilizador. Para o domínio da nuvem, propomos o PatrIoT, uma plataforma IoT *privacy-by-design* que estende o modelo de programação de fluxo de dados do HomePad para a nuvem. Este sistema tira partido de Intel SGX para evitar o acesso não autorizado aos dados de sensores por provedores de nuvem não confiáveis e oferece aos utilizadores uma abstração de segurança intuitiva chamada *flowwall*, que permite especificar políticas fáceis de usar para controlar fluxos de dados sensíveis geradas pelas aplicações. Finalmente, para o domínio móvel, propomos o Flowverine, um sistema para a construção de aplicações móveis com reconhecimento de privacidade que manipulam dados IoT confidenciais em plataformas Android não modificadas. O Flowverine adapta o modelo de programação de fluxo de dados a uma programação Android, a qual é muito mais complexa, e usa programação orientada a aspectos (AOP) para análise dinâmica de propagação de informação.

Complementarmente a esses três sistemas, esta tese também propõe técnicas adicionais para aumentar a segurança, tolerância a falhas e confiabilidade de sistemas IoT com base num modelo de programação baseada em N versões e no reforço da segurança do software.

# Keywords

## Keywords

Internet of Things (IoT)
Data privacy
Dataflow programming model
Privacy policy
Private-by-design systems

## Mots clés

Internet des Objets (IoT)
Confidentialité des données
Modèle de programmation de flux de données
Politique de confidentialité
Systèmes privés par conception

## Palavras-chave

Internet das Coisas (IoT)
Dados privados
Modelo de programação de fluxo de dados
Política de Privacidade
Sistemas privados por design

# Acknowledgments

This document is a product of hard work, sleepless nights and many cups of coffee. All the ideas, insights and results – nothing of these would be possible without the people, organizations and places I want to mention here. Do note, however, that if some of the names are missing, this is mainly due to my poor memory and does not in any way reflect a lack of my gratitude or appreciation.

First of all, I would like to thank my advisors. I was lucky enough to have several of those. Nuno Santos played an essential role in my PhD and guided me throughout these years. The amount of effort, patience and energy he put in this process is fascinating. From the bottom of my heart I thank him for showing me the right direction and leading me there, for teaching me everything I know about research, and for always pushing me to strive for the best. I also would like to thank Ramin Sadre for his continuous help and all the fruitful and incredibly friendly discussions we had. His precious feedback and insights helped to shape my research, and made me much more comfortable in presenting my findings. Next, I would like to thank Peter Van Roy for always being supportive and encouraging, and showing a genuine interest in my work. I thank him for all the hard work he did as my PhD program's coordinator, for inviting me to collaborate in a LightKone EU project, and for helping me with my research. Finally, I would like to thank Axel Legay for his continuous support which was crucial in the last years of my PhD. Without his help this thesis would be incomplete and hardly even possible.

To all my EMJD-DC, INESC-ID, and UCLouvain colleagues and friends, especially to Richard Gil Martínez, Shady Issa, Diogo Barradas, Nuno Duarte, Daniel Porto, David Gureya, Khulan Batbayar, Sana Imtiaz, Muhammad Bilal, Lionel Metongnon, Raziel Carvajal Gómez, Olivier Goletti and Fabien Duchêne, for all the joyful and insightful conversations, and for the great times we have spent together.

A special mention to Manuel Bravo Gestoso, Zhongmiao Li, Emmanouil Dimogerontakis, João Neto, Paolo Laffranchini and Illia Sheremet. These finest gentlemen truly became part of my extended family and made this journey full of wonderful memories that I will always cherish. Without their positive energy, smiles and support, these years would have been so much tougher.

To Portugal and Belgium, and in particular to the beautiful cities of Lisbon and Brussels, for hosting me and for letting me explore their local cuisine, culture and natural wonders. From warm Atlantic ocean's waves and freshly baked bacalhau, to picturesque canals and fries – all of these will remain in my memory forever.

To all the people who took care of all the administrative matters at each of the universities. Special thanks to Vanessa Maons, Sophie Renard and Paula Barrancos.

Finally and most importantly, I would like to thank my family for supporting me throughout this journey. To my parents for raising me and teaching me everything I know, to my sisters for always cheering me up, to my wife for her patience, love and support.

This work was supported in part by the Erasmus Mundus Joint Doctorate in Distributed Computing (EMJD-DC) funded by the Education, Audiovisual and Culture Executive Agency (EACEA) of the European Commission; the national funds through Instituto Superior Técnico, Universidade de Lisboa, and Fundação para a Ciência e Tecnologia (FCT) via the UID/CEC/50021/2013, SFRH/BSAB/135236/2017 and UIDB/50021/2020 projects; LightKone project in the European Union Horizon 2020 Framework Program under grant agreement 732505; and a CISCO research grant.

Brussels, August 30, 2021

*Igor Zavalyshyn*

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

## 1.1 IoT privacy and security issues

We live in a world in which Internet-connected devices, such as mobile phones, fitness trackers and smart speakers – all united under the umbrella of Internet of Things (IoT) – became part of our daily lives. Based on various reports, there are from 7 to 20 billion of such devices in use worldwide, and their number keeps on growing exponentially [74, 205, 31].

However, the growing popularity of IoT devices raises concerns over the sensitive data these devices collect and make freely available to their respectful manufacturers and IoT service providers. For instance, smart phones collect sensor data from their built-in sensors and other connected devices (e.g. fitness trackers) for various installed third-party applications, *apps*, to use. Location, health and biometric information, such as physical activity, sleep cycles and even heart rate and blood pressure can then be harvested and stored at the remote cloud servers without the user authorization. Similarly, smart speakers use always-on microphones to continuously listen and react to user voice commands, but at the same time they can be abused to provide a stealthy channel into private user conversations. Collectively, personal IoT devices have an unprecedented access to their users' most personal and extremely sensitive information, which can be misused in harmful ways.

Unfortunately, these concerns can be grounded on actual security exploits [73, 170, 132, 211], and on studies that reveal numerous vulnerabilities in commodity IoT technologies [96, 139, 236]. From video feeds streamed by smart cameras to audio recordings captured by smart speakers, highly sensitive data can be extracted, processed, and shared without end-users' awareness or permission. Anecdotal evidence confirms the existence of data abuse, e.g., for targeted advertisement [92], legal forensic purposes [122], corporate and government surveillance [162], eavesdropping or peeping [7, 2, 73], or of simple data mishandling [8, 73, 125]. Third-party IoT apps can further frustrate end-users' expectations by misusing sensitive data [96].

Although many security breaches in IoT devices and applications are caused by traditional software vulnerabilities, said concerns about improper data usage are structural. IoT platforms like Samsung's SmartThings or Amazon's Alexa act as sinks for all sensitive data collected from smart devices. End-users are forced to give up control of their data on behalf of IoT platform providers and third-party application developers, whose goal is to monetize it. To secure providers' revenues, internal systems tend to be opaque in terms of their implementation, operation, and user data handling. End-users are left in the dark as to the kind, amount, and purpose of data that is collected. As a result, this "data rush" has created a fragmented IoT market that operates under a privacy-aggressive paradigm which, ironically, has caused, according to some voices, a slowdown in IoT industry growth [104, 101].

The aforementioned "data rush" has led to numerous cases of unauthorized sensor data collection and sharing without user awareness. We refer to such sensitive data disclosure as a *data leak*. The causes and consequences of various data leaks are of primary interest for us, and this relationship has greatly motivated our research. However, figuring out why a certain data leak occurred can be complicated, as there can be a range of different causes. One way of understanding causes and consequences is through categorizing them. In the next section, we highlight the reasons and the impact of data leaks that occur in various IoT domains: within third-party applications (both for mobile platforms, such as Android, and smart home platforms, such as SmartThings), within the smart home platforms themselves, and, finally, within the IoT devices that may be subject to various security and privacy attacks, for instance, by interfering with the device software or communication channels.

## 1.2 Reasons behind IoT data leaks

There are various reasons behind sensitive data leaks and subsequent violations of users' privacy in existing IoT systems. We will now highlight the main reasons behind data leaks considered in this thesis.

On an application level, we differentiate between data leaks caused by IoT apps used on mobile devices (e.g. Android-based smartphones) and those apps that come as part of an IoT platform (e.g. Samsung SmartThings or Amazon Echo). In case of mobile apps, data leaks may occur when an app developer intentionally collects and sends out sensitive sensor data without the user consent. Unfortunately, data leaks may also occur accidentally due to the existence of security vulnerabilities in mobile apps' code, or by the inclusion of third-party libraries [40, 235] (e.g., ad libs) designed to harvest sensitive data in background and send it to untrusted parties. In both cases, permission-based data access control systems traditionally used in mobile context (e.g. Android or iOS) are too coarse-grained and fail to prevent such data leaks. Once a permission to access a given data source has been granted, there is no way for the user to control how sensitive data is later used by the app.

The situation is similar for IoT apps offered by third-party developers for various IoT platforms. These apps allow the end users to automate some processes in their smart homes, e.g., turn on the lights at a certain time of the day, and to connect various devices and web services together, e.g., send security camera footage to the user-defined cloud storage. Once installed these apps usually have direct access to the user sensitive sensor data and are expected to handle it with care. However, since they are implemented and maintained by independent third-party developers, the actual application behavior may deviate from the expected one, leading to potential data leaks. The device capability-based access control system used by modern IoT platforms (e.g., Samsung SmartThings) is not sufficient to prevent such data leaks, and often results in many apps gaining more privileges than they actually need [96]. As a result, the users are unable to estimate the risks of installing a given app and granting it access to sensor data. Similarly to mobile platforms, IoT ones fail to restrict data sharing and processing capabilities of the installed apps once the access to a data source has been granted.

On a platform level, the data leaks may as well be intentional or accidental. The harvested sensor data of the registered users may be intentionally shared with third-party partners without user awareness. This can be done for analytics, e.g. to improve user experience, or for commercial purposes, e.g. share device usage statistics with device manufacturers or external companies [137, 168, 177]. Accidental data leaks may happen due to platform configuration errors, improper encryption or due to privilege abuse by the system administrators who have direct access to platform infrastructure. The IoT platform may also become a target of an external attack in which confidential user data will be exposed. In all of these cases, the end-users are unable to control where their sensor data resides and who it is shared with, nor they can enforce any data access restrictions. Moreover, there is no mechanism for the users to verify the state of the IoT platform and whether or not it can be trusted.

Finally, on a device level, a data leak may occur when sensitive sensor data is transmitted without proper encryption. In some cases the data encryption method used by the device software is insecure and can be vulnerable to differential analysis attack [53, 105]. In other cases, the encryption may not be used at all, thus allowing any external observer having direct access to the device to recover raw data samples. Alternatively, even with a proper data handling mechanism in place the device may still be vulnerable to fault injection attacks targeting the data encryption procedure [123, 44]. In this case, a strategically placed fault may disrupt the encryption logic or cause the device to skip the encryption completely.

To address the mentioned data privacy and security issues, in the next section we define the requirements for building private-by-design IoT systems.

## 1.3 Private-by-design system requirements

In this thesis, we aim to address the concerns of the IoT users by building systems that offer the following features:

1. **A privacy-oriented IoT system** that allows the end users to manage and control the data flows generated by all of their personal IoT devices. Such a platform must provide a secure environment for data storage and processing, and offer a protection from external attackers trying to exfiltrate sensitive sensor data. The users must remain in full control of their devices, sensor data and the installed IoT apps. This platform must be open and independent from any commercial entity and may not be necessarily compatible with existing IoT platforms, devices and/or apps that often use obscure data handling techniques.

2. **A way for the users to express and enforce their security and privacy preferences** regarding IoT data collection, processing and sharing. This has to be done in a user-friendly yet effective way, so that to accommodate for various user expectations and perspectives on data privacy.

3. **A comprehensive and meaningful report on the collected sensor data origins, types and whereabouts.** The report must contain not only information about the IoT data collected, but also the processing and sharing capabilities of various third parties requiring access to it (i.e., IoT apps and services). Such an insight is crucial, since with a clear understanding of the granularity and purpose of the data collection the users can reason about the privacy and security risks they are facing.

4. **A mechanism to verify the security and privacy properties of the IoT system.** The end-users must be able to verify the state of the system before entrusting it with their IoT devices' data. This will ensure secure data processing and minimize the risks of data leaks.

## 1.4 Contributions

The main contributions of this thesis focus on three target platforms – local hub, for connecting and managing all IoT data flows within a home environment, e.g. smart home; cloud, for a secure data flows processing at the untrusted cloud environment; and, finally, mobile, for on device sensor data access and sharing control. Below we outline the details of each of these three contributions, followed by the description of additional techniques that offer enhancement to these platforms' robustness and security.

1. **A smart hub for privacy-aware data processing.** To address the privacy concerns of the smart home users, we proposed HomePad [231] – a privacy-aware smart hub for home environments. HomePad aims to determine how smart home apps access and process sensitive data collected by smart devices, and to block those apps that try to circumvent the privacy restrictions specified by the users. To achieve this goal, HomePad introduces a novel *dataflow programming model* and an application programming interface (API) for app developers to use. With this model, the apps are implemented as directed graphs of elements, with each element representing a special functional unit provided as part

of a HomePad API. The developers connect these elements together to build the apps of various complexity. By accurately modeling the behavior of the app elements and their interactions, HomePad allows to automatically verify the app's dataflow graph against user-defined privacy policies.

2. **A private-by-design IoT platform.** We proposed PatrIoT [233] – a system that expands HomePad's sensitive data flow control to the untrusted cloud environment. PatrIoT revisits the typical architecture of existing IoT platforms, and provides an alternative design where the end-user retains full ownership and control of IoT data even in a cross-domain setting. It leverages Intel SGX to prevent unauthorized access to the data by cloud providers, and offers home owners an intuitive security abstraction named *flowwall* which allows them to specify easy-to-use policies for controlling sensitive sensor data flows within IoT apps. PatrIoT implements an automatic attestation mechanism which gives users guarantees that their PatrIoT instance runs in a secure environment, and that the confidentiality and integrity of sensor data is preserved.

3. **A middleware for building privacy-aware Android apps.** Mobile apps often have unrestricted access to highly sensitive information obtained from mobile devices' sensors, connected wearables or smart home devices. Thus, providing security mechanisms that prevent data leaks is very important. To this end, we proposed Flowverine [117] – a middleware for building secure-by-design privacy-aware mobile apps running on legacy Android OS. Specifically, it allows developers to write their apps using the dataflow programming model such that all sensitive data flows are made explicit. Flowverine uses both static and dynamic taint analysis techniques to discover and track all the sensitive app data flows. Developers can specify per-app security policies that white-list only some explicitly stated sensitive data flows, and the app users can verify those and employ additional restrictions if needed.

4. **A mechanism to bootstrap trust in third-party IoT software.** IoT apps often rely on third-party libraries to perform a certain operation, e.g. speech recognition or data encryption. However, if a buggy or even malicious implementations of these libraries are used, serious security breaches can take place. To address this problem we proposed to use an N-version Programming (NVP) technique [230]. By using NVP, rather

than depending on a single library implementation, we utilize N different implementations (versions) that must concur to produce the final result. We envision different versions to be developed independently by an open community of developers. Insofar as the developers do not collude, N-version-based modules are no longer dependent on the correctness of any specific library implementation as it is the case for existing IoT apps.

5. **A framework to evaluate the fault-tolerance of IoT device software.** The IoT device software handling sensitive data has to be resistant to faults due to privacy, security and safety reasons. Hardening is a common way to make software fault-tolerant, but the security and performance implications of the selected hardening technique are not always obvious. To address this problem, we proposed Chaos Duck [232] – a framework for automatic software fault-tolerance evaluation. Chaos Duck strategically injects faults in a given software and evaluates their impact on security and performance. With Chaos Duck we offer an invaluable tool for a developer seeking to improve the safety properties of the developed software, and provide a guideline for selecting a hardening technique which helps to avoid the unexpected security pitfalls.

The source code of HomePad, PatrIoT, Flowverine and Chaos Duck systems is publicly available and can be found in the corresponding repositories: [229], [228], [116], [227].

## 1.5 Structure of this thesis

In the next chapter, we introduce a dataflow programming model which plays an important role in the design of our proposed systems. Then, in Chapters 3 4 and 5, we describe our contributions to building private-by-design IoT systems, focusing on local home, cloud and mobile environments respectively. We cover the design and implementation details of the proposed systems, and provide the results of a thorough performance and security analysis. We continue with the description of additional techniques that aim to enhance the security and privacy properties of these systems in Chapters 6 and 7. Finally, Chapter 8 makes a comparative analysis of all the proposed systems and discusses their limitations, and Chapter 9 concludes this thesis by summarizing the main contributions of this work and outlines future research directions.

# Chapter 2

# Dataflow programming model

Our idea of a private-by-design IoT system is one where the end users retain exclusive ownership rights over the sensor data generated by their respective IoT devices: various IoT apps and services can only acquire the access rights and capabilities that a user will explicitly decide to grant them. This is, however, in a stark contrast to an approach used by existing IoT systems that rely on a discretionary access control model in which each app requests permissions to access a given resource (e.g., a sensor reading). While this model provides the users with a basic understanding of app intentions (and a way to decline such requests), permissions fail to capture how the acquired resources will be actually used by an app, and are difficult to manage as the number of smart devices and apps grows. As a result, the apps often obtain more permissions than they actually need to perform a given task (i.e., overprivilege), and may use this opportunity to leak sensitive sensor data [96].

To address the limitations of existing permission-based systems, we propose a dataflow programming model. It provides easy-to-use programming abstractions for IoT developers to build privacy-aware apps or services with all internal data flows made explicit and easy to analyze. The app is implemented as a directed *elements graph*, in which elements represent functional units offered by the system itself (as part of an API) or implemented by the app developer, and the edges describe the only paths through which data can flow within an app. With each element having a well-defined specification, both in terms of interface and expected behavior, such element-based app structure allows for sound and efficient data flow tracking.

In the following sections we describe the main concepts behind the proposed dataflow programming model, namely, element-based app structure, sensitive data flows tracking, and a mechanism used to enforce user privacy and security preferences regarding data collection, processing and sharing.

## 2.1 Element-based app structure

This section describes how privacy-aware IoT apps can be implemented using dataflow programming model, and explains how this model achieves effective sensitive data flows detection and tracking.

### 2.1.1 Elements

Each element of the app graph has five important properties:

- **Element class**: Each element has an associated piece of code which determines its behavior. At runtime, an instance of that code (i.e., a class object) is initialized for each app element.
- **Element ports**: An element can have any number of input or output ports with attached data types. An arrival of a trigger event at any of input ports causes the element's code to execute and process the incoming event in a *First-In First-Out* (FIFO) fashion; at the same time, depending on the element's logic, an outgoing event may or may not be fired at the output port. Elements can optionally have an error port which is used by the system to output internal exceptions and stack traces. Each port type is denoted with a different notation and is statically typed.
- **Parameter string**: Element classes may optionally support parameters to initialize element's state and configure its behavior.
- **Element rules**: An element must be accompanied by Prolog rules that specify the (abstract) types of data sent as output in response to a given input, as well as the corresponding input and output ports involved.
- **Element state**: All elements are stateless. This way sensitive data samples cannot be aggregated between executions. The absence of state also helps to maintain the stability of the system: any failed execution of an element's code may be restarted safely with the same inputs.

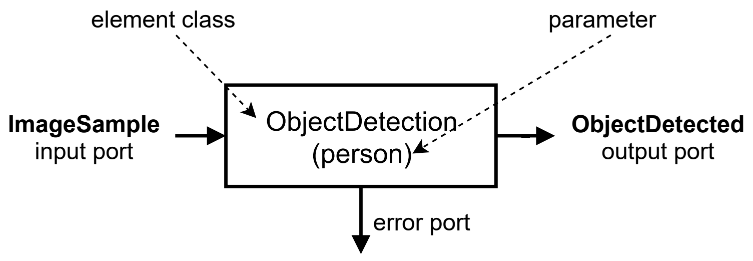


Figure 2.1 – Object detection element example.

Figure 2.1 displays a simple element, named `ObjectDetection`, which performs object detection on a series of input images. The `ImageSample` input port receives a camera frame for analysis, the element's code runs an object detection algorithm, and then sends an event with the list of identified objects on the `ObjectDetected` output port. This specific element is configured to only react to a 'person' object defined in a parameter string.

Our element notation was inspired by the notation used for programming the Click modular router [145]. We adapted and extended Click's notation accordingly, by adding error ports and output rules.

### 2.1.2 Functionality of built-in API elements

Dataflow programming model relies on a set of elements provided as part of an API, which allows for highly flexible app configurations. Some of these elements provide interfaces to physical devices and actuators, while the others offer a time-based functionality with timers and schedules. Next, we present an overview of functionalities provided by some of these elements:

**Interaction with sensors and actuators:** A crucial functionality is to enable IoT apps to access sensors (e.g., thermostats, cameras, and microphones) or actuators (e.g., locks, or light bulbs). The elements of this category act as a *'device shadow'*, mimicking real devices' interfaces and proxying their events and commands. While some of these elements can implement low-level functions such as simply reading / writing from / to a device, others can be more sophisticated. For example, `IPCamera` element can read data from multiple devices, know their location(s), and pull camera frames at a predefined rate.

**Communication with remote endpoints:** Elements can also enable communication with external entities. An `HttpRequest` element, for instance, allows an app to issue HTTP requests to any web service. An `OAuth2` element allows for easy integration with external services using OAuth2 authorization. There are additional elements for communication with mobile endpoints. A `PushMessage` element, for instance, allows to send a push message to any registered user's mobile phone, while an `SmsMessage` element sends a short text message to a given number.

**Data transformation:** This class of API elements aims to provide data transformation functionality. Examples include audio / video encoding, data compression or encryption, and data anonymization. The latter is exceptionally useful when sending sensitive data to external entities.

**Computation on sensitive data:** These elements provide various data processing capabilities, for instance, speech or face recognition, object detection, and image classification, among others. `ObjectDetection` element described earlier belongs to this category as well. These elements collectively offer a functionality that app developers would otherwise have to implement themselves, which can be a challenging task sometimes.

**Time-based dataflow control:** Some of the API elements allow to introduce time constraints on the app's execution logic. For instance, with the `TimeController` element, the app developers may provide a way for the user to specify time windows when the app will be active or not. With this element, for instance, an IP camera app may be allowed to record the video when the user is not at home (e.g. during the working hours), and denied to do so otherwise. Alternatively, a `RateLimit` element provides a way to specify the maximum rate for data transmissions. Following the IP camera example, the video stream might be restricted to one frame per second, when `RateLimit` element is configured with a '1 sec' parameter. Finally, `Timer` and `Schedule` elements allow an app to perform a given action after a certain time period or using a pre-defined schedule. Overall, this type of elements allow for the enforcement of various time-constraining rules by just modifying the app graph accordingly.

**Error handling and debugging:** Specific elements can help handle app errors and debugging, e.g., for sending bug reports to an app provider. To preserve anonymity, there can be instances of such elements that, in addition to packaging memory dumps or exception related data, can first anonymize that data so as to prevent exfiltration of sensitive user information.

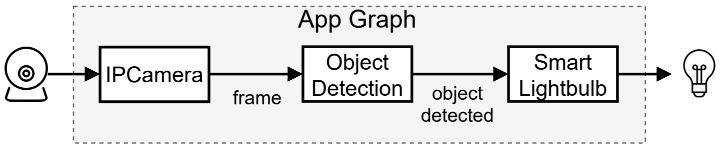


Figure 2.2 – Flow graph of AutomaticLight app.

### 2.1.3 Flow graph

Elements can be coupled together to form a directed graph, which we call *flow graph*, as long as their input and output ports are compatible and operate with the same data types. Two elements can be connected using asynchronous unidirectional links – in *simplex mode* – or synchronous bidirectional links – in *duplex mode*. A flow graph makes information flow explicit across elements and can be used to fully describe the data flows inside an IoT app.

Consider, for instance, an element-graph of a simple app named AutomaticLight showed at Figure 2.2. The app collects camera frames from `IPCamera` element (which serves as a proxy to a physical camera device), forwards those to an `ObjectDetection` element, which in turn sends a corresponding event to the `SmartLightbulb` element that turns on the lights whenever a person is detected in front of a camera.

To execute this app, a runtime system instantiates each element of the flow graph as a single object and establishes internal communication channels responsible for forwarding the messages between these elements according to the element connections as specified by the flow graph; no other data flows are allowed between elements other than those explicitly declared in the flow graph. Thus, from Figure 2.2 we see that camera frames are produced by `IPCamera`, which acts as a *data source*, and from there flow down to `ObjectDetection` element, which acts as a *data sink*. Then, another data flow occurs when `ObjectDetection` element (source) sends an `object detected` event to the `SmartLightbulb` element (sink).

### 2.1.4 Untrusted elements

Each element implements a specific functional unit and is provided as part of an API. Since built-in elements are part of the trusted computing base, we call them *trusted elements*. At the same time, app developers can also implement app-specific elements to be shipped along with the app package, which will be

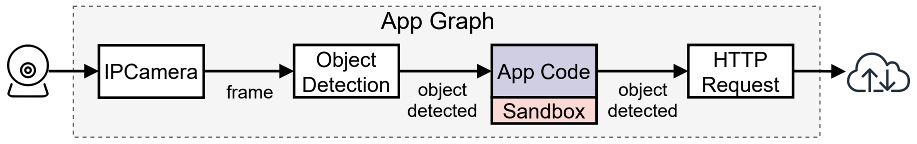


Figure 2.3 – Flow graph of SecurityAlert app.

incorporated into the app's flow graph and instantiated at runtime. Since code of such elements has not been verified and cannot be deemed to be correct, we call them *untrusted elements*.

Figure 2.3 illustrates an example of an app that uses both trusted and untrusted elements, colored respectively in white and grey. The SecurityAlert app automatically sends an alert to a security company (e.g., ADT) whenever a person is detected through the camera installed at the living room. The app's flow graph works as follows: camera `frame` from the `IPCamera` element arrives to a `ObjectDetection` element which in turn sends an `object detected` event to the `AppCode` element when a person is detected.

The `AppCode` element implements an app-specific logic and is provided by the app developer. It prepares a payload for an HTTP POST request sent to a security company through an `HTTPRequest` element. To enforce proper protection against buggy or malicious behavior, this element runs inside an individual sandbox so that to prevent direct access to sensitive data sources (e.g., camera feed), circumventing pre-defined graph connections.

### 2.1.5 Programming model

The dataflow programming model for writing IoT apps is rather simple. The app package essentially consists of a JSON file describing the app's flow graph, i.e. a *manifest*, which declares the names of all the trusted and untrusted element instances of the apps' flow graph and their respective connections, and a code implementing the logic of the app's untrusted elements (if any). The latter can be written in any programming language supported by the runtime.

Interestingly, using this model, one can implement IoT apps without writing a single line of code by simply arranging compatible trusted API elements. This is a case, for instance, for AutomaticLight app described earlier. Since its flow graph does not rely on any untrusted element, its app package consists of the manifest file only. Listings 2.1 and 2.2 show the manifest files for both AutomaticLight and SecurityAlert apps for comparison, and Listing 2.3 displays

```
{ "name": "AutomaticLight",
"elements": [ {
  "name": "IPCamera", "type": "IPCamera",
     "config": { "interval": 1000 } }, {
  "name": "ODetector", "type": "ObjectDetection",
     "config": { "object": "person" } }, {
  "name": "SmartLight", "type": "SmartLightbulb" } ],
"connections":[ {
  "from": "IPCamera", "outport": "FramePort",
  "to": "ODetector", "inport": "SampleImage","mode":"simplex"},{
  "from": "ODetector", "outport": "ObjectDetected",
  "to":"SmartLightbulb", "inport":"TurnOnLight","mode":"simplex"
}]}
```

Listing 2.1 – Manifest file of the AutomaticLight app.

```
{ "name": "SecurityAlert",
"elements": [ {
  "name": "IPCamera", "type": "IPCamera",
     "config": { "interval": 1000 } }, {
  "name": "ODetector", "type": "ObjectDetection",
     "config": { "object": "person" } }, {
  "name": "AppCode", "type": "untrusted" }, {
  "name": "HttpReq", "type": "HttpRequest",
     "config": { "endpoint": "adt.com" } } ],
"connections":[ {
  "from": "IPCamera", "outport": "FramePort",
  "to": "ODetector", "inport": "ImageSample","mode":"simplex"},{
  "from": "ODetector", "outport": "ObjectDetected",
  "to":"AppCode", "inport":"ObjectDetected","mode":"simplex" },{
  "from": "AppCode", "outport": "HttpPostPort",
  "to": "HttpReq", "inport": "HttpPostPort","mode":"simplex"}]}
```

Listing 2.2 – Manifest file of the SecurityAlert app.

a code sample of the SecurityAlert's `AppCode` element written in JavaScript.

Untrusted elements must be written by the developer as independent classes. In SecurityAlert app, the `AppCode` element's code extends the `AppElement` class (see Listing 2.3). The App function is the entry point for the events received from other elements in the app graph (line 3). The parameters are as follows: *source* defines the type of the element that generated the incoming event; *event* object holds the type and value of the event; and *portType* specifies the port type on which the event was received. For elements with multiple input ports, the latter parameter is essential to determine the execution logic. After receiving a new object event, the `AppCode` element prepares a payload for an

```
1 class AppCode extends AppElement {
2   constructor() { super(); }
3   App(source,event,portType) {
4     let payload = JSON.stringify({person: event.value};
5     let headers = {'Content-Type': 'application/json',
6                    'Authorization': 'Bearer BQCw82jABt'};
7     port.HttpPostPort.call(payload,headers) }
8 }
9 module.exports = AppCode;
```

Listing 2.3 – Implementation of AppCode element of the SecurityAlert app.

HTTP POST request (line 4) and declares the headers for the request, which include the payload type (line 5) and an authorization token (line 6). On line 7 the App calls a function which sends a new event on an `HttpPostPort` port with the specified payload and headers parameters. The URL for this request is specified in the manifest file as the `HttpRequest` element's config.

## 2.2 Privacy policy specification

Both AutomaticLight and SecurityAlert apps use IP camera feed to detect objects and then act upon those. The AutomaticLight app uses this information to control the lights at home, while the SecurityAlert app sends an alert to a security company. Both these apps require access to a sensitive data source, i.e., IP camera, but only one of them has access to network resources. Unlike AutomaticLight app which retains all the data flows within the perimeter of home environment, the data flows of SecurityAlert app are more concerning since they extend to a remote third-party cloud service. While the SecurityAlert app may come from a legitimate source and a respected developer, the end users might still want to verify that this app cannot potentially leak raw camera images to a security company, and by doing so, violate their privacy.

To prevent such data leaks, the users can define a privacy policy consisting of a set of rules that specify disallowed or allowed app flows of specific data types (e.g., a camera frame) from specific data sources (e.g., IP camera) to specific destinations (e.g., Internet). These rules are evaluated sequentially and applied atomically by the app runtime. Any app flows violating the rules defined in the privacy policy will result in the whole app being blocked.

Next, we describe the privacy policy rules' syntax, and present a user interface (UI) to specify these rules in an easy-to-understand, yet efficient way.

### 2.2.1 Rule syntax

To characterize app flows and to easily specify policies for blocking or allowing them, dataflow model operates with certain objects named *endpoints* that the apps may access to perform a given function. Endpoints represent system resources that can act as producers (i.e. data *sources*) or as consumers (i.e. data *sinks*) of sensor data. Each endpoint fits into one of three classes:

- **IoT endpoint:** Represents a particular IoT device or device type, e.g., IP camera. Each device type can generate specific types of sensor data, e.g., `Video` or `Image` data types. Concretely, each deployed IP camera is represented by an IoT endpoint featuring its own ID and an alias assigned by the end user, e.g., `LivRoomCam` for the living room cam.
- **Mobile endpoint:** Represents a mobile device used to interact with the system. It is identified by the phone number or other attributes, e.g., the IMEI, and has a user-defined alias such as `MyPhone`.
- **Web endpoint:** Represents an Internet location in the form of HTTPS URL patterns. For authenticated web services based on OAuth2, the end user's credentials must also be provided. Web endpoints can be labeled with aliases, e.g., `ADT` to indicate any host under the domain `www.adt.com`, or with a general `Internet` wildcard to denote any potential web endpoint.

A *data flow* can then be defined by the transfer of a specific sensor data type between source and sink endpoints. Following this definition, the format of a privacy policy rule can be expressed as follows:

$$\textbf{allow} \mid \textbf{block}\ \langle\text{data type list}\rangle\ \textbf{from}\ \langle\text{source endpoint list}\rangle$$
$$\textbf{to}\ \langle\text{sink endpoint list}\rangle\ [\textbf{at}\ \langle\text{time period list}\rangle]$$

The "allow" or "block" keywords indicate the *rule type*, i.e., whether the rule allows or blocks the data flows matched by the rule, respectively. The data type list indicates one or multiple comma separated types of data to be matched. They can be simple types, e.g., `Image`, or the wildcard `Everything` to indicate all possible simple types. The keywords "from" and "to" are followed by a list of source and sink endpoints, respectively, which may specify individual

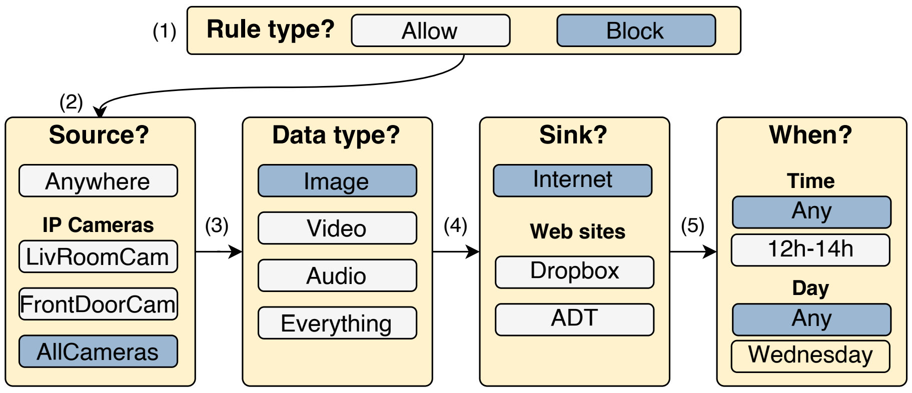


Figure 2.4 – Schematic representation of the UI workflow to specify a privacy policy rule using dataflow model.

endpoints, e.g., `LivRoomCam`, and/or include wildcards, such as `Anywhere` for all valid endpoints, and driver-specific terms, e.g., `IPCamera` to refer to all IP camera endpoints. Optionally, it is possible to specify time restrictions by using the keyword "at" followed by a time period, e.g., "12:00-14:00", and days of the week.

Taking the SecurityAlert app as an example, the end user's concern regarding the apps' access to both camera and web endpoints could be expressed as the following 'block' policy rule:

**block** Image **from** IPCamera **to** Internet

With this rule, app flows carrying camera frames (of `Image` type) from any connected camera to any web endpoint will be blocked. Furthermore, additional rules could be provided to restrict app activity even more (if needed). Having defined the privacy policy rule syntax, we will now describe how such rules can be specified by the end users in a simple and effective way.

### 2.2.2 User interface for policy specification

Manually specifying policy rules using the syntax presented above can be cumbersome for untrained users. To help with this procedure, our dataflow model exposes a simple UI that guides the user along a five step process (see Figure 2.4) which helps the user to reason in terms of privacy-sensitive/insensitive data flows he intends to allow/block.

To create a new rule, the user starts by selecting the rule type, i.e., “allow” or “block” (1). Then, he picks the source endpoint (2), tells what data types from that source he wants to allow or block (3), indicates the sink endpoint (4), and optionally provides a temporal restriction for the rule (5). To avoid overwhelming the user with too much information, in step 2, the system only displays existing valid source endpoints. Once the user selects the source endpoint (3), only the data types that can be generated by that endpoint are showed. Similarly, in step 4, only valid known sink endpoints are displayed. In a separate UI view, the user can manage the privacy policy: list all rules, change their order, modify them, or delete them.

More sophisticated policies, e.g., based on particular device state or certain data values, can be supported but must be carefully conceived as they may increase the complexity of the UI. In general, we believe that a good UI must be simple and provide a limited set of default configuration options, so as to avoid the negative effects of *decision fatigue* among the end users [95, 102, 94]. Any advanced settings could be offered to all experienced users in separate views.

## 2.3 Application verification

To enforce a privacy policy the dataflow programming model implements an app verification algorithm which decides whether or not the internal app data flows violate the policy rules. The verification is performed by first creating a model of the app flow graph in Prolog (named *flow graph model*), and then issuing a set of queries to determine the existence of illegitimate data flows. We consider a data flow to be illegitimate if it violates any of the policy rules. Next, we explain how the flow graph model is generated, and describe the verification process in details.

### 2.3.1 Flow graph formal modeling

To create a flow graph model of a given app, we need to analyze its flow graph. During this analysis, we generate a set of Prolog facts and rules describing the elements the app depends on, their functions, connections, and the data types they operate with. These facts and rules will then be used to identify and verify all the app’s data flows.

The generation of the model entails three steps:

```
el(ipcamera).
el(odetector).
el(appcode).
el(httprequest).
con(ipcamera,cameraframe,odetector,sampleimage).
con(odetector,odetected,appcode,odetected).
con(appcode,httppost,httprequest,httppost).
```

Listing 2.4 – SecurityAlert app's flow graph facts (elements & connections).

**1. Model the flow graph structure**: We begin to model the flow graph of an app by generating a set of facts in Prolog that declare the elements and their connections. The general format is represented by facts (2.1) and (2.2):

$$\textbf{el}(X). \tag{2.1}$$

$$\textbf{con}(X, P_{out}, Y, P_{in}). \tag{2.2}$$

Fact (2.1) declares $X$ to be an element of the graph, and fact (2.2) declares a connection from $X$ element's output port $P_{out}$ to $Y$ element's input port $P_{in}$. Following this logic, the SecurityAlert app's graph (see Figure 2.3) can be described with four elements and three connections, as shown at Listing 2.4.

**2. Model the elements' attributes and their types:** Each element operates on and/or generates a unique set of data attributes. For instance, an `IPCamera` element generates a `frame` attribute of `Image` type, and the `ObjectDetection` element receives this `frame` and generates an `object` attribute of `Boolean` type (i.e., a person is either detected or not). Each element's port is associated with a certain attribute it supports. So, `IPCamera` element's `CameraFrame` port is associated with a `frame` attribute. Having this *'port-attribute-type'* relation helps to achieve two goals: verify the correctness of graph connections (i.e., only compatible elements' ports can be connected), and track specific data attributes propagation between the connected app elements. Facts describing the element's supported attributes and their types, as well as corresponding ports, are expressed as follows:

$$\textbf{attr}(X, Y). \tag{2.3}$$

```
attr(frame,ipcamera).
attr(object,odetector).
attrtype(frame,image).
attrtype(object,boolean).
portattrtype([],[image],cameraframe,ipcamera).
portattrtype([],[boolean],odetected,odetector).
portattrtype(any,any,httppost,httprequest).
```

Listing 2.5 – SecurityAlert app's flow graph facts (attributes & types).

$$\textbf{attrtype}(X,\ T). \tag{2.4}$$

$$\textbf{portattrtype}(T_{in},\ T_{out},\ P,\ Y). \tag{2.5}$$

Fact (2.3) declares that element *Y* can generate an attribute *X*. Fact (2.4) specifies the type *T* of that *X* attribute. Finally, fact (2.5) declares that port *P* of the *Y* element can only receive attributes of type $T_{in}$ on its input and may output the attributes of type $T_{out}$.

Listing 2.5 shows attribute facts for the SecurityAlert app. Both `object` and `frame` attributes, as well as their corresponding types (Boolean and Image) are declared. Then follow the facts describing the supported attribute types for all the elements' ports used in the flow graph: a `odetected` port of the `ObjectDetection` element (named `odetector` for brevity), and a `cameraframe` port of `IPCamera` element. A fact describing `httppost` port of `HttpRequest` element contains `'any'` wildcard and can thus receive any type of attributes. This is due to the nature of an HTTP request which can carry virtually any payload. In our case this means that any data attribute can be received and sent by the `HttpRequest` element.

**3. Model the behavior of trusted elements:** The next step is to model how each element generates its outputs. Typically, an output is a function of the element's inputs and / or of the element's internal behavior. Since this function is dependent on the specific implementation of the element, to model element behavior, it is required that each element is associated with its corresponding Prolog rules termed *element rules*. These rules express how the element outputs are produced and the possible dependencies of these outputs from the element inputs. When creating a flow graph model, the rules of all the app elements are retrieved and added to the model. In general, element rules take the following form:

```
out(ipcamera,cameraframe,frame,_).

out(odetector,odetected,object,true) :-
    in(odetector,imagesample,frame,_).

out(odetector,odetected,object,false) :-
    in(odetector,imagesample,frame,_).
```

Listing 2.6 – Output rules of SecurityAlert app's trusted elements.

$$\textbf{out}(X, P_{out}, A_{out}, V_{out}) \text{ :- } \textbf{in}(X, P_{in}, A_{in}, V_{in}). \tag{2.6}$$

This rule states that the output data attribute of element *X* on port $P_{out}$ is defined as $A_{out}$ with value $V_{out}$ and depends on the input data attribute $A_{in}$ with value $V_{in}$ received on $P_{in}$ input port. Informally, rule (2.6) indicates that an element will produce a declared output as long as a given input is provided. The more formal declarative interpretation of this Prolog clause is: "An output $A_{out}$ is produced from element *X*, if an input $A_{in}$ reaches that element". Note, however, that each element may have a variant of this rule, or may even require more than a single rule. Some elements might not depend on any input and only generate output events (e.g. `IPCamera`).

Listing 2.6 features the output rules for SecurityAlert app's trusted elements. The `IPCamera`'s out rule indicates the port (`cameraframe`), attribute (`frame`) and a value of that attribute ( _ ) returned by the element. We use an anonymous Prolog variable `'_'` to denote any value of the camera frame, since the exact value is irrelevant for flow modeling in this specific case. In the `ObjectDetection` element, since two outputs are possible (with true or false values), it is necessary to specify two rules, one for each output. The element `HttpRequest` is omitted in the table. Since it has no output ports there is no need for any specific rules.

**3. Model the behavior of untrusted elements:** Just like in the case of trusted elements, untrusted elements must also be accompanied by Prolog rules that characterize the element's output data types. However, the app programmer cannot be relied upon to write these rules. To this end, we define a common rule for all untrusted elements. We take a conservative approach in modeling such elements by assuming that an untrusted element will try to forward all input data to the output ports in an attempt to leak as much data as possible. Thus, we can model an untrusted element using two generalized rules:

$$\textbf{untrusted}(X). \tag{2.7}$$

$$\textbf{out}(X,\ \_,\ Y, Z\ ) \text{ :- } \textbf{untrusted}(X),\ \textbf{in}(X,\ \_,\ Y,\ Z). \tag{2.8}$$

Rule (2.7) declares a specific element $X$ as *"untrusted"*. Rule (2.8) then says that if an element $X$ is untrusted then all its input attributes $Y$ and their corresponding values $Z$ received on any input port can be forwarded to any of its outputs. In case of SecurityAlert app, the `AppCode` element would be declared as untrusted and its outputs will be modeled using (2.8) rule.

**4. Model the connection behavior**: Now that the structure of the flow graph and the behavior of each element has been modeled, the last missing piece is to model the behavior of the graph's connections, which are responsible for propagating the outputs of upstream elements to the inputs of downstream elements. To model this behavior, we add rule (2.9):

$$\textbf{in}(X,\ P_{in},\ Y,\ V) \text{ :- } \textbf{con}(Z,\ P_{out},\ X,P_{in}),\ \textbf{out}(Z,\ P_{out},\ Y,\ V). \tag{2.9}$$

This rule can be read as follows: if an element $Z$ outputs a data attribute $Y$ with a value $V$ on its output port $P_{out}$, and there exists a connection between $Z$ and another element $X$ using corresponding $P_{out}$ and $P_{in}$ ports, then $X$ receives attribute $Y$ as input on its input port $P_{in}$.

With rule (2.9) the behavior of the flow graph is now completely specified. It is then possible to proceed with the automatic data flows tracking and verification against the user-defined privacy policy rules as explained next.

### 2.3.2 Information flow tracking

Based on the app model, it is possible to determine all possible data flows within that app. A data attribute is originally created at a source element and can be propagated through a chain of interconnected elements until it may

```
1 ?- flows(ipcamera,frame,odetector).
2 true.
3 ?- flows(ipcamera,frame,httprequest).
4 false.
```

Listing 2.7 – Query results for SecurityAlert app.

potentially reach a sink element. Thus, determining if data attribute $Y$ flows between any given source and sink endpoints – $X$ and $Z$, respectively – can be laid as the problem of checking if that attribute $Y$ can be observed as an input to $Z$. This problem can be formulated by the following rule:

$$\textbf{flows}(X,Y,Z) \text{ :- } \textbf{el}(X), \textbf{el}(Z), \textbf{attr}(Y, X), \textbf{in}(Z,\_,Y,\_). \tag{2.10}$$

By issuing this query to a first-order logic engine, existing solutions will be found by unifying it against the topology and the predicates of the app's flow graph model. If there is a sequence of interconnected nodes that propagate a data attribute from $X$ to $Z$, a result will be found and assigned to $Y$. The verification engine uses this technique to automatically query all the possible '*source-attribute-sink*' triplets of the app graph. The results can be used for two main purposes: *application profiling* and *policy enforcement*.

### 2.3.3 Application profiling

Application profiling allows the user to analyze the flow graph of an app and learn how the information can flow within it by determining, (1) what kind of information can be accessed by the app, (2) where this information can be obtained from, and (3) where this information can propagate to. The flows rule is used for this purpose. For example, considering the SecurityAlert app, in order to determine if raw camera frames from `IPCamera` can reach `ObjectDetection` and `HttpRequest` elements, we can issue two **flows** queries as shown at Listing 2.7.

The results of these queries state that raw camera frames can arrive at the `ObjectDetection` element (line 2), but not to the `HttpRequest` element (line 4). Looking at the graph of SecurityAlert app (see Figure 2.3) we can confirm that: only object detection events returned by `ObjectDetection` element can arrive to `HttpRequest` element.

Using this simple technique the Prolog engine can automatically query the app model and determine if any of the sensitive data attributes can arrive to a given element. The results are returned back to the user in a form of a report listing all the devices and their corresponding data attributes the app has access to, and how those propagate inside the app flow graph. Such information is essential when evaluating the privacy properties of a given app, since it allows to determine the app's data access, processing and sharing capabilities without relying on developer-provided and potentially inaccurate app description. The very same information helps the user to understand the potential risks of installing a given app and whether sensitive data access requests are justified.

### 2.3.4 Policy enforcement

While application profiling allows to generate "privacy reports" of various apps, policy assessment aims to ensure that an app can be installed only if it respects the privacy rules specified in a privacy policy. This process is completely automated and allows the users to define their privacy policy once and then enforce it on all the apps already installed or those installed in the future.

The enforcement algorithm is similar to the one used for app profiling. The verification engine checks all the possible combinations of app elements pairs, and for each pair executes the flows query on the app flow graph. Each discovered flow is represented by its corresponding '*source-attribute-sink*' triplet. Each of these triplets are then matched against every rule defined in the privacy policy sequentially. If at least one triplet is in conflict with any of the policy rules, the app installation is halted.

The policy rules are evaluated dynamically at runtime and can be modified by the user at any time. When new rules are added, all the currently installed apps are re-evaluated and enabled or disabled based on the evaluation results. Such a dynamic approach allows to enforce new privacy preferences immediately regardless of the previously authorized app permissions. This is useful in cases when new devices are added, or when external events prompt a stricter privacy control, e.g., newborn in a family. To illustrate the latter, below we provide examples of the privacy policy rules that can be specified by the user:

$$\textbf{block}\ \text{Frame}\ \textbf{from}\ \text{BabyMonitor}\ \textbf{to}\ \text{Internet.} \tag{2.11}$$

$$\textbf{block}\ \text{Motion}\ \textbf{from}\ \text{MotionSensor}\ \textbf{to}\ \text{SmartLight},\ \textbf{except}(\ \textbf{at}(17{,}23)). \tag{2.12}$$

**block** State **from** SmartLight **to** SMSMessage. (2.13)

The policy (2.11) addresses the concerns of young parents fearing that camera frames from their baby monitor may be secretly sent to the Internet by some of the installed apps. The policy (2.12) describes a device-to-device data flow and blocks the motion sensor events from triggering a smart light when the user is not at home, but allows that flow otherwise (from 5 till 11pm). Finally, the policy (2.13) blocks frequent smart light state change events from being sent via SMS notifications, which allows to reduce operational costs and save available resources for important events, e.g., smoke alerts. With this rule, any app attempting to send such events through SMS message will be blocked.

## 2.4 Related Work

The concept of dataflow programming has been known for a while and dates back to the 1970s [81, 82, 146, 108]. Initially, it was proposed as a way to improve the performance of computer systems and take advantage of their increasing parallelism. Conventional imperative programming languages were not suitable for this task due to their inherited side effects and memory locality problems. Hence, a novel dataflow programming model was proposed in which a program was represented as a directed graph. The nodes of such graphs implemented primitive arithmetic operations, and edges acting as FIFO queues defined how the data flowed between the nodes [35, 79]. A new input activated the graph and triggered the execution of the nodes that receive this input, compute on it, and forward the results to their immediate neighbor nodes (if any). Each node was stateless and behaved independently, and thus could process any subsequent input immediately after the current one. In fact, this was a key advantage of the dataflow programming model as compared to the conventional programming models, since it allowed for several nodes to operate in parallel without waiting for each other to finish. While we apply dataflow programming in a different context and with different goals, we still rely on the same properties of the dataflow graph, e.g., stateless nodes and FIFO edges.

Graphs have also been successfully used to represent the data access rights propagation in secure languages and systems. Spiessens et al. [133, 203, 204], for instance, defined a SCOLL language to model authority propagation in complex capability-based systems as a graph of interacting entities, and proposed a constraint solver SCOLLAR to determine any potential security violations. Their approach allows to verify if the safety requirements can be guaranteed given a certain set of capabilities of the partially trusted entities, and provides insights into building provably secure and safe systems. The graph-based representation of the authority propagation is essentially the same as the dataflow graph in our model: the capabilities of a subject in an access graph closely resemble the capabilities of the app developer available through elements in the dataflow graph. In our programming model we too seek to verify whether a given entity, be it an app developer or a platform provider, can obtain a capability that will allow it to violate user privacy or security preferences. Our work serves as an extension of Spiessens et al. contributions and goes beyond capability-based verification, allowing, among other things, to reason about the exact nature and context of data flows that occur in a system.

There's also a considerable amount of work in the field of formal verification [160, 61, 147, 33]. Deshotels et al. [83] use Prolog to model the policies of iOS container sandbox profiles and discover vulnerabilities in them. Still, this solution does not directly address our problem as that although it uses Prolog, it has a broader focus on assessing security rather than privacy properties.

Various software verification techniques have been proposed and used for quite some time [188, 191, 128, 120, 87]. State-based model checking methods, for instance, allow to verify the safety properties of a given software by checking all the possible states it can reach. Although these methods can provide high precision, their main shortcoming is a so-called *state-space explosion* – an exponential growth of system states which often makes a model checking ineffective. In our dataflow programming model we leverage some model checking ideas but operate with data operations instead of application states in order to fully model any IoT app. This allows to improve the verification performance dramatically and overall makes the model checking approach more practical. Furthermore, while classic model checking techniques are more suitable for control-flow analysis, our approach allows us to perform sophisticated and precise data-flow analysis, which is essential for user privacy and security guarantees.

## 2.5 Summary

In this chapter, we have presented the key concepts of the dataflow programming model used to build privacy-aware IoT apps. These concepts will guide the reader through the rest of this thesis, where we explain how they can be applied in real-life scenarios.

IoT systems often deal with highly-sensitive sensor data which when exposed to unauthorized parties can cause serious harm to the end users' privacy. Traditionally, permission-based access control systems have been used to restrict various apps' access to specific devices and their corresponding sensor data. However, permissions alone do not offer the desired privacy and security guarantees, as they only restrict access to a given device, but not the data processing and sharing capabilities of the app after the access has been granted. As a result, numerous apps end up gaining more permissions than they actually need to perform a given task, and often have unrestricted network access which allows them to leak sensitive data undetected.

To address the issues of permission-based IoT systems, we proposed an alternative clean-slate approach to building IoT systems. It is based on the dataflow programming model which provides a set of important features for ensuring sensor data privacy and security. Following this model, each app is represented as a directed graph of elements, with each element having well-defined behavior and data processing capabilities. Such an app structure allows for sound and efficient data flow tracking within the app. Combined with the mechanism for the end users to specify their privacy and security preferences regarding sensor data collection and sharing, this model allows to automatically assess the privacy properties of the IoT apps, and enable or block those based on their compatibility with the policy rules.

In the next chapters we will describe the design and implementation details of the IoT systems built using the dataflow programming model. We will analyze their performance and compare them to state-of-the-art solutions, as well as point out their strengths and weaknesses.

# Chapter 3

# HomePad: a private smart hub

## 3.1 Introduction

One of the biggest barriers to the widespread adoption of smart home technology involves concerns over user privacy. Today, smart home platforms such as Samsung SmartThings [21], Apple HomeKit [9] or Amazon Alexa [5] rely on a cloud-first approach, in which numerous connected devices, from smart lights and locks to thermostats and cameras, constantly stream sensor data to platforms' cloud servers for further processing, backup, and visualization. Ironically, however, the owners of these devices and consequently the users of aforementioned platforms have little to no control over how much and what kind of data is being collected or who it is being shared with, and rarely have a clear understanding of why this data is collected in a first place.

Moreover, the terms of use of smart home platforms tend to be extremely aggressive, forcing the end users to grant those platforms a lifetime and irrevocable, royalty-free license to use, share, display, and otherwise fully exploit the connected devices' data and user actions. In practice, this means that users have to yield full control of their data if they want to benefit from the desired services. However, such an aggressive approach is in conflict with the end users expectations and views on their data privacy. In fact, a recent study [213] reported that 87% of US consumers "are concerned about their personal information being collected and used in ways they were unaware of"; 27% mentioned this concern as the "main reason they do not currently own a smart device". Unfortunately, such fears are all too well justified, backed up by anecdotal cases of stealthy data theft [99, 106, 73] or undisclosed data sharing [93, 159] by IoT providers. Such fears end up hindering not only the sales of smart devices but also undermining consumers' trust in smart home technology. Smart home platforms also face increasing pressure from many countries to uphold strict personal data handling policies. Notably in Europe, since May 2018, the GDPR regulations [10] require service providers to pay formidable penalty fees in case of personal data misuse or user privacy violations. However, their common practice of aggressively collecting raw sensor data and shipping it down to their cloud servers can only increase such risks.

To address the privacy concerns of existing and future IoT consumers, we propose to shift the control of smart devices from the platform provider to the end users, so as to offer greater transparency and control over how their data is collected and used. To this end, we present HomePad – a privacy-aware smart hub for home environment, which extends the architecture of smart home platforms with the ability to process IoT device data and execute various third-party IoT apps at the edge. HomePad implements a *trusted hub device* deployed at home that manages connected smart devices and installed apps locally without necessarily depending on a service provider's centralized cloud services. As a result, whenever the functionality of an app does not strictly require the shipment of data to the cloud, the sensor data can be collected and processed locally, therefore reducing the risks of data exposure and misuse at the platform provider's backend.

HomePad apps follow a dataflow programming model described in Chapter 2 and can be implemented as *directed graphs of elements* provided as part of a HomePad API. Using these elements HomePad apps can perform numerous operations, e.g., interact with various sensors and actuators, make network calls, and perform various computations on sensor data (e.g., speech or face recognition, voice synthesis, or data anonymization). Furthermore, following the dataflow programming model HomePad provides a mechanism that allows users to easily examine whether a given app has the ability to violate specific privacy concerns expressed in a user-defined policy. This verification is performed at install time so that the users can refuse to install the app if it violates their privacy expectations.

**Threat model:** From a privacy perspective, HomePad aims to make users aware of how their sensor data is accessed and processed by the apps, and eventually prevent the installation of apps that the users deem to be too privacy-invasive. Therefore, our main adversary consists of potentially buggy or mali-

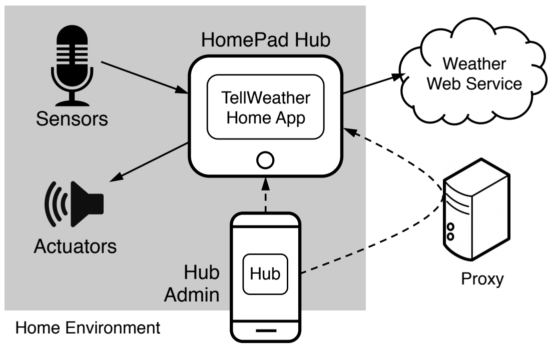


Figure 3.1 – HomePad deployment.

cious apps aiming to extract privacy-sensitive information from home sensors. An app may try to attain this goal by leveraging legitimate operations provided by the HomePad API. However, we assume that the hub platform itself is part of the trusted computing base. In particular, we do not focus on attacks which try to exploit bugs in the hub software or hardware, or attacks aimed at leveraging existing vulnerabilities in the smart devices themselves. We assume that the hub hardware is correct, that the software that implements the hub system is correct, and that potential software updates to the hub have been implemented and signed by trustworthy entities. We focus only on attacks that aim to exfiltrate sensitive data extracted from smart devices connected to the hub. Consequently, we do not prevent privacy breaches from rogue devices deployed at home that can connect to the Internet bypassing our hub.

## 3.2 Design

We designed HomePad with an idea that smart home users should be able to manage their devices and benefit from various third-party apps without necessarily depending on centralized smart home platforms and their underlying infrastructure. Instead, there is a locally deployed HomePad hub through which the users can control their devices directly and manage their installed apps. Such an approach not only allows to reduce the network latency, but also provides better user privacy protection since sensitive sensor data is being collected, processed and acted upon entirely within the home perimeter.

Figure 3.1 shows a HomePad deployment in a home environment. HomePad consists essentially of a smart hub that controls access to all smart devices at home and provides a platform for the execution of various IoT apps, called *home apps*. The HomePad hub provides an administration interface through which the homeowner can access the hub directly or tunneled through a proxy and manage it, e.g., to install apps, register new smart devices, install hub software extensions, or set up privacy policies.

Figure 3.1 also features a simple home app – TellWeather – installed at the hub, which listens for an audio command (e.g., "Tell weather in LA"), issues an HTTP request to a weather service, converts the response into audio signal, and forwards it to a speaker. All HomePad apps follow a dataflow programming model and are built by combining elements offered as part of the HomePad API or provided by the app developers. Due to their element-based structure HomePad apps have all of their internal data flows and data transformations made explicit and subject to evaluation based on their compliance with the user-defined privacy policy rules. The results of this evaluation determine if a given app is allowed to be executed at the hub or not.

### 3.2.1 Architecture

HomePad's architecture internally relies on several involved parties:

- *Users* interact with home apps and control their devices via HomePad's management interface.
- The *hub administrator* (typically the homeowner) maintains the hub, e.g., by installing apps and elements, setting up privacy policies.
- *App developers* create HomePad apps, which involves writing a manifest file specifying the app flow graph, supplying the code of untrusted elements (if any), and assembling the app package.
- *Element developers* implement new trusted elements for app developers to use in their apps. For each element they must write the element code, a Prolog rule describing element's input and output data types, and potentially a driver to be installed on the hub.
- *Platform developers* write and maintain the code of the HomePad core system installed in the hub. To this end, we envision HomePad's core code to be maintained by a trustworthy *code maintainer*, which can be a

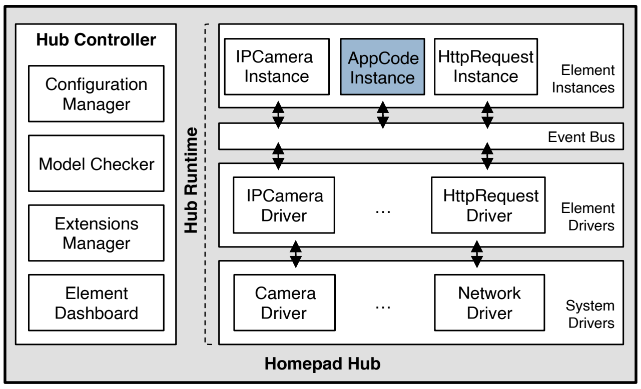


Figure 3.2 – HomePad hub architecture.

single reputable entity or a consortium, and released open source to help detect potential code vulnerabilities.

Figure 3.2 represents the main software components of the HomePad system running on the hub. The model checker manages a repository of Prolog rules and exposes a management interface to allow for privacy verification of apps. This repository contains: the output rules of elements installed in the system, the flow graph models of all apps installed in the system, and the privacy rules defined by the hub administrator. Before installing an application, the administrator can upload the flow graph of the app to the system, and check if it is compliant with the privacy policy. If not, the installation aborts, otherwise, the app package is deployed into the system.

The configuration manager maintains the state of all the installed apps, offers a management interface to the hub administrator, and supervises the apps' execution lifecycle. When a home app is installed on the hub, the configuration manager instantiates element objects on the kernel runtime and sets up connections between instances so as to reflect the flow graph specified in the app package. Each element object can interact with a local driver which serves the specific requests of that particular element. As for untrusted elements, the runtime kernel runs the respective app code inside individual sandboxes. The sandboxing mechanism prevents the use of shared memory and thus leaking information across elements.

Elements and drivers together implement the app functionality by firing events and routing them internally through the event bus. Figure 3.2 illustrates how this works for the simple security monitoring app which collects camera frames from the living room camera and stores them at a user-specified cloud storage, e.g. Dropbox. The driver of the `IPCamera` element is configured to read a new frame (one frame per second) from the living room camera, and forwards that frame to the app's untrusted `AppCode` element instance that prepares the payload for the web call and sends it further to the `HttpRequest` element. This payload is eventually forwarded to a driver responsible for the cloud call operation.

The HomePad hub architecture is general, allowing for future extensions with new element / drivers. These can be automatically fetched and instantiated by the extensions manager when new devices are added by the hub administrator, based on the device type or even an exact device model. Considering that there can be multiple device drivers available, the users may choose the desired one based on the driver's overall rating or the element developer's authority. Finally, the element dashboard provides an overview of all the elements / drivers installed on the hub. Through it the hub administrator can check elements' state, operation statistics and error logs (if any).

Additionally, HomePad architecture was carefully designed to avoid vendor lock-in scenarios. For this purpose, HomePad depends on a common runtime environment which can then be extended with API extensions contributed by the open-source community. Smart home service providers can build home apps to run on the hub, but enjoy no special privileges regarding the software property or access rights of the hub platform itself. Furthermore, HomePad is independent of the hub hardware and can be deployed on any home server or personal computer.

At the same time, HomePad is not necessarily compatible with existing IoT platforms. Nevertheless, we envision that these platforms can be easily integrated with our hub device by exposing REST APIs accessible to the hub. It is also not our goal to be fully compatible with existing IoT devices. Nevertheless, we assume that the IoT devices managed by HomePad have a public interface that allows for the communication between them and the hub.

### 3.2.2 App development

HomePad follows a dataflow programming model and offers a collection of trusted API elements provides essential functionality for app developers to build HomePad apps. There are elements allowing apps to interact with sensors and actuators, communicate with remote endpoints, perform computations on sensor data and/or transform it by encrypting or anonymizing it, and regulate app activity based on time of the day or a pre-defined schedule.

HomePad offers a simple programming interface for writing apps which is based upon a domain specific language (DSL) implemented in the Java programming language. Essentially, to implement an app, the developer must declare the element instances and element connectors of the app's flow graph. The trusted elements are instantiated based on built-in classes or extensions to the HomePad API which must be imported into the program. Untrusted elements must be written by the developer as independent classes with declared ports and input port handlers. From our experience, the effort of writing HomePad apps is comparable to the effort of writing apps for the popular smart home frameworks, e.g. Samsung's SmartThings, whose API is also based on a DSL developed in Groovy.

While HomePad apps usually contain at least one untrusted element, it is possible to build apps consisting of trusted elements only. This is a case, for instance, for the apps that follow a '*trigger-action*' model and change the state of one sensor in response to the state change of the other. Such apps' graphs are simple enough to be implemented with just a few trusted elements connected directly. Naturally, this kind of *'if-this-then-that'* apps could be potentially created by the users themselves through a visual interface similar to the one provided by the IFTTT [17] web service.

### 3.2.3 Hub configurations

The previous section described how to implement a single app as a single flow graph. In order to host multiple applications, HomePad must not only allow multiple independent flow graphs to coexist, one for each app, but also allow apps to interact with each other and with global system services. Furthermore, it is necessary to ensure that apps cannot interfere with each other, e.g., by modifying each other's flow graphs. Moreover, HomePad must enforce strict compliance with the homeowner's privacy preferences when installing apps.

To address these requirements, first, we extend the notion of flow graph to comprise not just a single app but the entire *hub configuration*. The hub configuration is represented by a fully connected flow graph that can be decomposed into two types of subgraphs: *system subgraphs* and *app subgraphs*. The former implement system-wide functions (e.g., event bus), and the latter represent installed apps. Second, installing (or removing) an app consists of patching the hub configuration so as to connect to it (or disconnect from it) the respective app subgraph. To ensure correct behavior, the connection of an app subgraph cannot be performed arbitrarily, but requires linking specific elements of both app subgraph and system subgraphs. Third, for security reasons, HomePad assigns principal IDs to subgraphs and defines *connection permissions* to restrict modifications to the structure of subgraphs (e.g., to prevent the installation of an app from tampering with the flow graph of another app). HomePad assigns the principal ID 0 to the system subgraph, and a new principal ID ($>$0) to each app subgraph. As a general protection rule, HomePad does not allow to interconnect elements of subgraphs with different IDs. However, connection permissions can override this rule.

## 3.3 Implementation

HomePad can be run on any dedicated Linux-based computer. We implemented home apps' untrusted element sandboxes using Java Security Managers to restrict access to network and underlying file system. As for the System Drivers, e.g., IPCamera driver, we used custom Python scripts to interface low level communication between devices and HomePad's system drivers wrapping Java classes. Sensor data is received by system drivers and forwarded to element drivers, which then serve it to apps' element instances through the event bus (see Figure 3.2). This dataflow is event-based and is fully implemented in Java.

To simulate device communication, we used Arduino Yun boards and implemented simple device drivers in C++ and Python, to interact with the HomePad hub. These boards communicate over Wi-Fi through AES-256 secure channels. To support application scenarios with different sensors, e.g., cameras, microphones, we established simple APIs to facilitate the management of these boards via the HomePad hub. To simulate video and audio sensors the boards were equipped with a Sony USB webcam and electret microphones.

At install time the Model Checker analyzes the app's DSL code in order to validate the app's privacy properties. This validation involves the generation of the app's corresponding Prolog model followed by a set of Prolog queries. The Model Checker component was implemented as a Java class with SWI-Prolog version 6.6.6 engine stubs. To provide the user with a visual representation of the app's structure and privacy properties, we implemented an HTML report generator using the Graphviz tool. This report shows the results of dataflow analysis from the Prolog queries. In order for users to specify their own privacy policies we developed a simple Android app offering a simple API that allows users to pick data sources and sinks, as well as exception rules such as time constraints or data modes (e.g., encrypted, anonymized). The app then sends the Prolog rules to Homepad through an HTTPS connection.

The source code of HomePad and all of its components was made public and freely available [1] under the Apache License 2.0.

## 3.4 Evaluation

We evaluated HomePad on three fronts: first, we evaluated its runtime performance, then we analyzed the programming effort required to develop HomePad apps, and, finally, we examined HomePad's app verification effectiveness. We also evaluated HomePad's privacy policy specification mechanism and its ability to model and express the variety of users' privacy concerns.

### 3.4.1 Use-case applications

To demonstrate the variety of apps supported by HomePad, we developed four apps using technologies and devices available today in the smart home environment. Some apps rely on open-source software, i.e., Kaldi ASR [18] for voice recognition and OpenFace [30] for face recognition.

1. LightsControl app - voice-activated lights control. Implemented using API provided by Philips Hue smart lighting system [180].

2. FaceDoor app - face recognition-based door control: opens the door lock automatically for authorized users by recognizing their face with a doorbell camera. The app also notifies the homeowner through an sms or push message when known users arrive home. Implemented with custom device drivers.

3. TidePooler app - voice-activated tide information service that performs text-to-speech conversion when informing the user about the tide level

[1] https://github.com/zavalyshyn/homepad

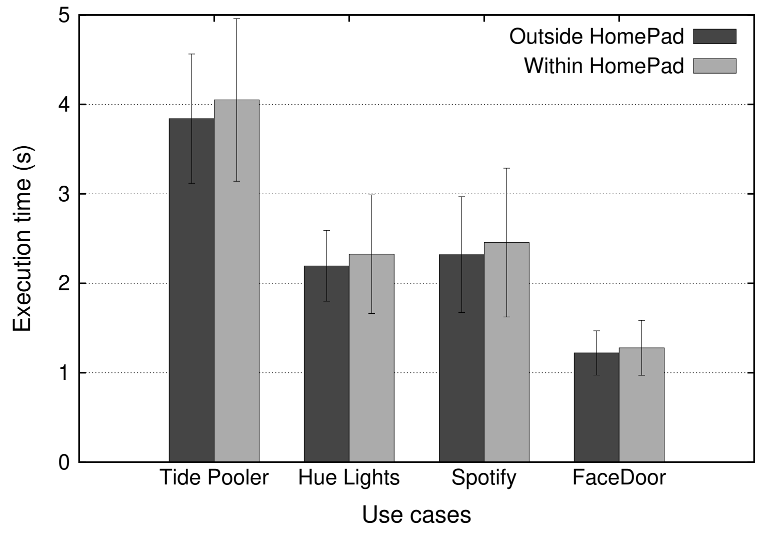


Figure 3.3 – Use case apps' runtime performace.

in a specific location. This app was ported from the Amazon Echo [6] skills collection.

4. SpotifyControl app - voice-activated Spotify player control.

The privacy risks associated with these apps come from the way they interact with the user. Voice and face recognition requires a constant access to the camera or microphone feed which is a source of sensitive information and may be used without the user's knowledge.

### 3.4.2 Performance evaluation

To evaluate the performance of HomePad, we adapted the four home automation apps described above to run under two different configurations: on HomePad and as standalone Java apps. This setup allows us to compare the performance overhead introduced by HomePad. To test the execution of these apps and measure their performance, we specified voice commands and pictures as inputs, according to each use case. The values presented reflect the average of 40 tests per app (i.e., 20 running inside + 20 running outside HomePad).

Figure 3.3 plots the execution time of our use-case applications when executed on HomePad (light grey) and on standalone mode (dark grey). HomePad

| **Execution** | **Lights Controller** | | | **Spotify Controller** | | | **Tide Pooler** | | | **FaceDoor** | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **Out** | **In** | **Over** | **Out** | **In** | **Over** | **Out** | **In** | **Over** | **Out** | **In** | **Over** |
| **Recognition** | 2.2s | 2.3s | 5.8% | 2.3s | 2.4s | 5.6% | 2.4s | 2.5s | 4.1% | 1.07s | 1.11s | 4.7% |
| **Actuators** | 34ms | 37ms | 7.7% | 0.7ms | 1.1ms | 63% | 0.7ms | 0.9ms | 32% | 30ms | 35ms | 15% |
| **Network** | – | – | – | – | – | – | 1.4s | 1.5s | 2.6% | 117ms | 119ms | 1.8% |
| **Core** | – | 3.6ms | – | – | 3.5ms | – | – | 8.8ms | – | – | 5.7ms | – |
| **Total** | 2.2s | 2.3s | 6% | 2.3s | 2.5s | 5.7% | 3.8s | 4.1s | 5.4% | 1.2s | 1.3s | 4.7% |

Table 3.1 – Use case apps' execution times breakdown.
**Legend:** *Out* and *In* define execution time obtained by running apps outside of or inside HomePad respectively; *Over* shows HomePad's execution overhead.

introduces an overhead which varies between 4.7% and 6%. This overhead is caused by the sandboxes implemented by HomePad. From our experience, considering that the total execution time varies between 2.2 and 4.1 seconds, these overheads do not significantly hinder the user's experience.

To better understand the factors that contribute to the overall performance of each app, Table 3.1 displays the total app execution time broken down into: *recognition*, *actuators*, *network*, and *core*. Recognition time is associated with the execution of voice or face recognition and it measures the overhead of running these algorithms following our privacy preserving sandboxing approach. Actuators comprise the time spent on commands to turn on lights, play the next track on Spotify, output tide information as an audio stream and unlock a door. Network time involves the communication with the outer world, whether to fetch tide information, or notify a user someone just entered his home. Core refers to the time spent on the event based communication within HomePad.

Most of the execution time of these apps is spent on voice and face recognition (between 61% and 99%), which constitute the most CPU-intensive tasks. On the other hand, the time spent on actuators represents a very small percentage, never bigger than 3% or longer than 40ms. The network communication accounts for 9% of FaceDoor's total execution time, as it only features an API call to a push messaging service. In Tide Pooler, networking cost amounts to 38% taking on average 1.5 seconds (due to the download and parsing of a large file containing tide information). Processing and routing internal messages within the HomePad core takes around 1% of the execution time.

The overall total execution time mainly depends on the app activity and remains consistent when running inside or outside HomePad. We see that for all the use case apps the total overhead of running apps inside HomePad is insignificant reaching 6% at most.

### 3.4.3 Application programming effort

To assess the programming effort needed to built a HomePad app we took our most complex use case application – Tide Pooler – and implemented it as two independent standalone apps using the speech recognition APIs provided by Amazon Alexa [5] and Google Speech [22] platforms. A version of the app using Amazon Alexa was implemented as an Alexa Skill [4] which leverages Amazon's backend to perform voice recognition and provides compatibility with Amazon's Echo device [6]. The second version was implemented using Google Speech API which provides a cloud-based voice recognition service that our Tide Pooler port uses when running as a Java desktop application. Keep in mind that our baseline HomePad implementation of Tide Pooler uses HomePad's native voice recognition system module based on Kaldi [18]. We assess the development effort in terms of the number of lines of code (LOC).

From our experience, we found that the development effort of implementing Tide Pooler across these platforms is quite comparable, requiring 331 LOC for Amazon Skill, 332 LOC for Google API, and 370 LOC for HomePad. In all cases, 35 LOC relate directly to the use case logic, 15 LOC relate to getting tide information from a server, and 250 LOC correspond to parsing the json file returned from the server. The remaining lines of code are specific to the API of each platform. In HomePad, specifically, 70 LOC are associated with adaptation to HomePad's module-element architecture.

### 3.4.4 Detection of privacy violations

To evaluate whether HomePad is able to detect policy violations by a malicious app, we altered the flow graphs of two use case apps, namely TidePooler and FaceDoor, in a way that allows to collect raw sensor data from camera or microphone and leak it to the cloud endpoint controlled by the attacker without the user's knowledge.

Figures 3.4 and 3.5 show the flow graphs of original (benign) and malicious versions of TidePooler and FaceDoor apps respectively. In both cases, we

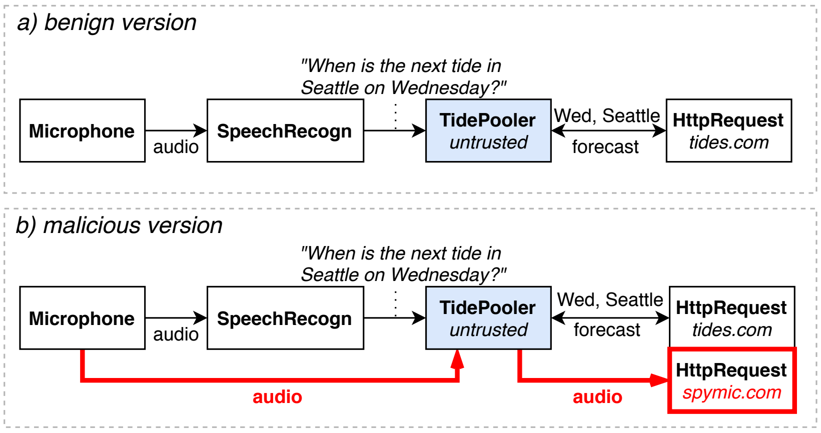


Figure 3.4 – Flow graphs of benign (a) and malicious (b) TidePooler versions.

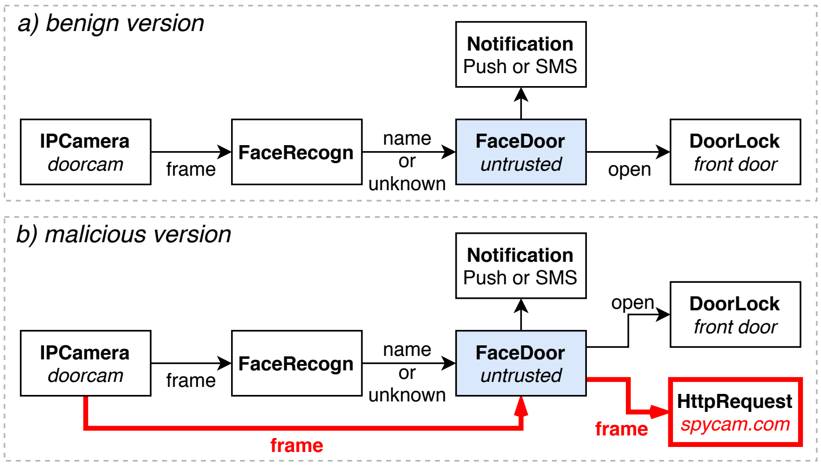


Figure 3.5 – Flow graphs of benign (a) and malicious (b) FaceDoor versions.

introduced an additional `HttpRequest` element configured to connect to a malicious endpoint, and a direct connection from the camera or microphone to the untrusted element.

Using the dataflow policy specification language we defined and activated at the hub the following privacy policy:

**block** Frame **from** IPCamera **to** Internet.

**block** Audio **from** Microphone **to** Internet.

| **Device** | **Description** | **Privacy Concern** | **Privacy Policy** |
|---|---|---|---|
| **Echo Dot** | Interactive voice assistant that records and responds to user commands prepended with an "Alexa" wake word. | *Can the device record the conversations even when the wake word was not pronounced?* | **allow** Audio **from** Echo **to** EchoAPI, **transformation**(wakeworddetection). |
| **Nest Cam** | Video surveillance and motion detection with cloud backup. | *Is the camera active when the owner is at home?* | **block** Frame **from** NestCam **to** NestAPI, **except**(**at**(20,8)) |
| **Barbie Doll** | Interactive doll that records and responds to children's questions. | *What can a toy say to the child?* | **allow** Audio **from** BarbieAPI **to** BarbieDoll, **transformation**(wordfilter). |

Table 3.2 – Privacy policy specification and translation.

This policy instructs HomePad to block any flow of raw audio or image data going to the Internet from microphone or IP cameras. We then executed the HomePad checker for each malicious version of the apps. In both cases, HomePad has detected privacy violations and correctly identified and blocked the malicious data flows.

### 3.4.5 Flexibility of privacy policies

HomePad benefits from the dataflow model's flexible privacy policy specification language. To demonstrate this flexibility, Table 3.2 presents three real-life use-case scenarios inspired by the recent reports on smart home user privacy concerns [20, 213]. In all of these use-cases the end users would benefit from HomePad's rich privacy policy specification and enforcement features.

The first example covers major concerns regarding always-on voice assistants. Users increasingly worry that devices like Amazon Echo can silently record and analyze their conversations [15, 12, 14, 23, 16]. Such concerns can be expressed in a HomePad privacy policy with a rule requiring the wake word detection before delivering the audio recording to remote service providers. This policy can be then enforced at runtime using a particular trusted element.

The second example illustrates a common concern regarding home cameras that have Internet connectivity. Users are essentially worried that their cameras are active when they are not supposed to thus violating user privacy [73, 1, 24, 13]. In this particular case, the user wishes for his bedroom's camera to be inactive from 8 PM to 8 AM. Within HomePad this restriction can be enforced using an exception rule configured to a desired time.

The last example expresses common concerns over smart interactive toys. Parents worry that such toys might leave children vulnerable to stealthy advertising or offensive content [106, 11, 25]. In HomePad a privacy policy can enforce word filtering on the data the app wishes to send to a toy's speaker. Similarly to the first example, this policy leverages a special trusted element.

## 3.5 Discussion

In this section we provide a brief security analysis of HomePad and describe its current limitations.

### 3.5.1 Security discussion

There are several attacks that a malicious home app may try to launch. If we assume that the hub system and installed elements software is correct, an attacker (i.e., a malicious app developer) may try to deploy malicious untrusted element code undeclared in the app manifest in an attempt to execute it on the hub. This attack, however, is prevented by HomePad, which only allows the execution of elements that were explicitly declared in the app's manifest file. Non-declared elements will not be instantiated and executed at the hub. Alternatively, all the untrusted elements that were properly declared, will be executed inside a sandbox with restricted access to sensor data and network.

An attacker may attempt to craft the flow graph in the app manifest, e.g., adding concealed connections between elements in order to bypass sensitive data to a data sink, or adding a large number of connections and elements in order to increase the complexity of the graph and obfuscate the flow of data. Such attacks can also be thwarted by HomePad, because it fashions a complete model of the flow graph which captures all elements and connections which can, therefore, be detected by the Prolog checker. A malicious app may also attempt to modify existing connections or elements of the system subgraph of the hub configuration. However, HomePad mitigates such attacks by assigning unprivileged IDs and constraining the app flow graph to be patched on properly authorized system elements.

A malicious home app may try to exfiltrate information through implicit flows, e.g., by omitting or issuing a call to the `HttpRequest` element and consequently to a remote web host. Sensitive sensor data may not be even present in the payload of such a request; instead, just a fact of making an

HTTP request may signal a certain boolean event, be it a motion at home or a door lock state. In general, such implicit data flows are difficult to detect and prevent. While HomePad effectively makes the data flows within a given app explicit, implicit flows may not be necessarily detected since they do not directly carry sensor data from a given source to a given sink. HomePad may only predict the existence of such implicit flows given certain characteristics of the app flow graph, e.g. untrusted elements having access to both raw sensor data and network resources, and point it out to the end users. They may then decide whether or not to proceed with the installation of a given app.

A limitation of HomePad is that its privacy verification depends on the correctness of both the output rules of elements and the rules of privacy policies. If errors exist in rules, the flow graph will no longer reflect the app's implementation logic which may result in undetected breaches. This problem is alleviated by the fact that the Prolog rules of elements are usually simple and relatively easy to analyze.

An additional limitation comes from the conservative approach used for data flow verification of untrusted elements. By default, HomePad assumes the output of untrusted elements to be the same as their inputs. Such a strict approach was selected in order to safeguard the user's privacy, even if it means to incur some false positives. Nevertheless, it is possible to refine the verification granularity, for instance, by using dynamic taint-tracking within untrusted elements to verify the input/output data types.

### 3.5.2 Operational considerations

A potential concern is that it might be complicated to manage HomePad hub for people with no computing background, especially to create the privacy policies. Moreover, the privacy policies can also grow in complexity depending on the number of installed apps. Creating and managing complex policies may cause the users to experience *decision fatigue*, a state in which a user gets overwhelmed by options and acts recklessly [95, 102, 94]. To maintain the privacy policies more manageable, HomePad includes pre-defined rules that can be used as is according to the profile of the user and the smart home devices he or she owns. These built-in rules contain best-practice privacy policies as recommended by industry experts or other tech-savvy HomePad users.

Another concern is related to HomePad's backward compatibility with existing smart home systems. However, we argue that the market pressure for enhanced privacy and data protection may well justify a departure from existing IoT models in favor of alternative secure-by-design IoT platforms, such as HomePad. Nevertheless, we plan to investigate in the future whether it would be possible to automatically (or semi-automatically) extract the dataflow model from existing platforms applications, e.g. Samsung SmartThings, so as to enable developers and smart home owners to reuse existing apps in HomePad.

## 3.6 Related work

There is a large body of work addressing home automation and IoT-related issues, such as the privacy of sensor-generated data. Allard et al. [28] combines the security of smart cards and the storage capacity of NAND Flash chips to take the control of personal data away from cloud providers back to the users. Centralized approaches have been proposed to address user personal data storage access and management [69, 161, 32]. However, all these contributions do not address the issue of apps sharing sensitive data in their possession. At the same time, Privacy Capsules [124] processes raw sensor data only inside sealed containers without network access. While Privacy Capsules limit access to the network, HomePad allows the users to decide if an app may access data and network resources dynamically.

Some recent works address these privacy issues from a network perspective. Davies et al. [78] propose the deployment of cloudlets to run applications and manage their access to raw sensor data. Yu et al. [225] suggested using routers to secure IoT devices by running micro network-security functions, acting as security gateways for each device. However, in both cases it is assumed the apps and functions are trusted respectively.

Fernandes et al. [135, 96] as well as Tian et al. [210] identified and addressed the problems of over-privileged apps in a popular smart home platform. However, all of these systems focus mainly on security implications of over-privileged apps and assume access to their source code. In ProvThings, Wang et al. [218] perform IoT platform log analysis to detect malicious device actions. They, however, assume the smart home cloud platform execution environment to be trusted, which is at odds to HomePad assumptions.

A decentralized trigger-action smart home platform DTAP was proposed in [98]. It protects OAuth tokens needed to control and manage IoT devices from being abused and shared with third parties. While DTAP renders compromised OAuth tokens useless, it does not allow to track and control the flow of user data to legit token holders. In contrast, HomePad allows to do so for any third party involved.

There are several contributions that although not directly related could complement our work. In the context of smart homes, HomeOS [86] simplifies the management and interoperability of various home environment technologies. Xapp [62] facilitates resource sharing among Android apps distributed on different home devices. All these contributions focus on managing the heterogeneity of the devices in smart home environment.

There are several systems that perform information flow analysis through taint tracking [48, 193, 222], but also leveraging static code analysis [167, 36, 219]. This approach has also been used in mobile contexts [221, 121, 91]. These systems, however, do not address the smart home environment and its complex interaction model.

Flowfence [97] uses information flow control to manage sensor data accesses from applications. The limitation of Flowfence is in its inability to dynamically modify the taint labels depending on the type of the data flow. For instance, if an app reads the video stream from IP camera and applies the face blur filter to the output stream, the taint label of the resulting data should be changed. HomePad implements such functionality by utilizing trusted modules that perform data filtering and obfuscation dynamically. On the other hand, FlowFence offers no way to automatically verify the privacy properties of an application against users' preferences, resorting instead to a pure runtime mechanism, incurring in considerable performance overhead.

## 3.7 Summary

In this chapter we presented HomePad a privacy-aware home hub that allows users to supervise how the data generated by their smart devices is processed and used by home applications. HomePad applications follow an element-based programming approach, which makes all the data flows between app elements explicit and subject to inspection. By laying out applications in this fashion, HomePad can automatically leverage its Prolog-based data flow verification mechanism in order to assess these applications' compliance with users' privacy policies. Additionally, Homepad's expressive privacy policy specification supports a broad spectrum of privacy concerns users have. By combining these two capabilities, Homepad provides runtime data control to its users.

HomePad makes a strong case for local-first data processing model in a smart home scenario. Various apps and services can benefit from such a model without necessarily sending sensitive sensor data to the remote cloud servers for further processing. All the required computational and storage resources can be provided by the HomePad hub. Processing sensor data at the edge has an additional advantage of reducing the network latency and service response time. The latter is essential for voice-activated applications that try to minimize the time between the user command and a triggered action.

While local-first approach is beneficial for both user privacy and application performance, certain smart home scenarios require sending sensor data to the external servers. This is a case for applications that rely on machine learning techniques to provide a given service, and need to send sensor data to the remote server with significant computational resources otherwise not available at the local hub device. Furthermore, some other applications may have integrations with various third-party services, e.g. Dropbox, as part of their legitimate logic. Controlling such external data flows is impossible with HomePad since they span beyond its security perimeter. HomePad can only block or allow these flows but may not enforce any further restrictions. In the next chapter, we describe the ways to extend data flows control to the untrusted cloud environment without sacrificing user privacy and application performance.

# Chapter 4

# PatrIoT: a private IoT platform

## 4.1 Introduction

Despite the growth and popularity of smart home devices and systems, this technology remains overshadowed by a cloud of security and privacy concerns. Today, by relying on IoT platforms like Samsung SmartThings, Amazon Alexa, or Apple HomeKit, homeowners can seamlessly control smart devices, such as smart locks, virtual assistants, or baby cams, and run third-party applications (*apps*). However, falling under the control of antagonist actors, these systems can be turned into authentic spying platforms. In fact, once installed various third-party apps can collect highly sensitive data, e.g., video, audio, or the environment sensor readings, which can be abused in harmful ways [236].

Various mitigation techniques have been proposed for verifying apps' security and safety properties [66] and improving access control mechanisms [97]. However, common across all these efforts is the assumption that IoT platform providers are to be considered fully trusted. Currently, the platform providers can fully control the IoT cloud backend and collect, store, and / or share users' sensor data. Unfortunately, such privileges have already caused serious data misuse incidents that fall under the direct responsibility of IoT platform providers, involving targeted advertisement [92], surveillance and forensic investigations [122], insider-related eavesdropping or massive data leakage [8].

We aim to revisit this assumption arguing that, in addition to malicious smart apps, platform providers themselves can be a major source of potential security and privacy breaches that have been previously overlooked. To protect against such threats, we present PatrIoT – a private-by-design IoT platform for smart home apps in which homeowners retain full control over sensor data generated by their devices. PatrIoT was designed with two goals in mind: (1) prevent any arbitrary access to sensor data by provider of the cloud server where PatrIoT is running, and (2) provide homeowners with a practical yet easy to use interface to control sensor data sharing with third party apps they install without overwhelming them with details.

To achieve the first goal, PatrIoT relies on a hardened cloud backend service that runs inside a trusted execution environment (TEE) supported by Intel SGX technology. SGX secure enclaves offer memory-isolated environments that provide confidentiality and integrity protection against untrusted privileged system processes. By processing sensor data inside SGX secure enclaves, PatrIoT can effectively restrict the data access privileges of the cloud provider. To reach the second goal, in analogy to a “firewall”, PatrIoT introduces the notion of *flowwall* which controls *how* third-party apps use the sensor data they request access to. Flowwall consists of an information flow control (IFC) monitor that enforces the global device policies specified by the users. In contrast to existing permission-based smart home systems [96], that are either too coarse-grained or require certain expertise from the users to evaluate the potential risks on a *per-app* basis, PatrIoT’s flowwall allows users to think in terms of *devices* they have and how those devices’ data may or may not be used by *any* app they install.

PatrIoT makes two central contributions involving its policy specification and enforcement mechanisms. As for policy specification, many of the existing privacy-oriented solutions have failed to provide an adequate user interface, overwhelming the users with low-level details and causing the decision fatigue [56]. To address this usability challenge, PatrIoT’s UI was designed to make the process of privacy policy specification intuitive and easy to follow for a regular user. To define a policy, users operate with familiar device names, meaningful data types, e.g. audio or video, and destinations where these data types can or cannot flow to. The policy rules are defined once and applied to all the apps installed in the future.

As for policy enforcement, it is necessary to efficiently track information flows within and across individual apps, and validate the user policy. To this end, PatrIoT relies on an element-based programming model and employs static analysis and policy validation at the API level. As in HomePad, PatrIoT apps are written in the form of a graph, where edges represent data flow paths, and the nodes functions provided by the API or by the developer. From the graphs of installed apps, PatrIoT generates a global and sound data flow model using first-order logic predicates to check for policy violations.

We built a prototype of PatrIoT by leveraging SCONE [34], which allows us to deploy the PatrIoT backend securely in a Docker container running inside an SGX enclave. PatrIoT provides a JavaScript API for app developers and runs on top of Node.js. We use Prolog predicates to generate and check the apps' data flow models.

We evaluated PatrIoT across multiple dimensions. Performance wise, we observed that, despite some considerable overheads introduced by the SGX technology, a single PatrIoT server can sustain the traffic generated by a typical-sized household. By emulating a realistic deployment scenario populated by 10 different smart devices, and by implementing 20 different smart apps, we were able to express a range of different policies, and validate that PatrIoT can block or allow the data flows generated by these apps, thus demonstrating the expressiveness and effectiveness of PatrIoT's policies. Lastly, to assess the usability and relevance of our system, we performed a field study involving 45 participants. We found that a majority of participants considered PatrIoT to be easy to use, and its policy rules to be useful in protecting their privacy.

## 4.2 Design

We strongly believe that any IoT platform must be private-by-design. Private-by-design means that the platform is implemented and functions in a way that prevents any sensitive sensor data collection, processing and sharing without user awareness and approval. The platform must not only detect and block any attempts to circumvent this requirement, but also provide a proof that it is capable to do so. In the next sections, we present a system and security models of such a platform, and then describe our design goals and a threat model.

### 4.2.1 System model

The proposed system model is presented in Figure 4.1. Its central component is the TEE-protected Smart App Runtime (TSAR). It consists of a software stack which runs on a cloud infrastructure and provides the basic backend services for managing smart devices and hosting apps. With a management mobile app, a homeowner (user) can securely interact with the TSAR service in order

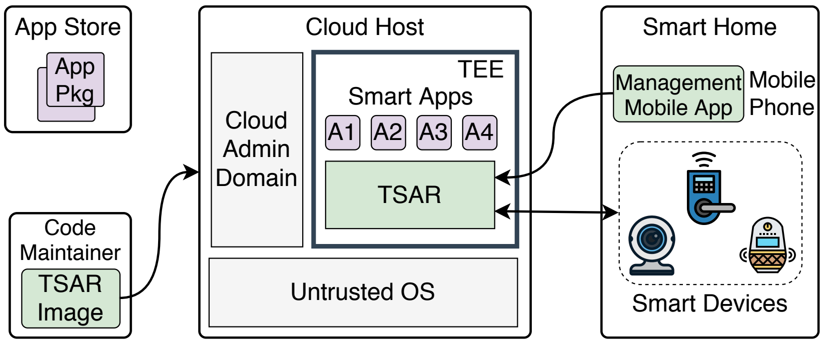


Figure 4.1 – System model of a private-by-design IoT platform.

to manage his smart devices, and install and configure apps downloaded from an app store. Once installed, these apps run inside sandboxes, and can access sensor data based on permissions and a global user-defined security policy.

In contrast to existing IoT platforms, the TSAR service is hardened in such a way that an IoT cloud administrator does not have any access privileges over the users' sensor data. This is achieved with two techniques: (i) by restricting the TSAR service external interfaces so that only the management app or smart devices are able to connect to it through TLS channels, and (ii) by running it inside a TEE so as to prevent privileged OS processes from accessing the TSAR memory where sensor data resides. A TEE is provided by a dedicated hardware such as Intel SGX. Thus, a home user will only need to trust in the implementation of the TSAR software, and acquire a proof of its secure deployment in a cloud so as to obtain exclusive access rights.

To build this level of trust, we envision a model where the TSAR software is maintained by a trustworthy *code maintainer*, which can be a single reputable entity or a consortium, and released open source to help detect potential code vulnerabilities. It can be shipped in the form of a container or VM image ready to be deployed on general-purpose cloud with SGX support (e.g., Microsoft Azure's ACC), or be offered as a service by cloud providers to all security-conscious smart home users (e.g., on a pay-per-use model).

PatrIoT offers a clean slate IoT platform design which is not necessarily compatible with existing devices, apps and platforms. While disruptive in its nature, we argue there are strong economic incentives in favor of PatrIoT's adoption. First, there is a huge demand for privacy-preserving solutions among consumers and think tanks [90]. Second, there is an increasing pressure from lawmakers for stricter data protection measures (e.g., GDPR in EU). Third, the smart home market is still very fragmented and lacking standards; as such, PatrIoT can make an important contribution to the consolidation of privacy-enhancing techniques for smart homes.

### 4.2.2 Security model

Existing IoT platforms such as Samsung SmartThings rely on a discretionary access control model where each app requests permissions to access a given resource (e.g., a sensor reading). Once granted, however, permissions alone fail to control *how* resources will be used by an app, and are difficult to manage as the number of devices and apps grows. To overcome these limitations, the TSAR service incorporates not only a permission-based model, but also a new security abstraction named *flowwall*.

A flowwall implements an IFC-based security monitor that allows users to: (i) reason about global data flows generated by devices rather than concentrating on individual apps, and (ii) block privacy-sensitive flows without overwhelming them with details. It supports three intuitive data flow patterns:

- **S2S: Smart Device → App → Smart Device:** These are internal flows within home, e.g., app reads the status of a presence sensor to detect someone's arrival, and turns on a smart light.
- **S2M: Smart Device → App → Mobile Phone:** Flows from a smart device to the user's mobile phone, e.g., app that streams a video feed from a front door IP camera to the user's phone.
- **S2W: Smart Device → App → Web:** These are some of the most sensitive flows, where sensor data is sent to Internet, e.g., an app sends motion event to a remote website.

To characterize such flows and to easily specify policies for blocking or allowing them, the flowwall is based on several concepts that Figure 4.2 helps to introduce. This figure shows an example of a home scenario where four smart apps are installed ($A_1$-$A_4$): a security surveillance app WatchMyHouse, a voice-activated WillItRain app for weather forecast check, a LightMyPath app for motion-triggered lights control, and a PhotoBurst app which notifies the user with the camera photo when motion or contact event is registered.To perform their functions, smart apps may request access to certain objects named

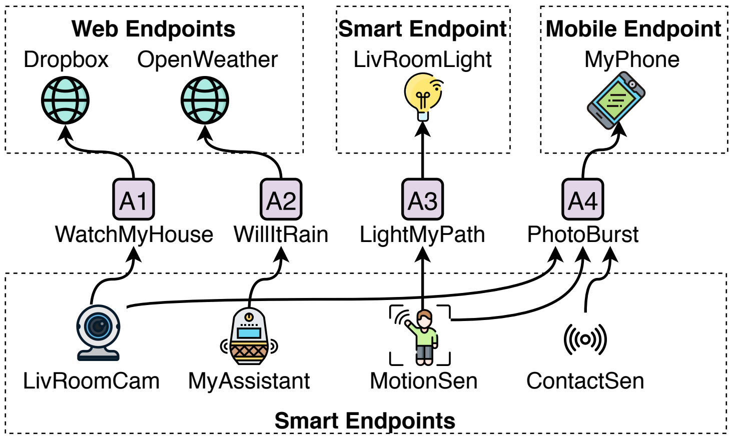


Figure 4.2 – Data flows in a smart home scenario with four installed apps.

*endpoints*. Endpoints represent system resources that can act as producers (i.e. data *sources*) or as consumers (i.e. data *sinks*) of sensor data.

Data flows are represented by the arrows shown in Figure 4.2. For instance smart app $A_1$ reads frames from the user's camera located in the living room (LivRoomCam) and uploads them to a user's Dropbox account, generating a Image data flow between these two endpoints. Collectively, apps $A_1$-$A_4$ illustrate all three data flow patterns, i.e., S2S, S2M, and S2W. The flowwall will i) keep track of all possible apps' data flows, and ii) allow or block specific flows according to the rules specified in a *security policy*. For instance, by blocking all flows from the living room's camera (LivRoomCam), apps $A_1$ and $A_4$ would necessarily be blocked. Next, we clarify our requirements to build an IoT system based on a flowwall security monitor.

### 4.2.3 Goals and threat model

To build a private-by-design IoT system as described above, we have three additional requirements: 1) the security policies must be easy to specify, 2) the system should perform well despite the introduction of new security mechanisms, and 3) the system should provide a developer-friendly API for writing apps. Note, however, that it is not our goal to preserve compatibility with existing IoT platforms or legacy apps. Likewise, some existing smart devices may not work off-the-shelf with our system. We redesign the IoT platform in the interest of improved security properties.

Our system must be secure against: (i) untrusted smart apps, which may attempt to use the API to circumvent the user-defined security policies, e.g., read sensitive data from a sensor and send it to an unauthorized party; (ii) network attacks, that aim to intercept the communications between the system components, e.g., to launch MITM attacks; and (iii) cloud server admins, with remote root-privileges, who may attempt to access or interfere with the volatile or persistent state of the TSAR container to extract sensitive sensor data. Note, we assume that these parties may not collude.

We assume that several components are trusted: the PatrIoT's TSAR service and a management app, the IoT devices firmware, the software components of our IoT platform, the cryptographic primitives adopted for the implementation of security protocols, and the underlying hardware infrastructure used by the cloud provider. In particular we assume that the cloud hosts are equipped with trusted hardware technology, namely Intel SGX, which we assume to be correct. The mobile device running the management app is trusted. Physical attacks and microarchitectural side-channel attacks are considered to be out of scope in this study.

Next, we present PatrIoT – a system that provides a private-by-design IoT platform – by focusing on its relevant design details.

### 4.2.4 TEE-protected smart app runtime

The core of our system is the PatrIoT TSAR service (see Figure 4.3). It was built by leveraging SCONE [34], which offers a secure Docker container execution environment on top of SGX-enabled CPUs and protects the container processes from external attackers. It implements a Library OS with a small trusted computing base. The TSAR service is provided by a containerized process that runs a Node.js binary cross-compiled against the SCONE libraries, and with a native Prolog engine add-on that is used for checking flowwall policies. Node.js then runs the PatrIoT TSAR-specific components, which are written in JavaScript.

The *runtime manager* is the heart of the TSAR service. It manages smart devices, apps, and user configurations for a given home environment. In particular, it controls the life cycle of apps and maintains their execution contexts. Apps interact with the environment through an API, which leverages an internal *event bus* for interfacing with *drivers*. There are multiple drivers responsible for interacting with smart, mobile and web endpoints, and for of-

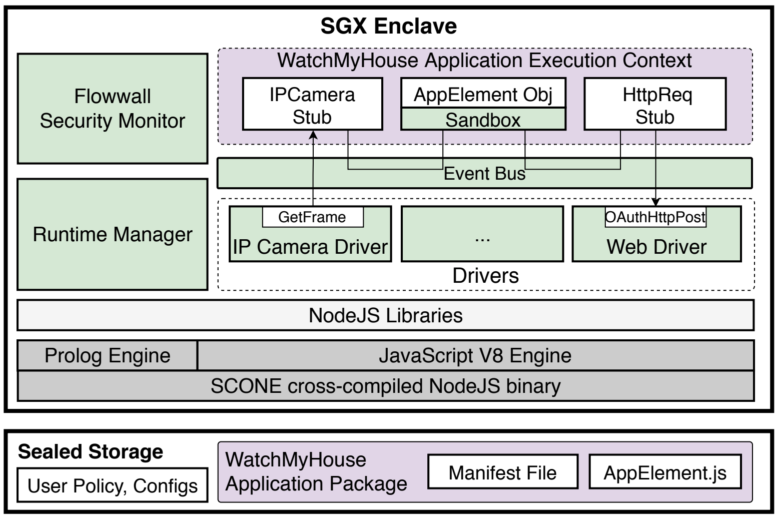


Figure 4.3 – Components of PatrIoT TSAR.

fering other services (e.g., timers). The flowwall *security monitor* tracks all apps' data flows and enforces a user-defined security policy. The persistent state consists of TSAR-specific files (e.g., security policy and configuration files), and app packages installed by the user. It is protected by sealed storage encryption techniques.

To obtain proof that the TSAR image has not been tampered with and runs inside a legitimate SGX-enabled CPU on a cloud host, PatrIoT implements a remote attestation protocol assisted by the SCONE Configuration and Attestation Service (CAS). The CAS allows to encrypt certain parts of the Docker container file system and decrypt them only after successful attestation (i.e., sealed storage). A newly instantiated SCONE container connects to the CAS and requests a remote attestation. The CAS validates the enclave by checking its hash value and other parameters. If the attestation succeeds, the CAS provisions the decryption key necessary to decrypt the content of the container file system. We use this feature to include a user-specific challenge inside the encrypted container file system: a TLS key and certificate. If the management app is able to connect to the TSAR service over HTTPS using said TLS certificate to authenticate the server endpoint, it means that the attestation was successful. At this point the PatrIoT backend is considered to be trusted and fully operational. Next, we explain how apps are programmed and supervised.

```
R1.   allow Everything from Anywhere to Anywhere.
R2.   block Everything from Anywhere to Internet
R3.   block Everything from Anywhere to Phone
R4.   allow Image from LivRoomCam to Dropbox
          at 12:00-14:00,Wed
R5.   allow Everything from Anywhere to MyPhone
```

Figure 4.4 – Policy example for the scenario in Figure 4.2.

### 4.2.5 PatrIoT API

As in HomePad, PatrIoT implements a dataflow programming model in which apps are represented as a graph of elements. It provides a rich library of API elements that were carefully designed not only to offer easy-to-use programming abstractions, but also to enable the implementation of a sound, meaningful, and efficient taint tracking mechanism for flowwall policy checking purposes.

At runtime (see Figure 4.3), the TSAR service creates an application execution context which consists of (i) element stubs that point to the drivers that implement the trusted elements used by the app, and (ii) stateless sandboxed instances of untrusted element code. These objects communicate through the event bus according to the paths that have been declared in the app's manifest. The flowwall security monitor oversees these flows, and decides whether or not the app is allowed to execute depending on the rules in the security policy.

### 4.2.6 Flowwall security policies

A flowwall security policy consists of a sequence of allow or block *rules* which are evaluated sequentially and applied atomically by the security monitor. The flowwall is initialized with an implicit default rule (R0) which blocks all possible flows, i.e., no app will be able to communicate unless R0 is overridden by a user-defined security policy. Next, we show how these policies are specified.

**Overview by example:** Unauthorized sensor data sharing with Internet destinations or arbitrary mobile phones may lead to potential data exfiltration. Figure 4.4 shows a simple policy that aims to whitelist the web and mobile endpoints considered to be trustworthy for the hypothetical scenario presented in Figure 4.2. It contains five rules (R1-R5) which are interpreted sequentially. The policy first overrides R0 by allowing flows of any kind to occur (R1), and then blocks all flows to the web and to mobile endpoints (R2 and R3); this allows only data flows to occur within the home environment. Next, two ex-

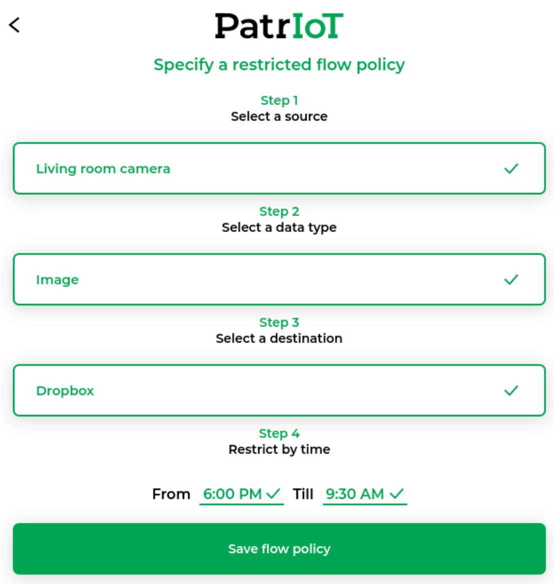


Figure 4.5 – Example of a privacy policy rule specified via PatrIoT UI.

ceptions are opened: R4 lets camera frame images to be collected from the living room's camera and uploaded to the user's Dropbox account during a certain time of the day (e.g. when the cleaning staff has access to the house), and R5 allows sensor data flows to the user's own mobile phone.

**User interface for policy specification:** PatrIoT's management app provides a simple UI for the user to specify their privacy preferences and expectations regarding data flows that occur in their smart homes and IoT setups (see Figure 4.5). The interface follows the design presented in Section 2.2.2 and requires the user to specify the data flow source, type and sink, and, finally, add optional time restrictions and/or exceptions.

Through the same interface, the end users may then view the list of all of their existing policy rules and edit them if needed. To make it easier to navigate the list, the rules may be filtered by the device or data type, creation time or by the rule activity (e.g. how often the rule was used to evaluate apps' data flows).

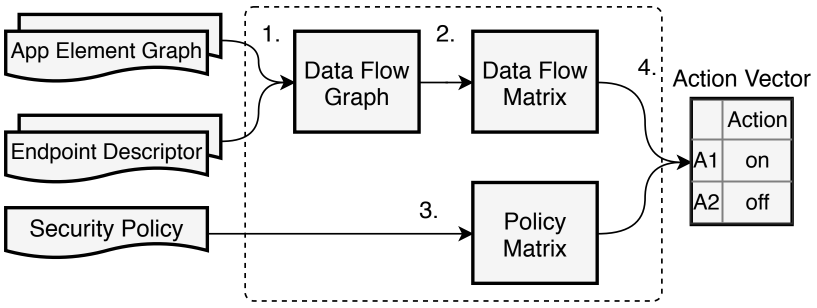


Figure 4.6 – Data structures operated by the security monitor.

### 4.2.7 Policy enforcement

In contrast to HomePad's hub controller which performs the verification of data flows within a given app at install time, PatrIoT implements a more sophisticated security monitor which performs such verification and policy enforcement anytime the policy rules are added or modified. We will now describe this mechanism in more details.

To enforce a security policy, the security monitor implements a policy evaluation algorithm which decides the execution state of every installed app based on whether or not the internal app data flows violate the policy rules. The algorithm updates an *action vector* (AV), where AV[$a$] indicates the intended execution state for app $a$: *off* means the app must be suspended, or *on* means the app can be started. Every time AV is changed, the security monitor disables or enables the apps accordingly.

**Policy evaluation algorithm:** Figure 4.6 shows the inputs, the output, and intermediate data structures generated by the policy evaluation algorithm. For inputs, it takes the element graphs of all installed apps, descriptors of existing endpoints, and the security policy. Based on these inputs, the algorithm generates two data structures which aim to model all possible data flows generated by the apps – the *data flow graph* and the *data flow matrix*; and a data structure that expresses the policy rules in an efficient manner – the *policy matrix*.

To explain how the algorithm works, consider the scenario of Figure 4.2. Assume that PatrIoT is configured with the policy shown in Figure 4.4 and that only WatchMyHouse (A1) and PhotoBurst (A2) apps are installed. To ease the explanation, we follow the algorithm along the four steps shown in Figure 4.6, assuming that the intermediate data structures are built from scratch:

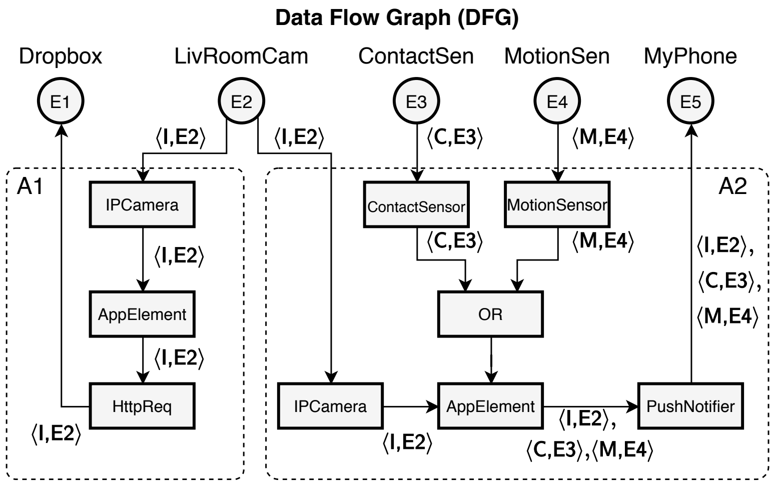
**1. Modeling of data flows:** The security monitor generates a model of all data flows that can potentially exist. This model consists of a set of Prolog predicates that specify a global *data flow graph* (DFG) based on the installed apps and existing endpoints. Figure 4.7 represents the resulting DFG for our example scenario. Nodes consist of the aggregate elements (represented as boxes) pertaining to all installed apps (A1 and A2) and the endpoints that these apps have access to (represented in circles). Directed edges connecting two nodes $n_1$ and $n_2$ indicate that data can flow from $n_1$ to $n_2$. The type of data and its provenance is indicated in the labels attached to the edge. Each label consists of a pair $\langle d, e\rangle$ which indicates the data type $d$ and its provenance $e$, i.e., $d$'s source endpoint.

If $n_1$ is an endpoint and $n_2$ is an element, it means that $n_1$ produces a data type generated by $n_1$'s respective driver and later forwarded to the element $n_2$. This is the case, for instance, of element IPCamera, which is used in the context of application A1 and reads an image from endpoint E2, i.e., the living room camera. The label associated with this edge is $\langle \mathrm{I}, \mathrm{E2}\rangle$ to indicate an image I that can be generated by E2.

If $n_1$ and $n_2$ are both elements, then the edges reflect the connections of the respective app's element graphs and the possible types of data that can be transferred through these connections. These data types are indicated by the label attached to the edge and determined by the output of element $n_1$. This output, in turn, tends to be a function of $n_1$'s inputs, but it depends on the specific functionality implemented by $n_1$. Below in this section we explain in more detail how this is performed, but assume for now that an element propagates taint from all its inputs to all its outputs, in other words, the label of each of $n_1$'s outputs results from the union of the labels of all its inputs. Thus, for instance, A1's AppElement propagates label $\langle \mathrm{I}, \mathrm{E2}\rangle$ from its input to its output, which means that HttpReq can receive image data from E2.

The last case is when $n_1$ is an element and $n_2$ is an endpoint, which means that $n_2$ is a data sink for the data types indicated in the edge's respective label. For example, HttpRequest can send to Dropbox an image originating from E2.

**2. Extraction of data flows:** The DFG model is used to determine all possible data flows between source and sink endpoints, and record that information in the form of a (sparse) data flow matrix (DFM). The resulting matrix for our example scenario is shown in Figure 4.7. Rows and columns indicate source and sink endpoints, respectively. DFM$[e_1, e_2]$ is empty if no flow exists from



**Data Flow Matrix (DFM)**

| > | E1 | E2 | E3 | E4 | E5 |
|---|---|---|---|---|---|
| **E1** | - | - | - | - | - |
| **E2** | [I,A1] | - | - | - | [I,A2] |
| **E3** | - | - | - | - | [C,A2] |
| **E4** | - | - | - | - | [M,A2] |
| **E5** | - | - | - | - | - |

**Policy Matrix (PM)**

| | Source | Sink | Datatype | Action |
|---|---|---|---|---|
| **R1** | E2, E3, E4 | E1,E5 | I, C, M | on |
| **R2** | E2, E3, E4 | E1 | I, C, M | off |
| **R3** | E2, E3, E4 | E5 | I, C, M | off |
| **R4** | E2 | E1 | I | off |
| **R5** | E2, E3, E4 | E5 | I, C, M | on |

Figure 4.7 – Intermediate data structures for policy evaluation.

The DFM shows that the images from the user's living room camera (E2) can be sent to Dropbox via A1 or to his mobile phone via A2, and that all possible destinations of motion and contact sensor readings are limited to his mobile phone only via A2. The PM shows that the action value for R4 is *off*: this means that this PM version is covering a time span where that targeted flow is not allowed, i.e., outside the 12h-14h time slot on Wednesdays.

$e_1$ to $e_2$; otherwise, it contains a list of pairs $\langle d, a \rangle$ which indicate the data type $d$ that can flow between them and identify the app $a$ responsible for that flow. To build this matrix, the security monitor executes a DFG Prolog query which computes the labels of the ingress edges of every sink $e_2$. From these labels, $d$ and $e_1$ are extracted; from the element linked to $e_2$, the app $a$ is identified.

**3. Expansion of the policy rules:** Before the final stage of policy evaluation, it is necessary to create an adequate representation of the security policy that allows to match the policy rules against the data flows described in the DFM. In particular, it is necessary to properly parse the references to groups of endpoints (e.g., Anywhere) and take into account the temporal restrictions in the rules (if any). This is the role of the Policy Matrix (PM) shown in Figure 4.7.

**4. Policy evaluation and AV update:** The last stage of the policy evaluation algorithm is to match the rules of the PM against the data flows described in the DFM and produce an action vector (AV) that tells which apps must be suspended or resumed. For each rule $r_i$, the algorithm obtains all the source-sink endpoint pairs $(e_1, e_2)_{r_i}$ and uses them to index the data flow table at position DFM$[e_1, e_2]$ and look up its value. If it is empty, no flow exists that matches the rule and the algorithm continues. Otherwise, DFM$[e_1, e_2]$ contains pairs $\langle d, a\rangle$ that tell the data type ($d$) of the matched flow and the identity of the app ($a$) responsible for it. Next, the algorithm only needs to check if $d$ corresponds to the data type indicated in the rule to verify if there is a full match. In that case, the action vector AV is updated according to the action instructed by the rule: if action is allowed, then AV$[a]$=*on*, otherwise, action is denied, and AV$[a]$=*off*. After traversing all rules the final version of AV is [*on*, *off*], i.e., A1 will be enabled, and A2 disabled.

### 4.2.8 Data flow graph model generation

As mentioned above, the security monitor generates a DFG model that can be used for extracting the data flows between any given source and sink endpoints. As in HomePad, we use first-order logic to create this model. For any given app, the security monitor reads the app's manifest file, and creates two kinds of predicates: *topology* predicates, and *output taint propagation* (OTP) predicates expressed as Prolog statements. The former represent the app's element graph; the latter tell how each element propagates labeled inputs to its outputs.

Following the dataflow programming model, OTP predicates for trusted elements are statically defined as part of the PatrIoT API. For each trusted element of the API, along with its JavaScript implementation, there is an accompanying file containing the element's OTP predicates. For the untrusted elements a general OTP predicate is used which models all such elements as *'funnels'*, i.e., the labels from all the element's inputs will be forwarded to every single output port.

When generating the DFG, the security monitor loads the OTP predicates for trusted and untrusted app elements into the DFG model. Based on these predicates, the security monitor can model the tainted labels propagation within the app. Finally, to determine all the data flows between any given source and sink endpoints, PatrIoT uses a **flows** rule (see Section 2.3.2), which aims to detect if a certain data type can flow from endpoint $e_1$ to endpoint $e_2$.

By issuing this query to a first-order logic engine, existing solutions will be found by unifying it against the topology and OTP predicates of the DFG model. If there is a sequence of interconnected nodes that propagate a data type from $e_1$ to $e_2$, a result will be found and assigned to X. The security monitor uses this technique to fill in the data flow matrix.

## 4.3 Implementation

We implemented a full prototype of the PatrIoT system. In total, we wrote $\sim$20K lines of JavaScript code. The TSAR container was built using a Docker image featuring a Node.js v.8.9.4 binary cross-compiled against SCONE libs. Node.js includes a native add-on that implements a Prolog query engine based on SWI-Prolog v.7.7.8 which was also cross compiled against SCONE libs to enable execution within an SGX enclave. We developed in total 17 drivers responsible for the implementation of 35 trusted elements. These include drivers to interact with various smart home devices, e.g., smart lights and IP cameras, web drivers for standard HTTP connections as well as OAuth2 ones, and various sensor data processing elements, e.g. speech recognition.

The management app consists of a React-based frontend that serves a dynamic web application to connected clients. This application was designed to be used on both mobile and desktop devices. Through this application, the users can connect new devices, add new policy rules and install or delete various third-party apps.

The PatrIoT backend was implemented as a REST API server provided by the runtime manager of the TSAR service. This backend manages user configs (e.g. user credentials and privacy policy rules), connected devices and installed apps. To sandbox untrusted app elements, we rely on the VM2 [3] implementation of a VM sandbox module for Node.js. Sandboxed code cannot import external modules, nor any global variables or classes from the main PatrIoT context.

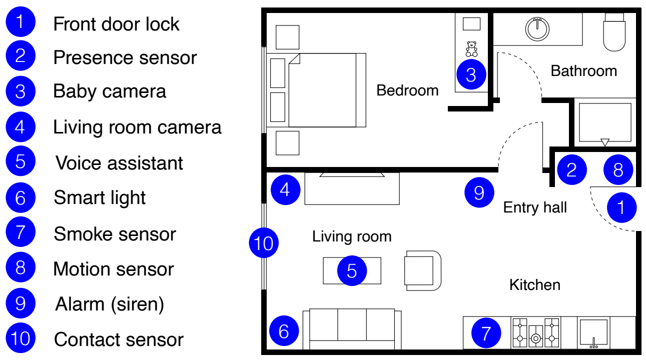


Figure 4.9 – Emulated smart home setup.

## 4.4 Evaluation

We present our evaluation of PatrIoT focusing on three main aspects: i) performance, ii) expressiveness, and iii) usability.

### 4.4.1 Case study

To evaluate our system, we recreate the smart home scenario displayed in Figure 4.9. This home belongs to a family of three: Samantha, John and their baby. A nanny comes occasionally to babysit. There is also a predefined schedule for cleaning staff to access the home.

We emulated ten devices deployed in the smart home. The front door lock was emulated using an Arduino-based contact sensor. Presence, motion, smoke sensors and an alarm were emulated using corresponding Arduino-based sensors (HC-SR501 PIR, MQ-2, piezo buzzer). IP cameras were emulated using a USB camera attached to a Raspberry Pi device and streaming an MJPEG video. The same Raspberry Pi equipped with a microphone and a speech-recognition software running on it was used to emulate a voice assistant device. Finally, we used a Philips Hue light bulb as a smart light device.

We implemented 20 PatrIoT applications for this smart home scenario. Their functionality ranges from device-to-device interaction (e.g. LightMyPath, Economie), to device-to-mobile (e.g. PhotoBurst, SmokeAlarm, BabysUp) and device-to-web (WatchMyHouse, SmartSecurity) interaction. A set of voice-activated apps can either interact with local devices (DoorCheck, LightItUp) or web services (SpotifyController, WillItRain).

### 4.4.2 Performance

To assess the performance of our system, we evaluated independently the system initialization time, the system maximum throughput, and the performance of applications.

**Experimental setup:** The system initialization time comprises three parts: attestation time, TSAR service bootstrap time, and app loading time. For the remote attestation we relied on a locally deployed SCONE CAS server running on the same machine. The attestation time includes the time needed to authenticate PatrIoT with a CAS instance, receive a session key, decrypt the PatrIoT core files, and start the TSAR service. Bootstrap and app loading times were measured separately after the remote attestation process.

We evaluated the maximum system throughput by stress-testing the TSAR service. We used the wrk2 tool running on a second machine in the same network and generating a constant throughput load. We then measured the observed latency. We set the number of concurrent connections equal to the number of devices in our case study (ten). We increased the throughput gradually until the latency started to degrade or socket connection errors appeared. We recorded the maximum throughput right before the saturation point. As a reference, we used the latest Apache2 web server.

To analyze the performance of PatrIoT apps we used a benchmark based on the use-case apps described in Section 4.4.1. We measured the time it took to execute a complete app data flow graph: from the time a trigger event was generated until the time it was fully processed by the app. We also measured the Prolog query time for each app's DFG model. This is the most time consuming step of the policy enforcement algorithm (see Section 4.2.7).

For our testbed, we used two servers running 64bit Ubuntu 18.04.4 LTS with a 16-core 3.60GHz Intel i9-9900K CPU and 16GB of RAM. We adopted the 19.03.9 version of Docker engine to run PatrIoT. PatrIoT core files inside a Docker image were encrypted using SCONE's File Shield. We evaluated the performance of PatrIoT running inside and outside of SCONE SGX separately. Obtained values were averaged across 20 runs.

**System initialization time:** Table 4.1 presents PatrIoT's attestation, bootstrap and app loading times. It takes on average 13.5 seconds to attest PatrIoT running inside an SGX SCONE enclave. Most of this time is taken by communication with a CAS server and decryption of PatrIoT core files after a successful

| Environment | Attestation time, s | Bootstrap time, ms | App loading time, ms |
|---|---|---|---|
| Inside SGX | 13.5 | 14.979 | 117.73 |
| Outside SGX | n/a | 4.258 | 1.012 |

Table 4.1 – Attestation, bootstrap, and app loading times.

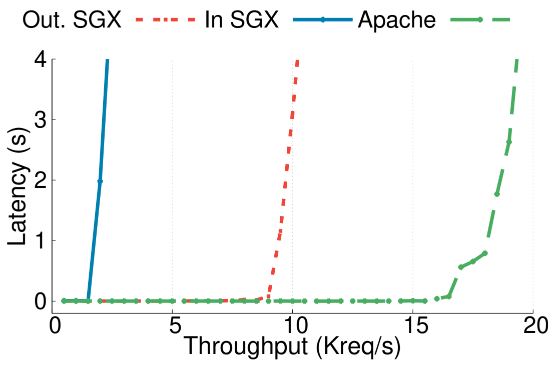


Figure 4.10 – Throughput versus latency evaluation.

attestation. Additional delay comes from the fact that SCONE needs to allocate the required memory resources at enclave start time which depending on the specified heap size might take more time. However, considering that PatrIoT is a server component which needs to be started only once and run continuously such a one-time delay can be tolerated. The bootstrap time overhead of using SGX is just 10 ms which is mostly caused by enclave transitions during system calls. The app loading time overhead reaches 118 ms, which is the time it takes to decrypt the app files in the container's encrypted file system.

**Load test:** Figure 4.10 features the results of PatrIoT server test when run inside and outside SGX SCONE enclave. PatrIoT Server performed similarly in both settings until the load reached 1900 requests per second, at which point the latency of the PatrIoT's SGX version started to degrade. The standalone version of PatrIoT reached a saturation point at around 9000 requests per second. Since many smart devices generate low-rate network traffic, this limit is acceptable. We observed nearly 5x performance loss when running PatrIoT inside an SGX SCONE enclave. This is consistent with the original reports by

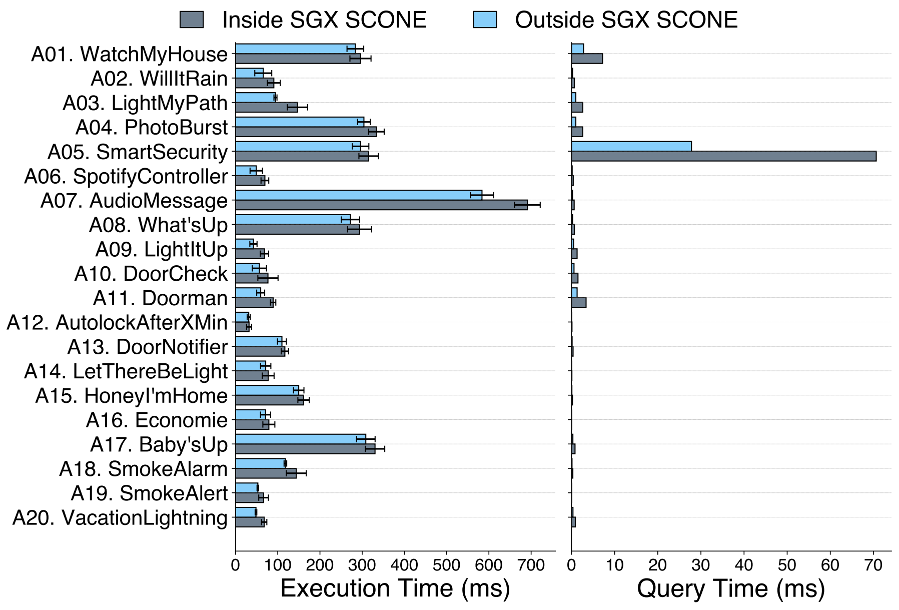


Figure 4.11 – PatrIoT app benchmark performance.

SCONE authors [34]. SCONE is not optimized for network-intensive applications like PatrIoT. Apache outperformed the TSAR service, since the former is multi-threaded, while the latter's Node.js engine is single-threaded.

**Application performance:** The left side of Figure 4.11 displays the execution times for each use-case app. Execution times are tightly dependent on each app's workload, ranging between 32 and 690 ms (inside SCONE). Apps that send sensor data to the Internet or as part of the push notification (e.g. AudioMessage, Baby'sUp) often have the highest execution time due to the network latency and the data transfer rate.

The right side of Figure 4.11 features the time needed to execute a Prolog query and extract flow information from a given app's DFG. The average query time is 4.7 ms and 1.84 ms (inside and outside SGX SCONE) for the apps with a simple DFG. If an app has a DFG with multiple data sources, Prolog's backtracking mechanism requires more time to inspect all possible data flows, e.g., SmartSecurity app with 11 elements 6 of which emit different data types. While its query time is in a stark contrast to other apps it is still below 70 ms.

| Rule ID | Rule Text |
|---|---|
| RB1 | `block Everything from Anywhere to Anywhere` |
| RB2 | `block Everything from Anywhere to Web` |
| RB3 | `block Everything from Anywhere to Phone` |
| RB4 | `block PresenceInfo from PresenceSensor to SmartLight` |
| RB5 | `block Audio from SmartAssistant to Web` |
| RB6 | `block Everything from BabyCam to Web` |
| RA1 | `allow Everything from Anywhere to Anywhere` |
| RA2 | `allow Image from LivRoomCam to Dropbox at 12:00-14:00,Wednesday` |
| RA3 | `allow Everything from LivCam,Alarm,Smoke/Contact/Motion Sens. to ADTSecurity` |
| RA4 | `allow Command from SmartAssistant to Spotify, NYTimes, BBCWeather` |
| RA5 | `allow Everything from Anywhere to John'sPhone, Samantha'sPhone` |
| RA6 | `allow Everything from BabyCam to Nanny'sPhone at 9:00-17:00, weekdays` |

| Rule ID | A01 | A02 | A03 | A04 | A05 | A06 | A07 | A08 | A09 | A10 | A11 | A12 | A13 | A14 | A15 | A16 | A17 | A18 | A19 | A20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| RB1 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 | 🟥 |
| RB2 | 🟥 | 🟥 |  |  | 🟥 | 🟥 |  | 🟥 |  |  |  |  |  |  |  |  |  |  |  |  |
| RB3 |  |  |  | 🟥 | 🟥 |  | 🟥 |  |  |  |  |  | 🟥 |  | 🟥 |  | 🟥 | 🟥 |  |  |
| RB4 |  |  |  |  |  |  |  |  |  |  |  |  |  | 🟥 |  | 🟥 |  |  |  | 🟥 |
| RB5 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| RB6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| RA1 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 | 🟩 |
| RA2 | 🟨 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| RA3 |  |  |  |  | 🟨 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| RA4 |  | 🟨 |  |  |  | 🟨 |  | 🟨 |  |  |  |  |  |  |  |  |  |  |  |  |
| RA5 |  |  |  | 🟨 | 🟨 |  | 🟨 |  |  |  |  |  | 🟨 |  | 🟨 |  | 🟨 | 🟨 |  |  |
| RA6 |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  | 🟨 |  |  |  |

Figure 4.12 – Summary of policy evaluation for use-case apps.

**Note**: red (🟥) and green (🟩) cells denote blocked or allowed apps respectively, apps with yellow cell (🟨) are conditionally blocked, empty cell means the app flows are out of the rule's scope.

### 4.4.3 Policy expressiveness

To assess the expressiveness of PatrIoT's flowwall policies, we have written several allow / block rules that make sense for our smart home scenario (see Figure 4.12). The first three block rules (RB1-RB3) are the most restrictive: RB1 blocks all the app flows, RB2 only blocks flows to the Internet, and RB3 blocks flows to mobile endpoints. RB4 rule displays how a S2S flow (see Section 4.2.2) can be effectively blocked. Rules RB5-RB6 prevent the most privacy sensitive data flows (voice assistant and baby cam) to the Internet. The allow rules (RA1-RA6) start with RA1, which allows all possible flows, followed by more restrictive ones based on certain conditions.

In general, John wants to prevent his smart home devices from accessing the Internet, unless for communication with known and authorized services.

For instance, such privileges are not needed to view the living room camera feed on John's or Samantha's phones. However, John may want to allow camera connections to his personal backup server (e.g. Dropbox) or security company (e.g. ADT) in case of a break-in (rules RA2, RA3). With another rule John can express his privacy concerns regarding a smart assistant device, which can continuously listen for voice commands and can potentially record user conversations and stream audio to unauthorized parties. To prevent this, John can block all the raw audio flows from the smart assistant to the Internet (RB5). For the voice-activated apps that require Internet connectivity specific rules can be defined to grant access to targeted services (rule RA4).

The bottom part of Figure 4.12 shows the results of these rules applied to our use-case apps. If we disregard an RB1 rule we can see that the majority of apps can operate nominally with all other rules in place. In fact, all of these apps operate with device-to-device flows which are usually deemed less privacy sensitive as compared to those that span across different domains (Internet, mobile, etc.). A quarter of apps that issue calls to mobile or web endpoints can be affected by rules RB2 and RB3. However, a set of custom endpoint-based allow rules could be added by the user to unblock these apps.

### 4.4.4 Usability

Our last evaluation goal aims to assess the usability of our system, namely by analyzing the added-value provided by PatrIoT's privacy controls and assessing the users' experience.

**Methodology:** We conducted a two stage user study with 45 participants (computer department employees) with a goal to determine common privacy concerns of the smart device users, and their ability to express these concerns within a PatrIoT's UI. In a first stage, the participants were given the smart home scenario described in Figure 4.9 and asked to decide if a given device data flow should be allowed, blocked, or allowed only in a certain condition. All three data flow types were exercised: S2M, S2W, and S2S.

The second stage of the survey was more practical. With the PatrIoT mobile app the participants had to register a new user account, define policy rules for a baby camera, and then verify a given app's data flows against those rules. In the end we asked the participants to tell us about their experience with PatrIoT, namely, how easy it was to use it and how flexible the policy specification language was when defining data flow rules.

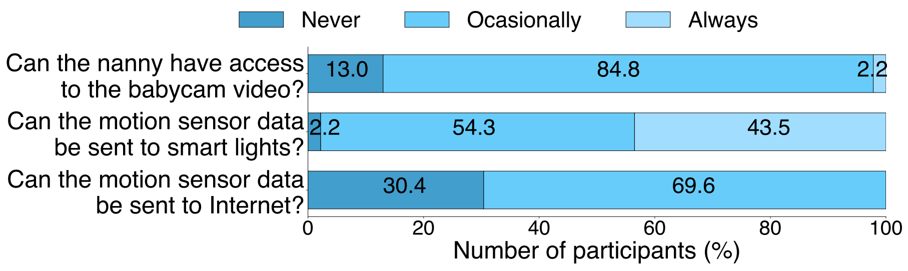


Figure 4.14 – Survey results: privacy preferences.

**Findings about privacy preferences:** Figure 4.14 presents our main findings for the first survey stage, which was split into three tasks. In a first task, we asked the participants to decide if data from the baby camera should be allowed to flow to a nanny's phone. Most participants (84.8%) chose to restrict this flow temporarily (i.e. when babysitting); 13% and only 2.2% decided to always block or allow such a data flow respectively. These results confirmed our expectations: most of the people consider such a data flow to be highly sensitive and want to limit access to it as much as possible.

As part of a second task, the participants were asked to decide if motion sensor data can flow to smart lights. On the one hand, the majority of participants (54.3%) decided to restrict such a data flow to a certain time of the day (when the user is at home). On the other hand, others (43.5%) decided to allow such a flow without any restrictions. Finally, only 2.2% opted for blocking it. The results are in line with our expectations: people are less concerned with the device-to-device data flows taking place entirely within their home domain.

Lastly, in a third task, a motion sensor and its data flows to the Internet were analyzed. All of the participants opted for restricting the data flows in one way or another. We can conclude that participants are cautious even with motion sensor data flows and prefer to restrict those when possible.

**Findings about user experience:** Figure 4.15 presents our findings in the second survey stage. Most of the participants found PatrIoT rules useful in protecting privacy (89.2%). Only a small fraction remained neutral (4.3%) or disagreed with this statement (6.5%). These results highlight PatrIoT's ability to express smart home user privacy preferences of various complexity in a clear and practical way.

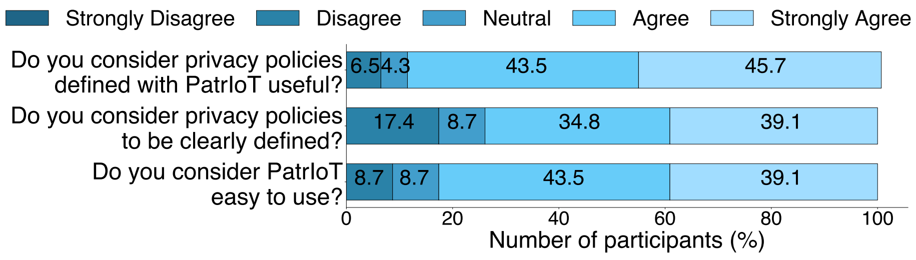


Figure 4.15 – Survey results: user experience.

Regarding the way the security policy rules are defined in PatrIoT, most participants found it intuitive and clear (73.9%). A small fraction of participants however found it slightly confusing (17.4%). Overall, the results are quite promising: a per-device privacy rule approach proved to be clear and easy to grasp and apply. However, some adjustments should be made to make it easier to understand and define privacy rules (e.g. provide a step-by-step tutorial at first run). The participants also suggested adding default policy rules for the average user to use from the start. These rules could be added automatically based on the connected devices.

Finally, 82.6% of participants considered the PatrIoT to be easy and straightforward to use. At the same time, only a small portion of the participants found it neither easy nor difficult (8.7%) or sometimes difficult to use (8.7%). Overall, the interface proved to be clear and intuitive for the majority of people. This is an important finding for us since many privacy-oriented tools often fail to provide a user-friendly interface or require certain expertise from users.

## 4.5 Discussion

Malicious apps may attempt to generate data flows that cannot be monitored by PatrIoT's security monitor. However, by simply crafting an app's element graph and using PatrIoT's API elements, this will not be possible because our system can preemptively create a sound model of all potential data flows based on the app's graph. The creation of covert channels based on communication patterns to authorized network destinations may be possible in the current system design, but they fall outside of our threat model. Devising methods for traffic shaping or bandwidth reduction is an interesting topic for further study.

Currently, our system is dependent on a relatively large trusted computing base (TCB). In particular, PatrIoT's TCB comprises its API and runtime code, Node.js, Javascript engine, and SCONE's library. In spite of this, the total TCB size is comparable with other SGX-based systems, e.g. [185, 181, 130]. Studying ways for reducing the total TCB size constitutes an interesting avenue for future work.

Finally, PatrIoT may be vulnerable to recently demonstrated side-channel attacks on SGX enclaves [26, 72]. While we consider such attacks to be out of scope, PatrIoT can take advantage of various mitigation techniques, e.g. [176], or use an alternative TEE, e.g. ARM TrustZone.

## 4.6 Related work

The general idea of leveraging TEEs and SGX-shielded execution on the cloud has been introduced for some years now [234, 195] and many systems have been proposed to secure various workloads on untrusted cloud infrastructure [47, 130]. PatrIoT is the first system that leverages these techniques to provide IoT service backend protections. Similarly to PatrIoT, a few papers [202, 178] have also suggested to use IFC techniques for prevention of IoT privacy breaches in the cloud. However, in contrast with our work, these authors propose classic IFC models operating at a very low level of abstraction and assume trusted platform providers.

An increasing number of systems has been proposed focusing on security and privacy of existing IoT smart platforms, mostly SmartThigs. Some solutions are concerned about: the physical safety of smart home systems [171, 85], enforcement of IoT network security [173, 209], efficient collection of logging data for ulterior forensic analysis [218, 39], or analysis of security and safety properties of smart apps [66]. PatrIoT complements these systems by focusing exclusively on detection and prevention of privacy-sensitive smart home data flows.

Several papers present refined access control systems for IoT environments that are also concerned about tracking information leakage [136, 67, 224]. However, in some systems the security policies are defined per app, which prevents tracking information flows across multiple applications. PatrIoT overcomes these limitations by providing an original IFC model that can globally track flows across all apps in home.

Lastly, we highlight the differences between PatrIoT and the Flowfence [97] system that we have described in previous chapters. Flowfence allows IoT app developers to split their apps into modules that operate with sensitive data sources and those that do not, and to track the data flows between those parts. However, in contrast to PatrIoT, Flowfence employs dynamic taint techniques that are vulnerable to timing side channel attacks.

## 4.7 Summary

We presented PatrIoT, a private-by-design IoT platform. PatrIoT ensures secure data processing by leveraging SGX-protected environments. It introduces the flowwall security monitor which allows end-users to obtain fine-grained control of data flows generated by IoT apps, and prevent potential privacy violations through the enforcement of a privacy policy. The privacy policy specification language was proved to be effective in describing user preferences regarding sensor data flows, and PatrIoT's UI was found to be easy to use. The latter is encouraging considering that many privacy-oriented solutions that were previously proposed require a certain technical expertise from their users. Our survey results suggested that was not a case for PatrIoT.

PatrIoT extends the dataflow control model from home environment to the untrusted cloud without sacrificing the privacy of its users. All the data processing apps and services that require significant processing and storage resources to provide a given functionality, can now benefit from a secure and flexible environment offered by PatrIoT's runtime.

In general, PatrIoT serves as a main building block for private-by-design IoT systems. With a local HomePad hub running computations at the edge, and PatrIoT complementing it with a secure and private cloud computing, the resulting system provides both privacy and availability guarantees to its users. Such guarantees are essential in a world where more and more devices having access to highly-sensitive user data are connected to the Internet.

Both HomePad and PatrIoT were designed with smart home scenario in mind, which means that their high-level structure closely resembles the general architecture of smart home platforms: local hub - cloud platform - management webapp. There are, however, other IoT scenarios that have a slightly different architecture, for instance, wearables. Various smart fitness trackers, watches and glasses usually require a companion app running on user's smartphone for sensor data processing and communication with a cloud backend. These companion apps have direct access to raw sensor data obtained either from the connected devices (e.g., heart rate) or built-in smartphone sensors (e.g., location), and can share it with unauthorized parties without user awareness. A permission-based access control mechanism employed by popular mobile platforms, e.g., Android and iOS, is unable to prevent such data leaks, since it can only control apps' access to a given data source, but not what the apps can do with the data once the access has been granted. This is the same problem popular smart home platforms face when dealing with home sensor data.

To provide a fine-grained control over data flows generated by the wearable devices, in the next chapter we desribe Flowverine – a system for Android OS which extends the dataflow programming model and its data privacy guarantees to the mobile environment. Floweverine provides a missing piece for building secure and private IoT systems, and closes the privacy gap between the device and mobile endpoints.

# Chapter 5

# Flowverine: private Android apps

## 5.1 Introduction

The number of mobile apps collecting highly sensitive user data, e.g., location, photos, or health-related data from built-in smarthphone sensors or various external IoT devices increased dramatically over the last few years. As the leakage of personal data can cause serious privacy breaches, app developers face the challenge of making sure such data is handled securely. For instance, a fitness-tracking app that reads the user's heart rate from a Fitbit fitness tracker must guarantee that this information can never be shared with unauthorized parties. However, ensuring the absence of bugs and security vulnerabilities is in itself a difficult task due to the complexity of the Android API. Furthermore, any third-party libraries [40, 235] (e.g., ad libs) included in the app, may have their own vulnerabilities, or, worse, contain malicious code leaking user data. Thus, it is important to have mechanisms in place that allow both app developers and users to control sensitive data flows within their apps, and consequently block those flows that can lead to security or privacy violations.

Unfortunately, despite the number of security improvements featured in the latest Android OS versions, no mechanisms are yet available for enforcing information flow control (IFC) policies. Many proposals from academia [49, 223, 68, 220, 221, 41, 129, 198] refine Android's coarse-grained permission system, but fall short at controlling how sensitive data is processed inside the apps. Some systems [36, 155, 158] use static code analysis, which, however, can result in high false positive rates, fail to track flows performed via Android API, or may be impractical to adopt when the source code is not available (e.g., third-party libraries). Other systems overcome these limitations by using dynamic taint analysis [91, 206], but require changes to the Android OS.

To complement existing techniques, we propose Flowverine, a system that allows app developers to build secure-by-design privacy-aware Android apps. Flowverine apps run on commodity Android devices and require no changes to the Android OS in order to track sensitive data flows and enforce security policies. The apps are written using a dataflow programming model and API (possibly including third party libraries) so that all sensitive data flows within an app can be tracked. App developers can specify security policies to whitelist sensitive data flows, e.g., "*heart rate readings from a user's fitness tracker can be sent exclusively to a specific cloud backend and nowhere else*". Users that install the app can verify such policies and employ additional restrictions.

Flowverine implements a taint tracking mechanism based on two techniques. Firstly, to increase the abstraction level and enable efficient static taint tracking, we built upon the concept of *element-based programming* and apply it to complex Android apps. As in HomePad and PatrIoT, in Flowverine, all sensitive data flows must become explicit by construction, i.e., an app must be written as a graph of *elements*, in which elements represent compute units and the edges represent data flows. Flowverine provides a set of *trusted native elements* that mediate an app's access to the Android native API. Because trusted elements come with a specification that describes how the data flows through the Android runtime, Flowverine allows for sound static taint tracking to be performed across Android API calls.

Secondly, if an app includes third party code that needs to access the raw Android API, Flowverine uses sandboxes to isolate such code inside *untrusted elements*, and Aspect-Oriented Programming (AOP) to intercept native Android API calls and perform dynamic taint analysis in such specific cases. AOP precludes the need to modify the OS, thus favoring compatibility.

Our performance evaluation shows that Flowverine has a relatively small impact on app execution time and has no noticeable impact to the user experience. We implemented three use case Android apps that showcase the ability of Flowverine to (1) prevent sensitive data flows that are not explicitly indicated in the app graph provided by the app developer, (2) allow for the strict privilege separation of multiple independent flows within any given app, and (3) support the main Android API programming abstractions.

### 5.1.1 Building privacy-sensitive Android apps

Android provides a popular platform for mobile apps. We highlight three major challenges faced by developers when building apps that manipulate privacy-sensitive data from local sensors or external connected devices. These challenges arise mainly from Android's programming and security models.

**Tracking direct sensitive data flows:** Tracking information flows between *source* and *sink* Android API calls – i.e., the calls that allow an app to obtain sensitive data and send it to remote parties, respectively – based on the inspection of a data flow graph can be cumbersome and error-prone as a result of the app separation into components (e.g., Activities) and the asynchronous nature of Android programming. Many static analysis tools can help to automate this task, but are seldom used in practice because of high false positive rates.

**Tracking indirect sensitive data flows:** Sensitive data flows can also be generated indirectly, i.e., outside the data flow paths between source and sink API calls. Some flows can be established through internal Android data structures, e.g., via an app context (akin to a global object store) in which independent app components can store and retrieve data using specific API calls. An indirect flow can then occur if one component stores data inside the app context and another one reads that data from it. Tracking such flows using existing taint analysis tools requires changes of the Android OS [91].

**Enforcing privilege separation:** Another difficulty lies in the fact that Android's permissions are too coarse-grained and many Android API calls have no differentiated access controls for different parts of a given app. This complicates privilege separation for different pieces of app logic. For instance, once the network access has been granted to an app, one cannot restrict the range of endpoints that the code (e.g., a third-party ad library) can connect to. Android does not support access control policies based, e.g., on the target URL.

### 5.1.2 Element-based programming for Android apps

Given the benefits provided by element-based programming within a smart home scenario (see Chapter 3), we propose to adopt it for building secure-by-design Android apps that provide the same privacy and security guarantees as HomePad or PatrIoT apps. As such, we introduce several innovations which we describe below:

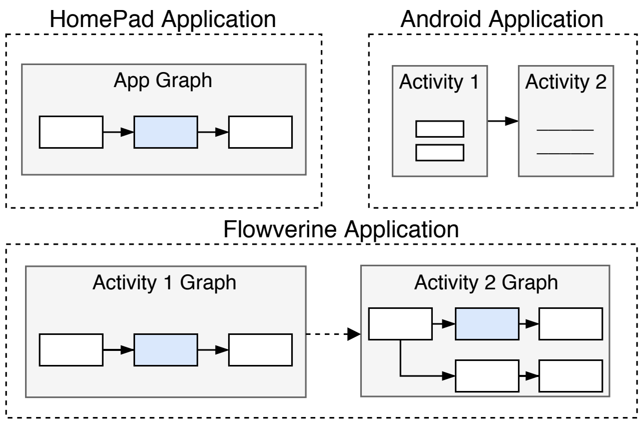


Figure 5.1 – Programming models compared.

**1. Android app components as element graphs:** In Flowverine, each app component is written in the form of an element graph (see Figure 5.1). As in HomePad, an element executes some functional unit, and can only interact with other elements through the explicit edges connecting them. The graphs are expressed in a declarative fashion, which allows for integration with popular visual programming tools for app development.

**2. Trusted elements adopted for mobile API:** The Flowverine API consists of a set of trusted elements. These are provided by certified modules that are assumed to work properly without undesirable side effects. Access to the native Android API, e.g., network calls, is mediated by specific trusted elements that can be used for different purposes, namely; i) obtain data from a given source (e.g., a hardware sensor, UI, or another app component), ii) send data to an external sink (e.g., a network host, UI, or another component), or iii) perform data transformation (e.g., data encryption). Each trusted element provides a well-defined interface.

**3. Untrusted elements to host unmodified legacy code:** As in HomePad, untrusted elements serve the purpose of running sandboxed code provided by the developer. In addition, Flowverine supports the inclusion of third-party legacy libraries, which often require direct access to the native Android API. To prevent privacy breaches Flowverine hosts legacy code inside untrusted elements and needs to implement additional runtime mechanisms to block any unauthorized API accesses.

### 5.1.3 Challenges related to Android specifics

Unlike HomePad apps that usually have a simple structure, Android apps contain multiple components (e.g., activities or services) which interact with each other through asynchronous callbacks. Tracking sensitive data flows in such an intertwined system of classes and methods is a challenging task. To address this challenge, Flowverine implements a middleware that provides an abstraction layer for all app components, including the UI ones, and controls the propagation of events carrying sensitive data between them. Flowverine intercepts native API calls and enforces runtime security policies without changes in the underlying OS. We provide more details in the next section.

## 5.2 Design

This section presents Flowverine. We begin by describing its architecture, and then discuss its most relevant design details.

### 5.2.1 Architecture

Flowverine provides a software framework for development of privacy-sensitive Android apps such that the developers and users alike maintain fine-grained control over the sensitive information flows generated by these apps. To this end, Flowverine provides a middleware that exposes an API based on element-based programming and a set of mechanisms that i) analyze the internal app data flows using static and dynamic taint tracking, and ii) check such flows against an information flow control (IFC) policy to identify potential security or privacy breaches. An app developer can specify an IFC policy to validate the app compliance with the terms of the service's privacy policy (which states how the personal data will be collected and managed) and the data protection rules imposed by law. The user can specify an IFC policy (through a user-friendly interface) which prevents the creation of specific data flows that the user deems privacy sensitive.

Figure 5.2 presents Flowverine's components. It includes an *app development toolchain* that allows developers to build their apps, link them against the Flowverine API, and check compliance against a developer-provided IFC policy. If the app satisfies all the security requirements, the developer submits a signed app package to an app store and registers it in the Flowverine *certification service*, which validates that the app has been properly instrumented by

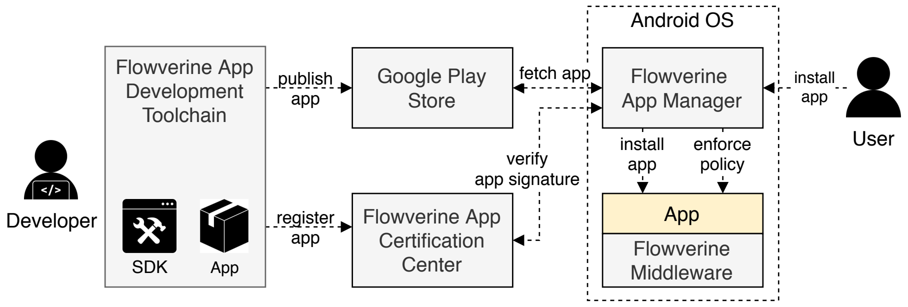


Figure 5.2 – Flowverine framework components and workflow.

the toolchain. A user can then install this app through the Flowverine *manager app* running on the user's smartphone. The manager app manages all Flowverine apps on the device, e.g., fetches the app package from the app store, and checks that the app has been properly certified by the Flowverine certification service. The manager app also provides a UI interface through which the user can specify an IFC policy and check apps' compliance with it. If the app passes the check, it can then be executed. Every Flowverine app is linked against the Flowverine *middleware* – i.e., a set of libraries – that provides all the runtime support for the execution of the app, which is based on an element graph. Next, we describe how a Flowverine app can be developed.

### 5.2.2 Application development

The process of developing a Flowverine app involves i) the implementation of the app itself, and then ii) using the toolchain to build the app, check IFC policy compliance, generate the app package, and submit it for public release.

To implement an app, the developer creates individual element graphs for every app component. To illustrate this process, imagine we want to implement a simple ClickCounter app that displays the number of times the button on a screen was clicked. As in traditional Android programming, in Flowverine, this app has an associated Activity and a UI layout file written in XML. However, since this Activity will be implemented as an element graph, it will be programmed as a Java subclass of Flowverine's API `ActivityGraphDescriptor`. This class provides methods that allow the developer to specify the elements of the graph and their connections. Figure 5.3 shows what this graph looks like.

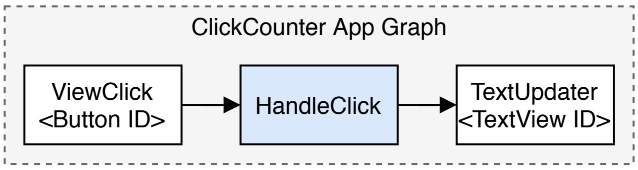


Figure 5.3 – Element graph of ClickCounter app.

```
@CustomElement(name="HandleClick")
public class HandleClick extends Element{
  int ctr = 0;
  @EventReceiver
  public void onEvent(...) {
    sendEvent(new Event<String>("Cnt: "+(++ctr)));
}}
```

Listing 5.1 – Implementation of HandleClick element.

This graph consists of two trusted elements, namely `ViewClick` and `TextUpdater`, and an untrusted one – `HandleClick`. The former implement UI functions and serve as interfaces to the button and a text view defined by their respective IDs. The latter contains the code that handles a button click event and increments the counter. This code of this element is provided in Listing 5.1. According to the app graph, the events generated by `ViewClick` will be routed by the Flowverine runtime to `HandleClick`, which in turn will increment the counter and generate an output event. The Flowverine runtime will route this event to `TextUpdater` which will display the counter value on screen. Next, we present the Flowverine runtime internals.

### 5.2.3 Application execution runtime

The Flowverine runtime (see Figure 5.4) consists of a middleware comprising several libraries, which are included in the app package along with the code responsible for the implementation of the app's element graph. At runtime, Flowverine materializes the elements of the app graph into three sets of Java objects : i) *stubs* that point to the implementation of the trusted elements referred to in the element graph, ii) *sandboxes* initialized with instances of untrusted elements' classes, and iii) a *path descriptor* which restricts communication between trusted elements' stubs and untrusted elements' sandboxes according to the connections in the app graph.

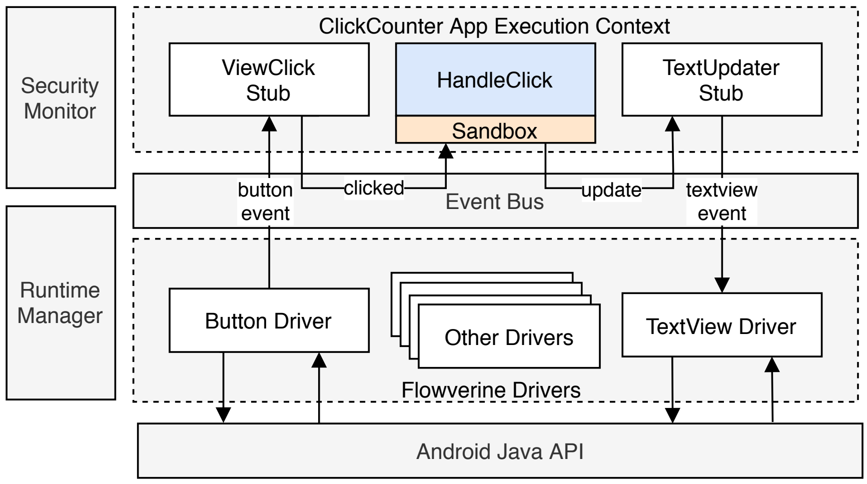


Figure 5.4 – Execution runtime of a ClickCounter application.

Elements communicate by sending events to the corresponding stub through an internal message broker: *event bus*. For instance, `ViewClick` element sends an event on every button click. These events are routed by the event bus strictly as specified in the path descriptor, therefore ensuring that no information flows can occur besides those specified in the app's element graph. The functions implemented by the trusted elements – through a set of built-in drivers – is covered below.

### 5.2.4 Trusted elements API and drivers

The Flowverine API consists of a set of trusted elements that developers can use to create their apps' element graphs. The logic of these elements is implemented by a set of drivers which are part of the Flowverine middleware. However, one of the potential obstacles in adopting such element-based programming for Android, is the complexity of Android API, both in terms of number of calls, and the sophistication of operations they implement (e.g., multithreading). To cope with this complexity, we created various types of drivers which interface with specific classes of functions provided by the Android API. Next, we briefly mention the most important Flowverine driver types:

**1. UI drivers:** As opposed to smart home apps, mobile apps have very rich user interfaces. The Android API has many classes for creating UI widgets named Views. A View represents a UI object on the screen which the user may interact with. Flowverine's API offers trusted UI elements, e.g., the `ViewClick` and `TextUpdater`, which provide standard functionalities of Button and TextView, respectively. These elements have specific input and output ports which can be connected to other elements. An input port of `TextUpdater` element can be used to update a TextView on the screen, and an output port of `ViewClick` can be used to emit a button click event to any downstream element in the app graph.

**2. Component drivers:** Activities are very common components. An Activity represents an app's screen and is in charge of UI-dependent tasks. Throughout its lifecycle an Activity instance transitions through different states and provides a set of callbacks that are invoked when it enters a new state – e.g., `onCreate` or `onDestroy`. With Flowverine, developers can handle these state changes by using the elements provided as part of the Activity Life Cycle Module. For instance, the `ActivityCreated` element notifies the elements connected to its output port when the graph's Activity is created.

**3. System drivers:** This class of drivers includes trusted elements for supporting multithreading, inter-component communication (ICC), and inter-process communication (IPC). For multithreading support, Flowverine apps can execute tasks in parallel with the `AsyncFork` and `AsyncJoin` elements. With ICC driver elements the graphs of different app components can be connected. Lastly, IPC drivers enable apps to interact with each other via the `Send` or `Receive` trusted elements.

**4. I/O drivers:** Flowverine implements several drivers for interfacing with network, storage, and sensors. For networking, Flowverine includes a Web driver which allows apps to perform HTTP requests through trusted elements, such as `HttpGetReq` or `HttpPostReq`. These elements must be set up with i) the destination URL, and 2) the expected data types received in the response. Other drivers provide access to bluetooth and location services.

### 5.2.5 Protection against untrusted element code

The code of the untrusted elements can be written by the app developer or be part of a third-party library. In either case, to ensure that the app's data flows are strictly bound to the data paths indicated in the app's element graph, such code cannot be allowed to execute without restrictions.

Flowverine adopts two mechanisms for securing legacy third-party libraries:

**1. Sandboxing untrusted elements:** To prevent untrusted element code from interfering with other classes of the runtime sharing the same ART virtual machine, we take advantage of Java's class loading model. Flowverine includes a custom-made sandbox classloader which is in charge of resolving classes within an isolated namespace. Each untrusted element instance is placed inside its own sandbox such that only the classes associated to it by the app developer can be loaded and instantiated. Any attempts to access (blacklisted) classes from the runtime environment will throw an exception. Some (harmless) classes are whitelisted and are delegated to the parent class loader, i.e., the class loader of the runtime.

**2. Weaving untrusted elements code:** It is also necessary to prevent untrusted elements' code from performing operations that circumvent the data paths defined in the app graph. This may cause a buggy code to interfere with the system or, worse, a malicious code (e.g., spyware shipped with a third-party library) to leak sensitive data. Therefore, an untrusted element code must be prohibited to perform the following operations:

- Direct calls to the Android API methods, which are reserved to be invoked by the Flowverine middleware.
- Execution of native (C/C++) code, which could be used to inject malicious code in the Flowverine runtime.

To this end, the code is sanitized using Aspect-Oriented Programming (AOP). With AOP, we define a set of execution points patterns to be executed only by the middleware. By weaving the app in search of points that match these patterns, and injecting a safety-guard code, we can assure that untrusted elements' code has no access to Android's API or to a Java native interface. In Flowverine, weaving is performed at build time, by a tool of Flowverine toolchain named *code weaver*. It runs on Java bytecode files and inspects all the app code provided by the developer, including any imported libraries.

Weaving is particularly useful in the case of legacy third-party libraries which have not been modified to use Flowverine's trusted element API. At runtime, if an untrusted element attempts to execute a flagged Android API call, the safety-guard code takes over and lets the security monitor (see Figure 5.4) decide what to do. The default procedure is to terminate the app, but the security monitor may allow the operation to proceed as long as the resulting data flow follows the app's graph connections. For instance, it can intercept an HTTP call and forward it to the Flowverine network driver, which, in turn, translates this call into an event compatible with a trusted HTTP element. If such an element exists in the app's graph and connects with the currently executing untrusted element, then this operation can be seamlessly carried out.

### 5.2.6 Validation of information flow control policies

By ensuring that an app can only generate information flows explicitly declared in the app's element graph, Flowverine helps prevent security breaches that may result from programming errors or by the inclusion of malicious libraries. Flowverine provides complimentary tooling support for validating the information flows of a given app against an information flow control (IFC) policy. Although for different contexts, IFC policies are useful to both app developers and users.

Following the dataflow programming model, an IFC policy consists of a set of rules aimed to flag specific information flows between sources and sinks in a given app. Flowverine validates if any of the app flows violate the IFC policy by using a Prolog engine to query the app's model based on the policy rules. Internally, there are two types of tools for IFC policy validation. App developers can use a *policy checker* included in the toolchain to check for undesired information flows. For debugging and testing purposes the app developers can specify their own IFC policy in JSON (then converted to a Prolog predicate and checked against the app's element graph).

App users can use the App Manager to supervise the information flows generated by Flowverine apps and block any sensitive flows. Figure 5.5 presents two screenshots of the App Manager's UI: app installer view (left) and an app privacy report view (right). To install an app, the user selects an app from a list provided by the Flowverine app certification center. During installation, the App Manager generates a default IFC policy that reflects the flows in the app's element graph. This policy is shown to the user, who can block specific data flows or disable the app. The user may additionally force the app to ask permission every time it attempts to obtain or send out a certain data type.

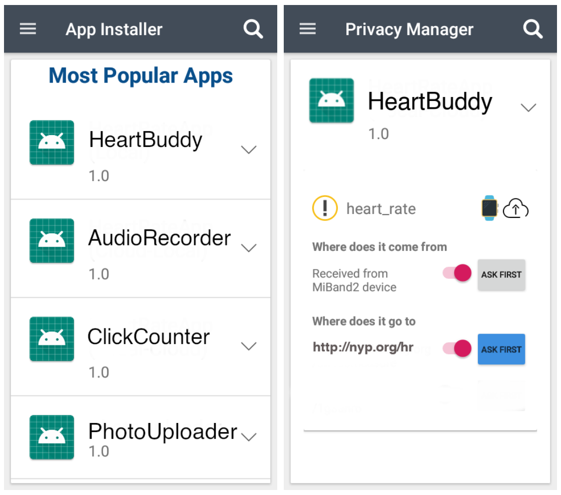


Figure 5.5 – App Manager interface views

## 5.3 Implementation

We implemented a Flowverine prototype, and prepared a public release as an open-source project. In total we wrote about 23K lines of Java code. This includes the Flowverine middleware and trusted element API (9K LoC), the App Manager (3.5K LoC), Flowverine toolchain (1K LoC), the certification service (0.7K LoC), and five testing Flowverine apps (9K LoC). We adopted tuProlog as Flowverine's Prolog engine, and leveraged AspectJ for code weaving. We implemented a specific Flowverine BLE driver for interacting with a Xiaomi Mi Band 2, which we used to develop a privacy-sensitive fitness tracker app.

Our current prototype has several limitations. Given the extent of the Android API, we have only implemented a representative set of trusted elements for the Flowverine API. In particular, our API is limited to: system drivers, Activity and Service components, five different UI views, and I/O drivers for networking, BLE interfacing, and location services.

## 5.4 Evaluation

We evaluated Flowverine on several fronts. We first examined the developer effort required to build Flowverine apps as compared to the standard Android apps. We then analyzed Flowverine's performance with respect to applica-

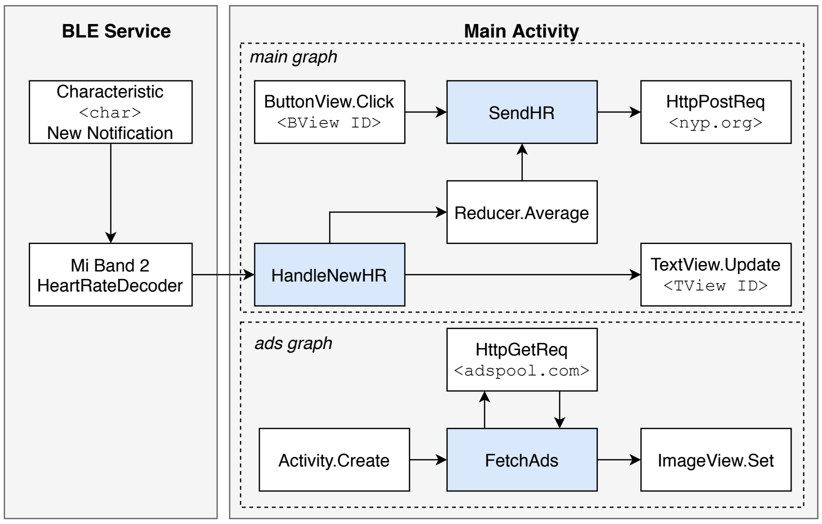


Figure 5.6 – Flowverine HeartBuddy app design.

tion compilation and packaging, as well as, runtime performance and memory consumption. Next, we present a case study used in this evaluation.

## 5.4.1 Case study

To help understand some key challenges in building secure mobile apps, we introduce a simple health-monitoring app named HeartBuddy. The app obtains a heart rate value from a connected fitness tracker – via a Bluetooth Low Energy (BLE) connection – displays it on the screen, and periodically sends an average value to a hospital's cloud service (nyp.org) for diagnosis of various heart-related diseases. The app also displays an ad banner fetched from a remote server (adspull.com).

Due to the private nature of a heart rate data the app developer must ensure that only the average values are sent to the specified cloud service and nowhere else. Likewise, the app users expect this property to hold at any time. Additionally, the developer needs to guarantee that there is no interference between the main app functionality and the ad library activity. The ad library can never have access to heart rate data.

In Flowverine, the HeartBuddy app can be implemented as a graph of elements displayed in Figure 5.6. On the left side, there is BLE Service activity responsible for interacting with a connected fitness tracker and properly decoding its signals (proxied by native Android API). On the right, we see two app activity graphs: one implementing the main app functionality, and the other one responsible for ad banner activity. A new heart rate data event emitted by a trusted `HeartRateDecoder` element arrives to an untrusted `HandleNewHR` element, which forwards it to the `Reducer.Average` element that computes an average heart rate value and feeds it to the second untrusted app element – `SendHR`. The latter one is responsible for preparing an HTTP POST request to a hospital's cloud server. This request will be sent when the user clicks the "Send" button on the screen (an event handled by the `ButtonView.Click` element). Finally, the `TextView.Update` element updates the current heart rate value on the screen.

The ad banner operations are controlled by a second isolated graph consisting of four elements. This graph starts execution when a new activity is created (invoked by `Activity.Create` element). The untrusted `FetchAds` element receives the latest ad data by making an HTTP GET request to an adspull.com service and displays it on the screen via `ImageView.Set`.

Since the main activity graph and the graph responsible for ad activity are completely separated, there are no data flows between their respective elements. Flowverine also ensures that the network calls are restricted to endpoints that were defined in the app package: `HttpReq` controls the destination and type of requests. By analyzing both app graphs, Flowverine can effectively track the heart rate data propagation and transformation. The security monitor detects the data type leaving the user phone (averaged value) which is in accord with the user expectations. The security monitor also ensures that a third-party ad library will not have access to heart rate data and will not be in conflict with any of the GDPR regulations.

### 5.4.2 Comparison with legacy Android apps

Our comparison between Flowverine and the legacy Android system is twofold. First, we analyze the security models of both systems: the former which is based on an IFC model, and the latter on a discretionary permissions system. Our goal is to evaluate if apps developed with Flowverine are more transparent regarding their sensitive data flows, and if our framework allows users to understand and have a fine-grained control over how installed apps treat sensitive data. To this end, we use the example HeartBuddy app (see Figure 5.5).

| App Name | Accessed Resources | Lines of Java Code (LoC) | |
|---|---|---|---|
| | | Traditional | Flowverine |
| ClickCounter | UI | 15 | 21 |
| PhotoUploader | UI, Filesystem, Internet | 98 | 64 |
| HeartBuddy | UI, Dialog windows, Internet, Bluetooth, Mi Band 2 services | 480 | 174 |

Table 5.1 – Apps created for bare Android and with Flowverine.

In Flowverine, the App Manager reports to the user that: (1) the app collects heart-rate data (i.e. data type) from a fitness tracker (i.e. source), and (2) the app sends collected data to *nyp.org* (i.e. sink). The user is then offered the option to either block a given flow or require the app to ask for permission each time the flow occurs. In vanilla Android, the permissions system allows the user to deny the app's access to BLE service, but not to the Internet. Thus, in scenarios where mobile apps need to send sensitive data to the cloud, Flowverine's reports are more informative and give the user better control over the app's activities than Android's native permission system.

Secondly, we assess the development effort required to write Flowverine apps in terms of lines of code (LoC) as compared to the standard Android app programming model. Table 5.1 presents the results of this comparison for three apps of various complexity: ClickCounter (see Figure 5.3), PhotoUploader (which uploads a photo to a cloud service), and HeartBuddy. We see that for very simple apps, Flowverine requires more lines of code. However, the LoC number is significantly lower (sometimes almost 3x less) for complex Flowverine apps that rely on multiple existing trusted elements to interact with various resources (e.g. device sensors, storage, network). Developers can thus benefit from higher-level programming abstractions for writing their apps.

### 5.4.3 Performance

To evaluate the performance of Flowverine, we used a server with a 2.80GHz Intel i7-7700HQ CPU and 16GB of RAM for build-time and validation experiments. To evaluate the Flowverine's runtime and App Manager performance, as well as its memory and battery consumption we used Neffos C5A Android 7.0 smartphone with a 1.30 GHz CPU, 1GB RAM and a 2300 mAh battery.

| **App Name** | **Build time (ms)** | | **Overhead (ms)** |
|---|---|---|---|
| | **Traditional** | **Flowverine** | |
| ClickCounter | 1556 | 2164 | 608 (39%) |
| PhotoUploader | 1938 | 2772 | 834 (43%) |
| HeartBuddy | 1692 | 2374 | 682 (40%) |

Table 5.2 – Build time for traditional and Flowverine apps.

To measure how much time Flowverine adds to application compilation and packaging, we compare build times of the three use-case apps developed using Flowverine and traditional Android programming models (see Table 5.2). Flowverine adds on average 700 ms to the build time. In all cases the overhead was mostly due to Code Transformer's weaving process.

On average it takes 7.7 sec for Flowverine to perform an integrity check on a newly published app package. The validation time depends mainly on the app size, but with the infrequent app release cycle, this delay can be tolerated.

Next, we analyzed the time that the App Manager needs to inspect an app graph, extract data flows information, and display this information to the user. The inspection time correlates closely with the app graph complexity: with more app elements generating various data types there are more potential data flows for App Manager to inspect. It takes between 2 to 7 sec to analyze the app graphs consisting of 3 and 21 elements respectively.

We also evaluate the Flowverine impact on apps' startup time and some of the common app activities. The results are presented in Figure 5.7. Flowverine adds, on average, 200 ms to an app's launch time, and 20 ms when switching app activities. However, for other app activities, e.g. network calls, the overhead is negligible (<1 ms). While Flowverine has a noticeable impact on app startup time, there is no meaningful performance loss on app activities after that. We note, however, that further performance optimizations are possible.

Lastly, Figure 5.8 features the results of memory consumption comparison. Flowverine apps use slightly more memory due to the sandboxing mechanism which replicates classes bytecode definitions consequently increasing the amount of memory used by the app process. Also note that in our experiments Flowverine had insignificant impact on app's battery usage.

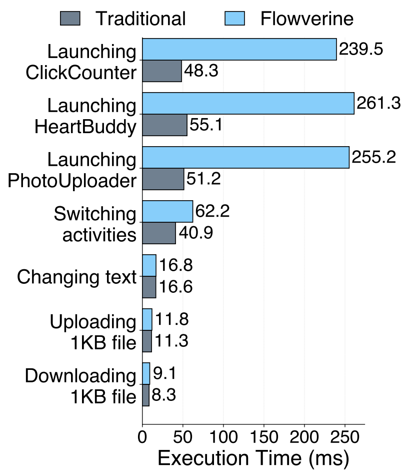


Figure 5.7 – Runtime benchmarking tests results.

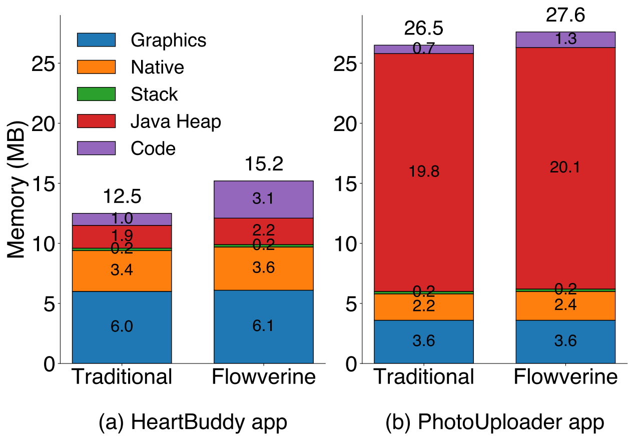


Figure 5.8 – Memory usage: traditional apps vs. Flowverine apps.

## 5.5 Discussion

Our system must defend against potential security vulnerabilities introduced by buggy or malicious code contained in a mobile app. Such vulnerabilities could result in the circumvention of the data path restrictions enforced by the app's element graph, and / or in the violation of a given IFC policy. To assess how Flowverine mitigates potential attacks, we consider four scenarios:

1. Untrusted elements interact directly with device resources through an Android API (direct access attack).

2. Unconnected untrusted elements of the same app graph sharing data with each other (data sharing attack).

3. Malicious code set to run outside untrusted elements by executing native C/C++ code (middleware bypass attack).

4. Altering the app's bytecode after the weaving-based sanitization has been performed (weaving disable attack).

Flowverine introduces several mechanisms to make app code more resilient to attacks. Its sandboxes prevent (1) and (2) by blocking the execution of dangerous classes that aim to access the device's resources, and loading the classes of untrusted element code in independent class loaders so that they do not share memory. Flowverine's code weaver tool checks all the apps bytecode, preventing attacks of the third kind. Lastly, the certification service of Flowverine only validates apps that have been correctly sanitized by the code weaver, essentially preventing attacks of the fourth type.

## 5.6 Related work

The most relevant solutions that aim to improve privacy in Android devices can be divided in to two categories: data-flow analysis tools that inspect apps either statically or at runtime, and extensions to Android's permission system that offer a mechanism to determine what data an app has access to.

**App data flow analysis tools:** The main goal of the data-flow analysis tools is to inform users about possible leaks or dangerous behavior by applications when treating sensitive data. Most of these tools [36, 155, 91, 163] employ some kind of taint tracking to inspect the paths of tainted data samples. While these tools provide high coverage, they may often lead to a high false-positive rate, and overlook some control-flow data leaks. MutaFlow [158] detects this last type of leaks but fails to detect a delayed attack.

Furthermore, a study carried in 2018 [187] shows that FlowDroid and IccTA fail to track flows that involve ICC calls with complex strings formed from sensitive data. TaintDroid [91] and TaintART [206] overcome these challenges, but require changing Android's core as they use dynamic taint tracking. Additionally, these tools operate at a variable level and are prone to side-channel attacks [38]. Flowverine provides a complementary technique that combines static and dynamic taint analysis without changing the Android OS.

Moreover, the systems discussed in this section only aim at informing the user about the potential data leaks but do not provide any mechanisms to enforce privacy policies. Some solutions, however, give users more control over their data by extending Android's permission system, which we discuss next.

**Extensions to Android's permission system:** Most proposed extensions [221, 41, 68, 220] enforce control over app data access but ignore internal app data flows, making it difficult to determine, e.g., the Internet locations where sensitive data is sent by a given app. Some systems that monitor how data flows within apps [129, 198], rely on TaintDroid [91] to detect leaks, which means they inherit TaintDroid's limitations discussed above. Furthermore, all these solutions [49, 223, 68, 220, 129, 198] but two [221, 41] involve changes to Android's core. Aurasium [221] and AppGuard [41] do not modify the OS, but instead rely on dynamic instrumentation and repackaging of apps which alters an original app signature. Modifications of the Android core or app packages interfere with Android's security ecosystem and raise compatibility problems. Another problem with the studied solutions is that they all control access to data at a very low abstraction level, such that it becomes hard for app developers and users to understand how apps use sensitive data and for what purposes.

In summary, Flowverine finds itself in-between two worlds. On the one hand, similarly to taint tracking tools, it checks the propagation of sensitive data samples from their sources to potential sinks. Flowverine however avoids overtainting by operating on a higher level of abstraction (variable level vs. user-friendly data type) and relying on predicates that describe data propagation rules within the app. On the other hand, Flowverine acts as a privacy enhancement to the Android OS and requires no changes to its core modules.

## 5.7 Summary

In this chapter, we presented the design and implementation details of Flowverine, a privacy-aware middleware that helps both Android users and app developers safeguard the formers' privacy. Flowverine successfully adopts a dataflow programming model in the context of mobile apps, and allows developers to transparently expose internal data flows of their Android apps for verification and analysis.

Android users can benefit from Flowverine's flexible privacy policy language to express their privacy preferences and expectations when installing a given Flowverine app. These preferences translated into privacy policy rules will be enforced at runtime for all the data flows generated by an app. In contrast to existing Android's permission-based access control system which only allows to control access to a given sensitive data source but not the app's data sharing capabilities, Flowverine provides a much more fine-grained access control, in which the user can restrict the explicit types of data the app has access to and the endpoints the app is allowed to communicate with. Additionally, users can enforce data obfuscation or anonymization whenever sensitive data samples need to be sent to the remote endpoints.

Flowverine can also be a useful tool for app developers seeking to ensure compliance with their advertised privacy terms or regulations enforced by law (e.g., GDPR). With all the data flows made explicit, Flowverine allows the developers to clearly separate data flows generated by the main app activities and those generated by third-party libraries, e.g. ad libs (if any). As a consequence, any unintentional disclosure of sensitive data will be discovered and blocked.

Our evaluation shows that Flowverine performs well, and that, despite the introduction of a new programming model, new private-by-design apps can be created without any significant effort. In fact, with the rich set of built-in elements, Flowverine apps often require less lines of code as compared to vanilla Android apps. Furthermore, Flowverine can make app development even more accessible to a wider community, as it can be easily integrated with visual programming tools.

Flowverine represents a final building block in private-by-design IoT systems spanning across local home, cloud and mobile domains. The dataflow programming model which is the central component of all three systems proved to be effective in capturing user privacy preferences and enforcing those at runtime. It is also highly-extensible and pluggable making it easier for developers to create and share new elements with the community. However, third-party elements pose a security and privacy threat when obtained from the untrusted sources. In the next chapter, we will describe a way to bootstrap trust in these elements even if they were developed by potentially malicious developers.

Chapter 6

# Bootstrapping trust with NVP

## 6.1 Introduction

Dataflow programming model relies on a set of trusted elements provided as part of an API. These elements mediate access to sensitive sensor data (e.g., IP camera's frames), perform common data computations (e.g., face or speech recognition, data anonymization, etc.), or send data to the remote endpoints on behalf of the untrusted app code. They are provided by a community of third-party developers and are deemed to correctly implement the desired functionality. The problem, however, is that if a buggy or even malicious element implementation is installed in the system, serious security breaches can take place. Our goal is to investigate the adoption of N-version programming (NVP) in the design of IoT systems using dataflow programming model as a way to enhance security and prevent leaking raw sensor data through ill-behaved elements.

By using NVP, rather than depending on a single implementation, each API element depends on N different implementations (versions) that must concur to produce the final result. The runtime system feeds sensor data as input to each of the N element versions, and determines the overall output result based on a particular decision policy. For example, with total agreement policy, all partial outputs must be equal otherwise no output is released. A quorum policy requires only a quorum of equal partial responses to be reached. We envision different versions to be developed independently by an open community of developers. Insofar as the developers do not collude, N-version trusted elements are no longer dependent on the correctness of any specific element implementation as it is the case in existing IoT platforms.

Although applying NVP to the IoT architecture is relatively straightforward, the degradation of utility and performance can undermine the viability of this technique. The utility is penalized if an N-version module, i.e., a group of N implementations of the same element, frequently blocks any output to the application due to result divergence reasons. Additionally, the performance of an N-version module tends to be bound by the slowest element implementation involved in the output decision. In our context, the impact to utility and performance will greatly depend upon how elements are implemented. If elements are developed from scratch, we expect most of the negative effects to be caused by implementation or performance bugs introduced by the developers. On the other hand, if elements are built upon pre-existing code (e.g., libraries) such effects may also stem from incoherent specifications. The decision policy employed also plays a critical role in determining the behavior of modules.

In this chapter, we provide an extended case study about the feasibility of NVP for securing IoT systems. It seeks to characterize the impact of NVP on utility and performance of API elements. To this end, we perform an in-depth study focusing primarily on two main causes: software flaws and specification incoherence. We built multiple test modules performing a variety of privacy-sensitive functions, such as image blurring, voice scrambling, k-anonymization, face recognition, and speech recognition, among others. Then we tested them extensively in different N settings and decision policies.

Our study reveals that NVP has a considerable potential for practical application within an IoT environment. In particular, we found that: (1) for N-versions that implement the same algorithm and follow the algorithm specification, it is possible to provide an N-module offering high utility as long as the number of software flaws is residual, (2) for N-versions that do not follow the same algorithm but perform the same task, we observe that although module utility can be negatively affected by output divergence, it can be improved by leveraging decision policies tailored to the problem domain space, and (3) N-version module performance is typically bound by its slowest version, a condition that can be mitigated by leveraging versions redundancy.

Next, we provide a more extensive overview of our motivation, approach, and goals. In Section 6.3, we introduce an IoT system architecture based on NVP. Then, we present the main contributions of this work: a comprehensive study of the impact of NVP to elements' utility (Sections 6.4 and 6.5) and performance (Section 6.6).

## 6.2 Overview

### 6.2.1 Trusted elements: goods and ills

To prevent unlimited access to sensor data, all the systems we have described in the previous chapters allow their APIs to be extended with *trusted elements* (TEs) aimed to implement high-level operations that mediate access between the application and the raw data. TEs are developed by third-party developers that are fully trusted to implement them correctly. As long as the latter holds, such TEs constitute an effective approach to securely processing sensitive data. However, malicious TE implementations can perform serious attacks:

**A1. Incomplete results:** during processing, a malicious TE could intentionally omit parts of the results in an effort to disturb users' actions, e.g., hide the part "and John" when recognizing the user voice command "send a message to Rachel and John".

**A2. Incorrect results:** similarly to the previous attack, a malicious TE could alter the results, in order to trick the user into performing an unexpected and potentially harmful operation, e.g., replace the name of the person the user wants to call with a premium number, when recognizing the user voice command, or dismiss a smoke detector alert silently putting user at risk.

**A3. Data inferences:** in collusion with a malicious application, a malicious TE could not only perform the operation it intended but also make inferences on the raw data and disclose it to the application, e.g., identify the people in the room in addition to recognizing the user voice command.

**A4. Raw data leakage:** the most devastating attack is the one where a malicious TE colludes with a malicious application and leaks raw data, e.g., send a raw camera frame as face recognition output.

### 6.2.2 Leveraging N-version programming

While the effects of attacks A1 and A2 can also stem from naive implementations, which are difficult to distinguish, we argue that attacks A3 and A4 are the sole product of lack of platform control over TE outputs. Thus, we seek to understand whether relying on multiple TE implementations can mitigate these attacks. In particular, we aim to study the ability of NVP to prevent malicious TE implementations from exfiltrating sensitive user data.

TE implementations are expected to follow a TE specification. To this end, we assume that the TE specification is publicly available among the developers and users. As for a TE implementation, the TE binary needs to be publicly released, possibly even after being properly obfuscated. An NVP-based TE system must be able to detect the deviations in the elements' outputs and react accordingly.

The N-version decision algorithm used to merge the outputs of multiple trusted elements' implementations must be efficient in terms of execution time and utility. Too strict algorithm will render the element useless, while the relaxed one might alter the security guarantees. Overall, the overhead introduced by employing N-version technique should not be significantly higher compared with a single version of trusted element execution.

Our main adversary consists of the potentially buggy or malicious code of a trusted element implementation. This implementation may try to output the sensitive user data as is without processing it but such a result will not be consistent across the outputs of all other implementations of the element, and will be ignored by the decision algorithm. We assume that various implementations of the same trusted element do not collude and are developed independently. We also assume that the software and hardware platform of the system where the trusted element executes are secure, and that IoT apps and TEs execute in sandboxed environments. It is not our primary goal to secure against side-channel attacks. The capabilities of the attacker consist only of the ability to write arbitrary code as part of trusted element implementations.

## 6.3 Trusted elements modules

In this section, we present a general security architecture for IoT systems based on N-version programming. This architecture relies on a set of *N-version-based trusted elements' modules* (henceforth called "modules"). A module provides the functionality of a single TE implemented internally in a N-version fashion, with each of the N versions being provided by independent developers. Each of these versions, called *units*, are required to implement the same *trusted element specification*.

Whenever an application issues a request, the inputs are forwarded to all N units and their outputs are compared with each other before a final output is returned back to the application. Deciding whether or not a final output result

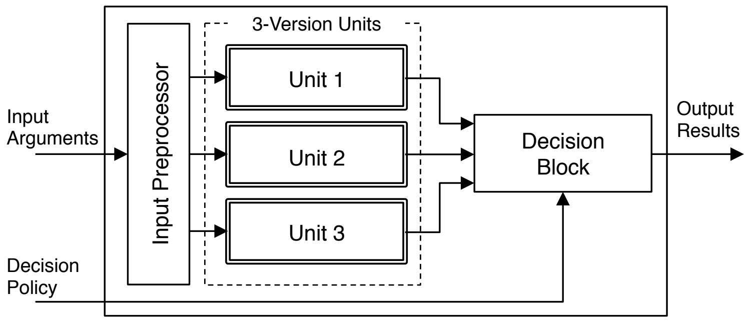


Figure 6.1 – N-version trusted element module (with N=3).

is provided and what that output result will be depends on a *decision policy* defined by configuration. In a particular policy, all N units must produce the same result, which is then returned as output result, otherwise, the application is informed that no result was generated. Thus, if any single implementation unit produces a malicious output, this output will differ from the remaining N-1 units (assuming no collusion) causing the final result to be suppressed, preventing the malicious unit from propagating its effects to the application.

Figure 6.1 shows the internals of a module implemented by 3 units. The input preprocessor feeds the input arguments to each unit and the decision block implements a decision algorithm according to the provided decision policy. The decision policy is a configuration parameter decided by the system administrator. Each unit is implemented by a program that runs in an independent sandbox. The input processor and the decision block logic must belong to the runtime system, which is also responsible for setting up the units' sandboxes and the data paths represented by arrows in Figure 6.1.

### 6.3.1 Module lifecycle

The lifecycle of each module comprises four stages. In the *specification* stage, a cooperation between the platform and community developers results in the production and public release of module TE specifications. The decision on the creation of new modules is based on the community needs. A specification features either the algorithm or high level function to be implemented, the input and output data formats, as well as a group of custom decision policies.

Once the specification is out, the module enters the *development* stage in which third-party developers independently implement their TE versions. This approach is similar to existing community-based software projects, e.g. Debian, where the members define task requirements and control the development process. Each TE version must be packaged and signed by the developer, and uploaded to the platform repository. By using a key that is certified by a certificate authority, it will be possible to assess the identity of the developer and prevent Sybil attacks, i.e., the same developer releasing and signing multiple malicious versions of the module's TE. Once authenticated the TE version is packaged in the TE module and subsequently either made available for users to install in case of a new module or automatically pushed for subsequent platform module update.

The next stage is *installation* of the module by the users. Users can download the latest version of the module from the repository and instantiate it locally at their IoT system. Default module settings work out of the box, however experienced users may add or remove module units, and redefine the decision policy according to their needs. Once the module is installed, the module enters the *execution* stage in which applications running on the hub are allowed to issue requests to the module. Note that modules may become temporarily out of service in order to perform software updates (e.g., installing a new unit or updating an existing one) and may also be permanently removed.

### 6.3.2 Detection of unit result divergence

The decision taking process is at the core of what makes N-version programming effective at countering adversarial units. In the perfect scenario, each unit is assumed to execute one of two possible versions: *benign* or *adversarial*. A version is benign if it consists of a flawless implementation of the module's trusted element specification. A version is adversarial if it deviates from the intended specification in order to tamper with or leak sensitive data. Thus, if deviations exist between unit outputs, then at least one adversarial version is present. Since different security properties can be attained depending on the number of units in agreement, we define three decision policies providing three agreement conditions:

**Total agreement (TA) policy**: This policy offers the strongest security guarantees. All $N$ units must agree on the same output result in order for an output to be returned. If this condition holds, the resulting value is returned, otherwise an error is yielded. Thus, 1 benign version only is required to exist in order to suppress the return of a corrupted result. In fact, for an attacker to be successful, all $N$ versions must be both adversarial and collude in producing the same output.

**Quorum agreement (QA) policy**: Only a quorum $Q = \lfloor N/2 \rfloor + 1$ units (i.e., a majority) needs to reach consensus on a common return value. If $Q$ is found, the module returns the agreed upon value, otherwise it reports failure. The QA policy is weaker than the TA policy because $Q > 1$ benign units need to be present to thwart an attack. Furthermore, a successful attack requires $Q < N$ colluding adversarial units.

**Multiplex (Mux$_i$) policy**: This policy is the weakest of all and can no longer be considered to provide N-versioning security benefits. Under a Mux$_i$ policy the decision block simply selects one unit output to be fed to the module output. The unit selection is parameterized by a number $1 < i < N$. This policy is useful mostly for debugging purposes during the testing stage of the module's lifecycle.

The divergence between unit outputs in a module occurs due to the rational behavior of a malicious developer who intentionally had not implemented some version according to the trusted element specification of the module. However, other causes may lead to undesired output divergence that may cause undesired side-effects, namely: software flaws, and module incoherence.

### 6.3.3 Nondeterministic inputs

One cause of unit divergence is *operational* and occurs whenever a specific trusted element depends on nondeterministic inputs, e.g., a random number, the system time or date, etc. If different units obtain different readings for the same intended input value, units' computations will likely return different results which may lead to failure in reaching a total or quorum agreement conditions and harm module's utility.

To avoid this problem, all nondeterministic inputs must be provided by the preprocessor. Sandboxes must prevent units from issuing nondeterministic system calls. If the version code depends on such calls, the input preprocessor can execute those upon request and pass the same value to all units. A request is declared by overriding the `init` method of the class of input parameters. The `init` method of this class is invoked by the input preprocessor and can be inherited by a subclass with the purpose of prefetching nondetermistic values. To prefetch an input value in a module, the trusted element specification only

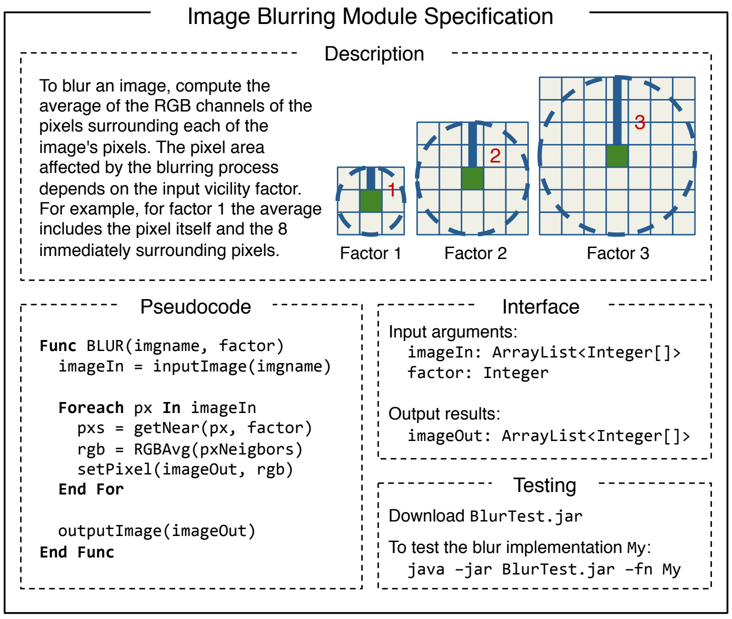


Image Blurring Module Specification

Description

To blur an image, compute the average of the RGB channels of the pixels surrounding each of the image's pixels. The pixel area affected by the blurring process depends on the input vicility factor. For example, for factor 1 the average includes the pixel itself and the 8 immediately surrounding pixels.

Pseudocode

```
Func BLUR(imgname, factor)
  imageIn = inputImage(imgname)

  Foreach px In imageIn
    pxs = getNear(px, factor)
    rgb = RGBAvg(pxNeigbors)
    setPixel(imageOut, rgb)
  End For

  outputImage(imageOut)
End Func
```

Interface

Input arguments:
```
  imageIn: ArrayList<Integer[]>
  factor: Integer
```

Output results:
```
  imageOut: ArrayList<Integer[]>
```

Testing

Download `BlurTest.jar`

To test the blur implementation `My`:
```
  java -jar BlurTest.jar -fn My
```

Figure 6.2 – Image blurring module specification.

needs to assign this subclass to the type of the respective input argument. By constraining all units to receive the same input, this approach prevents the aforementioned operational causes for divergence.

### 6.3.4 Software flaws

A second unintended cause for internal result discrepancy is *accidental* in nature, and is caused by flaws in versions' software that cause the actual unit execution to deviate from the expected value as defined in the trusted element specification. In addition to harming module utility, flaws may negatively affect the correctness of the module. As shown in past studies, programmers tend to commit the same flaws in the same code regions, which may end up resulting in the generation of incorrect results that can eventually appear at the module's output depending on how many units have reached consensus on the same incorrect value and on the decision policy in place.

To reduce these negative effects, we define a format for trusted element specifications that aims to be both unambiguous and human readable so as to reduce the change of software flaws. Figure 6.2 depicts a simplified version of the specification for an image blurring trusted element. The specification format comprises: a *description* of the intended functionality, an *algorithm representation* in the form of pseudocode, the *interface* of the module indicating the input and output parameters and respective types, and a *testing procedure* which may include specific testing code. While the description and the algorithm representation aim to clarify misunderstandings about the specification, the testing parts aim to help debugging. Since the specification is public, the source code of the testing classes and types of input arguments / output results must be provided.

### 6.3.5 Module incoherence

Module incoherence occurs if two or more units inside a module implement different trusted element algorithms. For example, a face recognition module may be based on software that implements face recognition using different techniques. As a result, one version may be able to identify a face that a second version cannot. Speech recognition is another example in which different algorithms may yield very diverse outputs, for instance being able to detect some words in a whole sentence, but not others.

A natural question that arises when dealing with the incoherent module is whether it can be used for countering malicious version implementations. In fact, even assuming the absence of software flaws, it will be difficult to determine whether the divergence of results is due to a malicious version or due to semantic differences between versions themselves. Faced by this challenge, we make two decisions.

First, we require the modules must be explicitly specified as *strict* or *loose*. A strict module is one in which all versions must implement the same algorithm. For this reason, all versions are expected to strictly implement the algorithm described by the trusted element specification. In contrast, a module is loose if the implemented algorithm does not satisfy the specification completely. Version developers must clearly indicate the type of a given version. Otherwise, installing a loose version on a strict module will cause internal unit output divergence thereby severely degrading the module utility.

Second, to improve the utility of loose modules, we allow replacing the standard decision algorithm of the decision block by a customized one (which could be provided along with the trusted element specification). Since the standard decision algorithm simply tests the equality of units' outputs, algorithms that generate slightly different outputs will immediately fail the test. To prevent this, a customized decision algorithm may perform domain-specific tests that may overcome small differences between outputs. The side-effect, however, is that by relaxing the equality requirement, an adversary may attempt to exploit that degree of freedom, e.g., to encode sensitive data for a remote party. Thus, by deciding whether or not to adopt a customized decision algorithm, an end-user can chose between the modules' utility and security.

Until now, we have presented an architecture for an IoT system based on N-version trusted element modules. To thwart adversarial versions, each module compares the results of output units and checks for total agreement or quorum agreement conditions depending on the decision policy chosen by the system administrator. We have also seen that the utility and security of each module can be affected by other factors, namely software flaws and module incoherence. The next sections focus on studying the impact of both these factors and on performance evaluation.

## 6.4 Impact of software flaws

In this section we study the impact of version software flaws on the overall behavior of modules. We specifically focus on strict modules performance. Since they implement the same algorithm, it allows us to concentrate on discrepancies due to software faults. For our study, we implemented several test strict modules that feature common privacy-preserving algorithms for a smart home sensor data.

### 6.4.1 Experimental methodology

We picked five different algorithms, and gathered three different implementations for each of them, with the help of five different volunteer developers. The versions for each algorithm were developed independently by different developers. For each developer, we provided a complete specification and a testing tool. The code was to be written in Java. Given the simplicity of the algorithms involved, we requested developers to submit their implementations before and after using the testing tool for debugging. While the implementations after testing recorded no bugs, the implementations before testing featured some bugs. Considering the purpose of this study, here we focus on the pre-testing implementations. The algorithms to implement were as follows:

**Image blurring algorithm:** An image blurrer can be used to protect users' privacy, namely by anonymizing the video data gathered by cameras (see Figure 6.2). Given an image file as input, this particular algorithm calculates the average of the RGB channels of the pixels in the vicinity of every pixel, and returns the correspondent blurred image file. We ran a simple battery test consisting of the blurring of 10 different pictures over vicinity factors of 1, 2 and 3. Afterwards, we made a byte-wise comparison between the expected result and the implementation produced files, in order to assess the implementations' correctness. In total, we executed 30 tests.

**Voice scrambling algorithm:** A voice scrambler can be useful in mitigating attempts to identify the speaker and other nearby individuals. This algorithm receives an audio clip as input, and after applying pitch shifting and distortion, it outputs a modified audio clip where the voice sounds robotized. With respect to testing, we exercised each implementation with 30 different audio clips.

**Data encryption algorithm:** RC4 is a stream cipher algorithm that can be used to encrypt certain sensor data before transmitting it. The algorithm receives a message and a key as input and returns the correspondent encrypted content. The final testing tool features 153K tests comprising tuples $\langle message$, $key$, $cyphertext\rangle$, where both $message$ and $key$ were randomly generated with increasingly longer sizes.

**Data hashing algorithm:** MD5 is a well-known hashing function useful in assessing the integrity of data, e.g., RC4 encrypted data received by the recipient. The algorithm takes a message as input and returns the hash of said message. The final testing tool featured 41K tests. These tests consist of tuples $\langle message, hash\rangle$, where every $message$ was randomly generated with increasingly longer sizes.

**K-anonymity algorithm:** Lastly, Mondrian is a top-down greedy algorithm for strict multidimensional partitioning, with the goal to achieve K-anonymity. Such an algorithm could be used in anonymizing IoT data (e.g., power consumption readings), so that the user could, for example, supply that information to an interested third party. With this approach, users' privacy would be kept, and the data would still be statistically useful. The algorithm receives an aggregation of tuples that represent users' data, the tuple indexes representing the quasi-identifiers, and a K-anonymity factor. The algorithm is then expected to output that same aggregation of tuples, this time broken

| **Module** | **Image Blurring** | | | **Voice Scrambling** | | | **Data Encryption** | | |
|---|---|---|---|---|---|---|---|---|---|
| | **V1** | **V2** | **V3** | **V1** | **V2** | **V3** | **V1** | **V2** | **V3** |
| **Tests Passed** | $\frac{30}{30}$ | $\frac{30}{30}$ | $\frac{30}{30}$ | $\frac{30}{30}$ | $\frac{0}{30}$ | $\frac{0}{30}$ | $\frac{153K}{153K}$ | $\frac{0}{153K}$ | $\frac{153K}{153K}$ |
| **Number of Bugs** | 0 | 0 | 0 | 0 | 4 | 4 | 0 | 1 | 0 |
| **N-mode Tests** | TA: ✓, QA: ✓ | | | TA: –, QA: ✗ | | | TA: –, QA: ✓ | | |

| **Module** | **Data Hashing** | | | **K-Anonymization** | | |
|---|---|---|---|---|---|---|
| | **V1** | **V2** | **V3** | **V1** | **V2** | **V3** |
| **Tests Passed** | $\frac{41K}{41K}$ | $\frac{41K}{41K}$ | $\frac{0}{41K}$ | $\frac{0}{210}$ | $\frac{210}{210}$ | $\frac{210}{210}$ |
| **Number of Bugs** | 0 | 0 | 1 | 1 | 0 | 0 |
| **N-mode Tests** | TA: –, QA: ✓ | | | TA: –, QA: ✓ | | |

Table 6.1 – Evaluation results of strict modules under total agreement (TA) and quorum agreement (QA) decision policies with the output defined as correct (✓), incorrect (✗), or silent (–).

down into partitions satisfying the K-anonymity factor for the quasi-identifiers inputted. The testing tool features 210 tests. These tests comprise tuples $\langle dataTuples, k, qids, result\rangle$, where $dataTuples$ are statically grouped in 5 files each comprising 1 million entries, and $k$ and $qids$ are automatically generated and increased anonymity factors and quasi-identifiers respectively.

## 6.4.2 Main findings

Table 6.1 summarizes the N-version study results, where V1, V2 and V3 correspond to three different version implementations. We highlight three main findings. These results show, on the one hand, whether each single implementation has passed all the verification tests, and on the other hand, whether multiple implementations successfully passed the tests when executed in N-version mode according to two possible decision policies: *total agreement* or *quorum agreement*. We highlight three main findings.

First, under the TA decision policy, only the image blurring module yields an output. This is possible because all unit implementations passed the 30 tests. Since they produced the same result, the TA policy concurs on outputting the same result. This finding is consistent with the lack of bugs found in the code which could compromise the resulting output. For the remaining modules, however, faults have caused some versions to fail individual tests thus undermining the overall result.

Second, under the more relaxed QA decision policy, we observe that four modules can successfully reach a consensus and produce an output: the image blurring module—whose individual implementations output consistent results—and three additional modules in which two out of three implementations generate the same result, thereby allowing a consensus to be reached. In these cases, functional divergence occurred due to the existence of bugs. In the data encryption module, we identified a bug in V2 that consisted of a wrong value swap between two variables. Regarding the data hashing module, we detected one bug in V3 which was later found to be a variable poorly initialized. In the K-anonymization module, V1 contained a coding error stemming from a wrong pseudocode interpretation of the scope of a variable. Specifically, a global variable used by several functions was supposed to be initialized in a certain function, but V1's developer declared the variable as local to that function, leading to issues in the other functions handling it. Lastly, in one case, the voice scrambling module produced an incorrect response under QA. This happened because two versions, namely V2 and V3 experienced the same 4 bugs each. More specifically, the bugs originated from the wrong interpretation of a loop upper bound.

Given these numbers, we conclude that when versions yield different results, NVP actually detects (except for side-channels) implementation deviations created with rational intent. The exception being when the majority of the versions output the same erroneous result. Although this happened in the case of the voice scrambler module test, in practice it is not likely to happen since faulty module units will be rejected during the testing stage of the module's lifecycle. Accidental mistakes can cause a reduction in the utility of the module. If a very conservative decision policy is employed (TA) this loss will be considerable (up to 80%). On the other hand, under QA, the utility drop is smaller, as four out of five modules can still produce the same result.

The influence of bugs or intentional specification violation can be minimized or completely eliminated with the extensive testing of all the versions before they are incorporated as modules' units. Similar approach is used in open-source communities, e.g. Debian, where new applications' versions go through several testing stages before making their way to the stable OS release.

## 6.5 Impact of module incoherence

This section studies the impact of units incoherence on the modules' overall behavior and utility. For our study, we implement test loose modules which do not follow the same specification, yet implement the same high level function: *face recognition* and *speech recognition*.

The module implementing the face recognition element uses three existing open source face recognition libraries as building blocks: OpenCV (with Face module) [58], OpenBR [143], and OpenFace [29]. The libraries code remained unchanged but was wrapped around the N-version module's API. Based on these libraries, we defined several module configurations. We tested the effectiveness of the face recognition module when trained with a training set of 2250 images and a testing set of 250 images. In total, we trained the recognition of 250 different people with 9 pictures each. All these images where extracted from the UFI dataset [154]. Microsoft Face API was used as state of the art face recognition implementation. It was trained and tested using the same dataset.

The speech recognition module uses three independent speech recognition libraries—Sphinx [151], Julius [153], and Kaldi [184]—and was also tested in different module settings. Every configuration was exercised with 130 sentence tests from CMU's AN4 speech recognition dataset [119]. AN4 features almost 50 minutes of speech with both male and female speakers, in a total of 948 utterances averaging 3 seconds in duration each. As with face recognition libraries, we developed an API wrapper for all the speech recognition libraries. We use Google Speech API as state of the art speech recognition system which requires no training.

### 6.5.1 Face recognition module study

Table 6.2 presents the success rate of our tests for the three face recognition versions evaluated individually, and the representative three module configurations, namely total agreement, quorum agreement and an intersection of the two versions that showed the best recognition results.

The first important observation is that the efficacy of the open source libraries is smaller than Microsoft Face's, which reaches 99% success rate. OpenCV stands out as the least effective library (only 62% success rate). The difference between OpenCV and OpenBR stems from the algorithms they im-

| | | OpenCV | OpenBR | OpenFace | MS Face API |
|---|---|---|---|---|---|
| Recognition | ✓ | 156 (≈62%) | 219 (≈88%) | 228 (≈91%) | 249 (≈99%) |
| | ✗ | 1 (≈1%) | 1 (≈1%) | 0 (0%) | 0 (0%) |
| No Recognition | | 93 (≈37%) | 30 (≈11%) | 22 (≈9%) | 1 (≈1%) |
| Total | | 250 (100%) | 250 (100%) | 250 (100%) | 250 (100%) |

| | | Decision Policy | | |
|---|---|---|---|---|
| | | Total Agreement | OpenFace ∩ OpenBR | Quorum Agreement |
| Recognition | ✓ | 137 (≈55%) | 202 (≈81%) | 220 (88%) |
| | ✗ | 0 (0%) | 0 (0%) | 1 (1%) |
| No Recognition | | 113 (≈45%) | 48 (≈19%) | 29 (11%) |
| Total | | 250 (100%) | 250 (100%) | 250 (100%) |

Table 6.2 – Success rates of face recognition measured in correct (✓), incorrect (✗) and no recognition.

plement, namely Eigenfaces and 4SF respectively. The small difference between OpenBR and OpenFace comes as a surprise, given that OpenFace implementation uses neural networks for face recognition, theoretically more effective than OpenBR's 4SF.

Table 6.2 then shows the success rate for three face recognition module configurations. Configuration *total agreement* consists of a module that employs all three libraries—OpenCV, OpenBR, and OpenFace—and yields "success" if and only if all libraries identify the same individual. We can see that the face recognition accuracy drops considerably to only 55%, which is explained by the significant differences that exist between the algorithms implemented by each library.

In a second configuration, we used only two libraries – OpenFace and OpenBR – and in this case the success rate increased substantially to 81%. The best results were achieved when we used three libraries side by side, but with a merging policy function that outputs success every time at least two libraries produce the same results. In this particular configuration (*quorum*), the success rate reaches 88%, which represents a reduction of only 3% when compared to OpenFace alone.

| Implementation | Sphinx | Julius | Kaldi | Google |
|---|---|---|---|---|
| **Sentence Match** | $\frac{20}{130}$ (≈15%) | $\frac{36}{130}$ (≈28%) | $\frac{88}{130}$ (≈68%) | $\frac{103}{130}$ (≈79%) |
| **Word Intersection** | $\frac{578}{902}$ (≈64%) | $\frac{570}{902}$ (≈63%) | $\frac{719}{902}$ (≈80%) | $\frac{722}{902}$ (≈80%) |

Table 6.3 – Speech recognition confidence.

Considering these results, we argue that the best mechanism in merging face recognition results in an N-version setting is to gather the majority of the results given by a module's units. Note, however, that result intersection is not always a sound solution. If we consider the case where a module has fewer honest units than intentionally ineffective ones, e.g., units that produce wrong results with the goal of preventing face recognition, then the success and consequent effectiveness of the module is compromised. To address this issue, we believe a reputation based approach for unit selection could be used.

### 6.5.2 Speech recognition module study

Although, word error rate (WER) is the metric generally used to measure the accuracy of speech recognition, it cannot be applied to the situation where there are multiple recognition results. Moreover, in a smart home scenario, voice commands can still be interpreted correctly even if some words are not recognized or come in a wrong order. We, therefore, opted for a sentence match and word intersection merging functions as the main performance parameters for speech recognition modules.

Table 6.3 shows the results for each library evaluated based on two criteria: *sentence match* and *word intersection*. Sentence matching consists of the exact match between the entire original sentence and the recognized result returned by each library. Word intersection counts the number of words that exist in the original sentence and are also present in the recognition results returned by the library (902 is the total number of words present in all sentences). Across both these dimensions, Sphinx and Julius clearly fall behind Kaldi, which offers the highest success rates (68% sentence match and 80% word intersection). At the same time, Kaldi' numbers are not far off Google Speech's.

Table 6.4 lists multiple module configurations that we used to produce speech recognition units based on these libraries. Each entry of the table cor-

| **Decis. Policy** | **TA** | **Sphinx ∩ Julius** | **Sphinx ∩ Kaldi** | **Julius ∩ Kaldi** | **QA** |
|---|---|---|---|---|---|
| **Sent. Match** | $\frac{13}{130}$ (≈10%) | $\frac{13}{130}$ (≈10%) | $\frac{19}{130}$ (≈15%) | $\frac{34}{130}$ (≈26%) | $\frac{40}{130}$ (≈31%) |
| **Word Inters.** | $\frac{455}{902}$ (≈50%) | $\frac{455}{902}$ (≈50%) | $\frac{554}{902}$ (≈61%) | $\frac{557}{902}$ (≈62%) | $\frac{666}{902}$ (≈74%) |
| **Word Union** | $\frac{753}{902}$ (≈83%) | $\frac{706}{902}$ (≈78%) | $\frac{745}{902}$ (≈83%) | $\frac{735}{902}$ (≈81%) | $\frac{753}{902}$ (≈83%) |

Table 6.4 – N-version speech recognition confidence.

responds to a specific module configuration. The columns indicate which libraries constitute the units of the module, and the lines indicate the merging function that was used to produce a successful speech recognition output. We adopted three merging approaches: *sentence match*, which is similar to the criteria used for the individual solutions and issues an output if all units identified the same sentence; *word intersection*, which returns only the words that all units identified successfully; and *union*, which returns the union of all words identified by all units.

As shown in Table 6.4, *sentence match* tends to yield very poor results, displaying a success rate between 10% and 26% between any pair of units. Even when we consider quorum agreement, i.e., when at least two out of three units return the same result, the success rate only reaches 31%, which is very far from Kaldi's 68%. Still, given that most voice controlled devices, e.g., Amazon Echo, use a grammar based approach, in which they ask users to repeat unrecognized words, exact sentence match is an unreasonable metric.

With word intersection, the results improve significantly up to 62% between any pair of units, and up to 74% when we consider the quorum for the results produced among them. Because of the intersective nature of the merging functions *sentence match* and *word intersection*, the adoption of an increasing number of units does not necessarily yield better results. This happens because the overall success rate is always bound to the performance of the worst unit. This can be seen in the last column of the table. For instance, although the pair Julius and Kaldi yields a 62% success rate for the *word intersection* function, the addition of Sphinx bounds the three units overall success to the result yielded by the worst Sphinx pairing result, i.e., the result of the pair Sphinx and Julius (50%). The table also shows that for this type of functions the best approach is to use a quorum policy, i.e., the consensus between at least two units, which yielded success rates of 31% and 74% for *sentence match* and *word intersection* respectively.

Overall the highest success rate is achieved when word union is employed. As can be seen in the table, the function *word union* yields success rates of at least 78%, and 83% in the best case, surpassing even Google Speech. Contrary to *sentence match* and *word intersection*, the success rate of this function is the same for the combination of all three units and the quorum consensus (83%). This happens because quorum also implies the output of all three units. As a result, both functions produce the same output.

Still, we argue that union is not a fair result merging function for two reasons. On one hand, semantically, the union of the output of two or more speech recognition units may differ significantly from a speech recognizer expected result. On the other hand, this union function can potentially endanger the privacy of the user. For instance, as long as there is one rogue unit that extracts information from the audio source, e.g., a voice detector that derives the number of people in the room based on the background sound, the whole module could be compromised, as its result would feature that information.

Finally, we can make three conclusions: (1) exact sentence match is a poor N-version result merging function for a speech recognition case, (2) word intersection recognition success rates are limited by the worst unit, but are reasonable when used in a quorum consensus approach, and (3) although word union success rates are the highest among the configurations studied, its semantics and privacy limitations render it unusable in merging N-version results.

Consequently, we argue that quorum-based word intersection is the best approach of the three in merging this type of results. Similarly to the face recognition case, it can also be complemented with a reputation based approach, in order to address the issue of the intentionally ineffective sub-modules.

## 6.6 Performance evaluation

This section aims at assessing the performance overhead introduced by our proposed N-version approach as opposed to running a single instance of an element implementation. Specifically, we present performance measurements for both strict and loose modules, as well as the performance of the result merging approaches used.

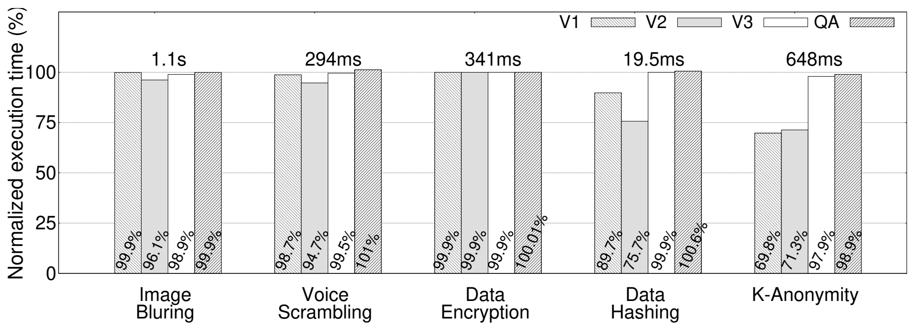


Figure 6.3 – Strict modules performance

## 6.6.1 Experimental methodology

The performance evaluation comprises the execution time measurements of each of the aforementioned N-modules. These measurements feature the execution time of each individual modules' units, and the execution time of the quorum and total agreement merges. Each of these measurements consisted of computing the average of 50 tests, each with the same input. More specifically, we chose a 1280x720 pixel image and a factor of 2 for the image blurrer; a 10 second voice clip for the voice scrambler; a randomly generated 256-byte key and 1MB plaintext for the data encryption module; 1MB worth of randomly generated text for the data hashing module; and a set of 100000 tuples and a K-anonymity of 500 for the K-Anonymization module. For the face recognition module, we provided a training dataset of 150 pictures of three different people, and an additional picture as test input; and for speech recognition, we provided a general acoustic and custom language models as knowledge base, and a voice clip as input. The experiments were conducted on a laptop equipped with an Intel i3-3217U 1.80GHz CPU and 4GB of RAM.

## 6.6.2 Main findings

Figures 6.3 and 6.4 present the performance results for strict and loose modules respectively. For a matter of consistency, we use the TA policy as baseline. Note that the most significant performance differences among the different strict modules' units relate to either ineffective loop implementations, or recurrent use of data type casts. However, for the loose modules, the main performance difference stems from units' underlying algorithms diversity and efficiency of their implementations.

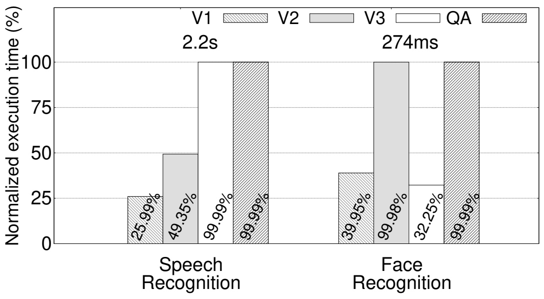


Figure 6.4 – Loose modules performance

The first finding is the confirmation that the parallel execution nature of our approach bounds the two merging approaches' execution times to the slowest unit's execution time. This is most evident for the strict K-anonymization V3 unit. For loose modules the difference between unit execution times is even more noticeable. For the speech recognition module, V1's execution took a quarter of the time needed to execute V3. The same is observed for the face recognition module, where V3 outperformed V2.

Secondly, there is a significant execution time difference between loose module units. Note again that loose modules rely on heterogeneous versions. As a result, the underlying algorithms of units and their complexity may vary, leading to performance differences. Unlike strict modules, where the performance of units is usually similar, the impact of the slowest units on loose modules' performance is higher.

The third finding relates to the cost of the merging approaches. While we defined the TA policy as baseline to compare the performance of the three units and merging approaches, we can see that quorum agreement is sometimes more expensive than total agreement. This happens because, total agreement implies at most two comparisons, i.e., between V1 and V2, and between V2 and V3, while quorum agreement, in the worst case, requires three comparisons to yield a result. On the other hand, in the best case, quorum agreement can be achieved with one comparison only.

## 6.7 Discussion

Traditionally, NVP has raised two main objections. First, N-version is regarded as a technique demanding significant human resources to implement the N different software versions. However, considering our targeted scenario, this concern may be alleviated by relying on open source communities for the development of TE implementations. In fact, such communities have shown good results in maintaining large scale projects, e.g., Debian packages, python modules, and IoT specific ones, e.g., apps and automation recipes.

A second objection to NVP is the connotation of poor failure diversity among independent versions. With this respect, it has been shown [144] that statistically, the number of common errors is relatively low and the diversity of implementations makes the overall system robust to failures. Therefore, it is hard for an adversary to exploit a common flaw across all the N-version modules. Although at a small scale, our software flaw study seems to confirm this idea, since in five different TEs, common flaws occurred only once. Even so, although this occurrence was detected by simple debugging tools, another reason behind it could be our specification effectiveness, which was not experimentally tested. Nevertheless, NVP considerably raises the bar for adversaries since the number of latent vulnerabilities would be smaller compared to single version executions.

Our approach's open source nature may also hinder TE utility, as the number of naive or malicious TE units outputting incorrect results may be higher than that of correct units. We propose two approaches to address this issue. First, a TE developer reputation scheme could provide insights regarding the effectiveness and quality of a TE unit. This information could then be used to filter unwanted units when packaging modules. Second, at least for loose modules, their effectiveness could benefit from commercial software, which from our experience, requires little adaptation effort with our approach. Similarly to app markets for popular smartphone platforms, such approach may introduce the required diversity of the implementations and open a way for developers to get rewarded for their efforts.

Performance wise, the QA policy's positive results seem to suggest that the impact of the slowest unit for both loose and strict modules can be eliminated by taking advantage of unit redundancy. Instead of waiting for the slowest unit to finish, the decision block may process unit outputs up until a majority is formed. This approach addresses the performance problem and provides a reasonable tradeoff between module performance and user privacy.

As for malicious behavior it is not in our scope to prevent malicious application attacks. This holds true for both attacks targeting system security mechanisms, e.g., sandboxing, and TE module security, e.g., bug exploitation by sending crafted inputs to modules. Nevertheless, to address TE module security, our design could be complemented with unit address space randomization techniques [77].

Another attack our approach does not prevent is a DoS in which a malicious version of a given N-version module consistently outputs incorrect results thus affecting the overall result. However, such an attack could be easily eliminated by detecting and replacing a faulty version with another one.

## 6.8 Related work

NVP [70] has originally been introduced to reduce the likelihood of error and bugs in the software development. Multiple independent teams of programmers developed several versions of the same software and then ran these implementations in parallel. The diversity of the implementations helped to survive some of bugs introduced during development and as a result improve overall reliability and fault-tolerance of the software. Since then, NVP has been used in several fields.

**Software maintenance:** Veeraraghavan et al. [215] proposed multiple replicas of a program to be executed with complementary thread schedules to identify and eliminate data race bugs that can cause errors at runtime. DieHard [50] used randomized heap memory placement for each replica to protect the software from memory errors, e.g. buffer overflow or dangling pointers. Imamura et al. [131] applied N-version programming in the context of genetics to reduce the number and variance of errors produced in genetic programming. Some systems [63, 110], applied N-version to the process of updating software, in order to detect and recover from errors and bugs introduced by the new versions. While these approaches consider only one developer of multiple software versions, we assumed multiple independent developers and versions.

CloudAV [172] provided antivirus capabilities as a network service and leveraged NVP to achieve better detection of malicious software. However, nothing prevented it from exploiting private user data. Demotek [115] employd N-version to enhance the reliability and security of several components comprising an e-voting system. Still, it assumed the modules were honest, and its main goal was to make it difficult for an attacker to compromise the whole system. Overall, none of the aforementioned systems relied on N-version to bootstrap trust in system components, focusing instead on improving individual modules' reliability and availability.

**Attacks detection:** Additionally, NVP has been used to detect and prevent system security attacks such as inadvertent memory access [77, 194]. This, however, required a custom memory allocation manager and modifications to the OS kernel. Moreover, these systems trusted multiple versions of the same software and assumed only the input data to be potentially malicious. In our case the input is by default trusted since it consists of the sensor data collected by the smart home devices. The implementations of various N-version components are, however, untrusted and may act maliciously.

**Privacy protection:** NVP has also been leveraged to ensure personal information confidentiality and prevent information leaks. Most of these systems employed techniques in which two replicas of the same software were executed with different inputs [226], under different restrictions [65] or on different security levels [84]. To the best of our knowledge, our work is the first to study the feasibility of NVP in securing IoT platforms.

## 6.9 Summary

Dataflow element-based programming model is an essential part of all three systems we have described in the previous chapters. It relies on a rich set of trusted API elements which can be used to build IoT apps. These trusted elements are expected to operate correctly and perform a desired function, be it speech recognition or object detection. The whole trust model depends on the *trustworthiness* of these *trusted* elements. However, considering that they are provided and maintained by the open community of developers, ensuring trust in the elements' implementations becomes a challenging task. In this chapter, we looked into one way to bootstrap trust in these elements.

By using an N-version programming approach multiple versions of the same element provided by independent developers run side by side to produce a final result. The final result is calculated based on the total or quorum agreement between the individual versions' results. As long as the majority of versions agree on the same results, we can ensure the correct behavior of the trusted element. Even if there is one malicious element implementation, trying to exfiltrate sensitive sensor data, it will not succeed since its actions and results will be in contrast to the results of the other legitimate implementations.

We performed an in-depth study of NVP ability to protect the privacy of sensor data in IoT environment. The results were quite encouraging, showing that NVP can be viable option provided that the developers of different versions do not collude. We determined that the selected decision policy plays an important role in NVP module's performance and the quality of the final module's results. While for some strictly-specified modules, the decision policy expects at least the majority of the results to have an exact match, for loosly-specified modules the decision policy must be relaxed to accommodate for potential results divergence. Performance-wise the quorum agreement policy proved to be the most efficient one since it allowed to converge on the final result without waiting for all module's versions to finish processing. With its little impact on user experience this policy can be effectively used in IoT apps.

In the next chapter we will look into another way to provide privacy and security guarantees in the IoT environment. This time, however, we will concentrate on the IoT devices' software components and their ability to withstand fault-injection attacks. Such attacks aim to modify the expected software behavior by strategically placing a fault in its execution context. The fault itself can be as simple as a bit flip or a more complicated as the one changing the target of a particular branch instruction to a different value. In both cases, the resulting software can either crash and fail to operate normally, or, what is worse, perform an unexpected and highly undesirable action, e.g. transmit sensor data in plaintext or leak an encryption key. We will review common software hardening techniques that aim to detect and prevent fault-injection attacks and evaluate their efficiency and performance impact.

Chapter 7

# IoT software hardening analysis

## 7.1 Introduction

Millions of people worldwide use Internet of Things (IoT) devices, such as smart lights, door locks and watches, to enhance their households and have more control over their daily lives. The nature of the data these devices operate with is extremely personal and sensitive, ranging from door lock state updates to heart rate measurements. In order to preserve the end-users privacy, these devices usually encrypt the data before transmitting it (e.g., to a local hub, mobile phone or cloud server). However, a single fault in the encryption logic, introduced either accidentally (e.g. electromagnetic glitch) or intentionally by a malicious attacker, may cause sensitive data leaks (see Figure 7.1).

To make the devices more resistant to faults, IoT device manufacturers may choose to harden the software components by adding a safety logic, that aims to detect the presence of faults and minimize their impact. The hardening can be applied to a hardware on which a given software runs [207] or to the software itself [107, 182]. While there is a variety of hardening techniques, in practice, IoT software developers rarely have a clear understanding of the real impact of a chosen hardening technique on their software's fault-tolerance and performance. In fact, some of the hardening techniques may have a negative impact and actually increase the software vulnerability [166, 196, 201].

In this chapter, we present a thorough analysis of five common software-based hardening techniques applied to an implementation of PRESENT – a lightweight block cipher intended to be used in low-power resource-constrained

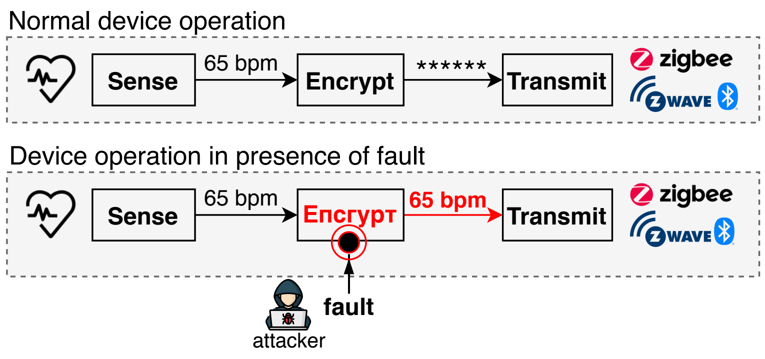


Figure 7.1 – Heart rate monitor operation in normal mode and under attack.

IoT devices. We evaluate the effectiveness of the hardening techniques on three fronts: (1) we start from evaluating their ability to prevent sensitive data leaks in presence of faults; then, (2) we study their general fault-tolerance and analyze the impact of each fault type; finally, (3) we measure the impact of hardening techniques on software performance and binary size.

To facilitate the analysis we have developed Chaos Duck – a tool for automatic software fault-tolerance evaluation. Without any intervention from the developer Chaos Duck injects faults in a given software and evaluates their impact on security and performance. It supports six different fault types ranging from bit flip to branch faults, and explores all the possible fault locations.

## 7.2 Case study: PRESENT

Securing IoT devices is necessary now that more and more of those are being used in private, secure, or mission critical environments. This section introduces PRESENT, the encryption algorithm used as the case study in this work.

PRESENT [55] is a block cipher that was specifically developed for low-power resource-constrained IoT devices, that due to their hardware constraints cannot use conventional AES cipher. In our case study, we use a canonical size-optimized version of PRESENT implemented in C with an 80-bit key[1].

A high-level overview of PRESENT algorithm and how it is used in a heart rate monitor is shown in Listing 7.1. Each of the 31 encryption rounds consists

[1]http://www.lightweightcrypto.org/implementations.php

```
encrypt(state,key) {
  int round = 0;
  while(round < 31) {
    addRoundKey(state,key);
    sBoxLayer(state);
    pLayer(state);
    round++;
  }
  addRoundKey(state,key);
}
reportcycle() {
  state = sense();
  key = {0xd3, 0xe4 ... 0xba};
  encrypt(state,key);
  transmit(state);
}
```

Listing 7.1. A pseudocode of a heart rate monitor's software using PRESENT.

of an XOR operation to introduce a round key using S-box and permutation layers. After that, an additional operation performs a final key XOR.

A heart rate monitor runs a regular report cycle during which it obtains a new heart rate value (i.e. state), encrypts it with a hardcoded key and transmits it to an external receiver (e.g., a mobile phone).

## 7.3 Fault injection attacks

IoT devices are often exposed to fault injection attacks that aim to challenge device robustness and security [126, 103]. This section overviews how such faults can occur and characterizes them with fault models used in this work.

Fault injections can be achieved by introducing faults either via hardware [123, 140, 148] or software [118, 134]. The impact of a fault on the device's behavior varies considerably, ranging from no effect, to software crashes, or security vulnerabilities. For instance, a simple power drop, i.e. a *glitch*, may cause data corruption or loss. Similar glitches may result in a weaker data protection by disrupting the encryption logic of the device firmware.

In this work we focus on faults that may introduce a security vulnerability in the device software causing the leakage of information that was meant to be secret. These faults are particularly dangerous when injected in software components implementing encryption algorithms.

To illustrate a fault injection that leads to a vulnerability, consider the assembly code shown in Figure 7.2. This code was generated by compiling the C implementation of PRESENT for the ARM architecture. A while loop executes 31 rounds of encryption checking the value of the `round` variable at each round, as shown in Listing 7.1. A check for `round < 31` is performed at address `c34`, followed by a conditional branch instruction at `c38` which restarts the loop if the condition holds. Otherwise, the execution continues to

ARM assembly code (extract)

```
00000668 <encrypt>:
 */ ... /*
     while(round<31) {
 718:  ea000144  b    c30 <encryption+0x5c8>
 71c:  e3a03000  mov  r3, #0
 */ ... /*
        round++;
 c24:  e55b3007  ldrb r3, [fp, #-7]
 c28:  e2833001  add  r3, r3, #1
 c2c:  e54b3007  strb r3, [fp, #-7]
    while(round<31) {
 c30:  e55b3007  ldrb r3, [fp, #-7]
 c34:  e353001e  cmp  r3, #30
 c38:  9afffeb7  bls  71c <encryption+0xb4>
    }
    addRoundKey(state,key);
 c3c:  e3a03000  mov  r3, #0
 c40:  e54b3005  strb r3, [fp, #-5]
 */ ... /*
```

ARM assembly code with branch fault at 0xc38

```
00000668 <encrypt>:
 */ ... /*
     while(round<31) {
 718:  ea000144  b     c30 <encryption+0x5c8>
 71c:  e3a03000  mov   r3, #0
 */ ... /*
        round++;
 c24:  e55b3007  ldrb  r3, [fp, #-7]
 c28:  e2833001  add   r3, r3, #1
 c2c:  e54b3007  strb  r3, [fp, #-7]
    while(round<31) {
 c30:  e55b3007  ldrb  r3, [fp, #-7]
 c34:  e353001e  cmp   r3, #30
 c38:  9affffff  bls   c3c <encryption+0x5d4>
    }
    addRoundKey(state,key);
 c3c:  e3a03000  mov   r3, #0
 c40:  e54b3005  strb  r3, [fp, #-5]
 */ ... /*
```

Figure 7.2 – Example of a branch instruction fault attack.

the next instruction at `c3c` and runs a final key XOR. A specific fault injection could modify this last branch instruction to alter the function behavior. For instance, by changing the target of a branch instruction at `c38` from `71c` to `c3c` the function will perform just 2 rounds of encryption leaving the program vulnerable to a differential key recovery attack [53]. Similar faults may cause a program to skip the encryption procedure entirely by modifying the target of other branch instructions.

Injecting faults via hardware is relatively difficult and requires expensive specialized hardware [44, 164]. The reproducibility and accuracy of such injections can be low depending on the fault type. Injecting faults on a software level requires fewer resources and offers higher accuracy. We therefore concentrate on software fault injection which allows us to simulate faults at the exact locations of a program's binary. By inspecting all the potential fault locations and analyzing their impact on program behavior we can measure the program's fault tolerance. Below we outline the faults considered in our study.

**Branch faults:** The goal of branch instruction faults (both unconditional *B* and conditional *BC*) is to disrupt the control flow graph of a given program in a way that is beneficial for an attacker (e.g., skip the encryption or fault detection logic). This can be achieved by modifying the target addresses of branch instructions to point to different locations in the program's address space.

**Bit flip faults:** The *FLP* fault operates on instruction bits and simulates flipping a single bit. This type of fault often occurs naturally (due to radiation or electromagnetic activity) or can be injected manually using specific software [142, 189].

**NOP faults:** The *NOP* faults replace an instruction at a given address with a `nop` instruction, effectively skipping the replaced instruction. This kind of modification may lead to small affects like skipping a variable assignment, or more serious ones like skipping a function call.

**Zeroing faults:** The *Z1B* and *Z1W* faults set a single byte or word respectively to zero. This kind of faults is more likely to be related to hardware effects like an EMP of the memory or bus, and is most effective when targeting values used in program logic, e.g. the number of encryption rounds [105, 113].

## 7.4 Hardening techniques

One approach to address fault injection vulnerabilities is to use various hardening techniques that aim to detect and minimize the consequences of incorrect program behavior. Hardening techniques may be implemented either via hardware [207, 212], software [107, 182], or both [197]. In this work, we focus on software hardening techniques that allow us to detect the faults at runtime and prevent erroneous program execution. To this end, we selected five state-of-the-art techniques commonly used to harden the software implementations of cryptographic algorithms in embedded systems [45, 208].

Below we outline the key concepts of these techniques applied to the implementation of PRESENT as illustrated in Listing 7.1. For extensive discussion of each, we refer to the cited works.

**Classic Loop Hardening (CLH).** This technique has been widely discussed [43, 57, 71, 80, 88, 186] and relies on duplicating the loop iteration counters and exit conditions forcing a second check at loop exit (see Listing 7.2). The rationale behind this technique is as follows: if an injected fault corrupts the main loop counter the duplicated counter will still hold the correct value and will signal an error on exit condition check. We extend this technique further by once again verifying all the duplicated loop counters at the end of each code block. This is particularly important in case of block cipher implementations that often include functions consisting of multiple *for* or *while* loops.

```
encrypt(state,key) {
  int round = 0, round_dup = 0;
  while((round < 31) &&
  (round_dup < 31)) {
    addRoundKey(state,key);
    sBoxLayer(state);
    pLayer(state);
    round++; round_dup++;
  }
  if (round!=round_dup) error();
  addRoundKey(state,key);
}
```

Listing 7.2 – Classic loop hardening (CLH) technique.

```
encrypt(state,key) {
  int round = 0, round_dup = 0;
  while(round < 31) {
    addRoundKey(state,key);
    sBoxLayer(state);
    pLayer(state);
    if(round!=round_dup)error();
    round++; round_dup++;
  }
  if (round!=round_dup) error();
  addRoundKey(state,key);
}
```

Listing 7.3 – Variable duplication (VD) technique.

**Variable Duplication (VD).** This technique implements redundancy on variable level [71, 190]. Each variable is duplicated and both copies are modified in the same manner (see Listing 7.3), i.e. every write operation performed on the original variable is also performed on its copy. At each read operation the copies are compared for consistency: if the values do not match an error is raised. Unlike the *CLH* technique which concentrates on loops and only checks the counter variables once, *VD* performs this check every time any variable is updated or used in conjunction with another variable.

**Statement Counters (SC).** This hardening technique (with minor alterations) has been previously proposed by several authors [27, 71, 150, 175]. It relies on counters that are incremented and checked against the expected value after executing each source code block (i.e. a function, a loop, or even a single statement). This allows detection of attacks that disrupt control flow of the program, e.g., by modifying the target of branch instructions, since the maliciously modified branch target would be executed in the unexpected order. We implement the variation of this technique proposed by Lalande et al. in [150] which suggests a per-statement counter granularity for a better CFG control (see Listing 7.4). In this case, the attack will be detected if any of the two adjacent statements in the source code are not executed in the right order. There are also additional counters for function calls, *for* / *while* loops or *if* / *else* blocks.

**Function Duplication (FD).** With this technique all the sensitive program functions are duplicated and operate on the same inputs, but their outputs are stored in different variables [42] (see Listing 7.5). These variables are com-

```
#define DECL_INIT(cnt,x) int cnt; if((cnt=x)!=x) error();
#define CHK_INC(cnt,x) cnt=(cnt==x ? cnt+1 : error());
#define RESET_CNT(cnt_while,val) (cnt_while==1||cnt_while==val) ?
 cnt_while=1 : error();
#define CHK_LOOP_INC(cnt_loop,x) (cnt_loop==x)?cnt_loop+=1:error();
#define CHK_LOOP_END(cnt_loop,val) if (cnt_loop!=val) error();
```

```
encrypt(state,key) {
  DECL_INIT(enc_cnt,1);
  CHK_INCR(enc_cnt,1);
  int round = 0;
  CHK_INC(enc_cnt,2);
  DECL_INIT(while_cnt,1);
  CHK_INC(enc_cnt,3);
  DECL_INIT(loop_cnt,0);
  CHK_INC(enc_cnt,4);
  while(round < 31) {
   RESET_CNT(while_cnt,6);
   CHK_LOOP_INC(loop_cnt,round);
   CHK_INC(while_cnt,1);
   addRoundKey(state,key);
   CHK_INC(while_cnt,2);
   sBoxLayer(state);
   CHK_INC(while_cnt,3);
   pLayer(state);
   CHK_INC(while_cnt,4);
   round++;
   CHK_INC(while_cnt,5);
  }
  CHK_INC(enc_cnt,5);
  CHK_LOOP_END(loop_cnt,31);
  CHK_INC(enc_cnt,6);
  addRoundKey(state,key);
  CHK_INC(enc_cnt,7);
}
```

Listing 7.4. Statement counters (SC) technique.

```
reportcycle() {
  state = sense();
  key = {0xd3, 0xe4 ... 0xba};
  for (int i=0; i<8; i++) {
    copy[i] = state[i];
  }
  encrypt(state,key);
  encrypt_dup(copy,key);
  for (int i=0; i<8; i++) {
   if(state[i]!=copy[i])error();
  }
  transmit(state);
}
```

Listing 7.5 – Function duplication (FD) technique.

```
reportcycle() {
  state = sense();
  key = {0xd3, 0xe4 ... 0xba};
  for (int i=0; i<8; i++) {
    copy[i] = state[i];
  }
  encrypt(state,key);
  decrypt(state,key);
  for (int i=0; i<8; i++) {
   if(state[i]!=copy[i])error();
  }
  transmit(state)
}
```

Listing 7.6 – Decryption at place (DaP) technique.

pared on function exit: if the resulting values are different a program throws an error. The technique can be further improved by changing the logic of the duplicated function so that the same fault could not be effectively used twice.

**Decryption at Place (DaP).** This technique is a variation of *FD* and specifically targets implementations of encryption algorithms. After encrypting a given plaintext a resulting cipher is sent to a decryption function and its output is compared with the original plaintext (see Listing 7.6). If the encryption (or decryption) function was corrupted the resulting comparison would fail. A determined attacker would then need to corrupt the decryption function in the same way or attack the part of the program responsible for result verification.

## 7.5 Chaos Duck

In this work, we considered fault injection on the binary level, as this reflects the realities of fault injection on an IoT device. Some faults require care in their injection, e.g. branch faults (*B* or *BC*), while others, e.g. bit flip faults (*FLP*), can be injected into any part of the binary.

To assist in finding and applying fault injection, we developed and here introduce Chaos Duck – an automatic fault-injection tool that is agnostic to the binary implementation. The only assumption Chaos Duck makes is that a binary was compiled for a specified processor architecture.

**Strategically injecting faults.** Chaos Duck disassembles a given binary and parses its assembly code collecting information about the instructions set (e.g., address space and size), branch instructions locations and their targets, initialization of static variables and their corresponding values. This information is later used to produce *faulted* binaries with injected faults.

For branch faults (*B* or *BC*), Chaos Duck modifies the target of a branch instruction to point to a different location. A new target address is picked sequentially within the program's *text* section including addresses in the middle of a valid instruction. For each branch instruction in the original binary multiple faulted binaries are produced with a different target address.

For *NOP* faults Chaos Duck checks all the possible outcomes by replacing each original instruction with a `nop` instruction sequentially. The same approach is used for *FLP* faults for each instruction bit.

Chaos Duck looks for variable declarations that have a numeric value of up to four bytes and 'zeros' them with *Z1B* or *Z1W* faults producing a new faulted binary. This fault type targets variables controlling the number of encryption rounds or loop counters.

**Evaluating outcomes.** Depending on the injected fault the resulting binary can behave differently during the execution. Some faulted binaries may produce a result that was not expected under normal circumstances, e.g., an invalid cipher that will be impossible to decrypt. Other faulted binaries may cause a plaintext or even an encryption key to appear in the output leading to sensitive data leaks. At the same time, faulted binaries may also terminate abruptly due to faults in their logic. Some may fail with a segmentation fault or crash with an error code, while the others might end up in an infinite loop.

Chaos Duck automatically executes all the generated faulted binaries and collects the results (stdout/stderr outputs, exit codes, and timeouts). It accepts an encryption key, plaintext and an expected cipher as input parameters. It then performs several checks on the binary output. First, it checks whether a plaintext or an encryption key appears in any of the faulted binaries' outputs. Second, it checks if the produced cipher (if any) is the expected one. In case of an invalid cipher, Chaos Duck records the fault's type and an error code.

## 7.6 Methodology

To evaluate the hardening effectiveness we compared six different implementations of PRESENT: the *baseline* canonical C implementation with no hardening, and five implementations hardened with techniques described in Section 7.4. All the implementations were compiled for the ARM architecture using the *arm-linux-gnueabi-gcc* compiler with no optimization (`-O0` flag). We specifically targeted the ARM architecture since most IoT devices are ARM-based. The resulting binaries accepted a 64 bits plaintext and a 80 bits key as inputs and output a 64 bits cipher.

With Chaos Duck we applied the faults described in Section 7.3 to the baseline and hardened binaries and generated faulted binaries, i.e. copies of the original binary with a single injected fault. For each fault model, every possible fault location was considered. We used a set of three encryption keys and three plaintexts resulting in nine executions per binary with a 3 second timeout for each. We then measured the total number of faulted binaries for baseline and hardened versions and collected statistics on fault types and their success rate measured as the number of binaries generating an invalid cipher or leaking sensitive data samples. The latter served as an indicator of hardening technique efficiency against fault attacks.

We also analyzed the ability of a non-hardened and hardened PRESENT implementations to withstand a key recovery attack when combined with a cryptanalytical attack (CA). During such an attack the last 31st round of encryption is skipped making it easier to extract the encryption key as part of a differential fault analysis (DFA) [105, 217]. We simulated this attack by manually setting the rounds number to 30 and recording the resulting ciphers for all *key,plaintext* pairs. We then checked whether any of these ciphers appeared consistently in the outputs of hardened and non-hardened faulted binaries.

Finally, we measured the average execution time for baseline and hardened binaries across 10000 executions with randomly generated *key,plaintext* pairs and with the first 200 execution results skipped to avoid caching concerns. Additionally, we recorded the size in bytes for all binaries.

## 7.7 Evaluation

We evaluate hardening techniques on three fronts. First, we analyze their overall effectiveness in preventing sensitive data leakage, e.g. a plaintext or an encryption key, in presence of faults. Second, we study their general fault tolerance against different types of faults, and analyze the impact of each fault type. Finally, we analyze the performance impact of each of the hardening techniques on program runtime as compared with the non-hardened version.

### 7.7.1 Sensitive data leakage

For each hardening technique we count the number of faulted binaries that leaked a plaintext or an encryption key in their output (stdout or stderr). Each leak type is presented in two categories: *normal* faulted binary execution and an *interrupted* faulted binary execution based on a 3 sec timeout (marked with 'timeout'). In case of the latter, the output of is a (potentially) non-terminating stream of bytes which may include a plaintext or encryption key.

Table 7.1 features the results of our analysis. We observe that with hardened and non-hardened faulted binaries it is only possible to leak the plaintext, and never the key, indicating that leaking the key is in general hard to achieve even for a non-hardened code. We can see that hardening techniques *CLH*, *SC*, *FD*, and *DaP* all reduce the number of leaks, while the *VD* technique is largely ineffective at protecting against leaks, and instead causes more leaks.

Two faulted binaries of a baseline PRESENT implementation were vulnerable to the DFA attack. Their 31st round of encryption was skipped consistently for any *key,plaintext* pair. A bit flip (FLP) fault was the cause in both cases: by

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Binaries** | **1314549** | **4818348** | **9267993** | **52341831** | **3580380** | **3902400** |
| Leaked key (normal/timeout) | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| Leaked plaintext (normal/timeout) | 1713 / 180 | 1449 / 0 | 3559 / 4 | 567 / 0 | 0 / 72 | 108 / 72 |
| DFA vulnerable | 2 / 0 | 9 / 0 | 2 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |

Table 7.1 – Sensitive data leakage across five hardening techniques.

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Binaries** | **1713 / 180** | **1449 / 0** | **3559 / 4** | **567 / 0** | **0 / 72** | **108 / 72** |
| FLP | 45 / 108 | 9 / 0 | 28 / 0 | 234 / 0 | 0 / 0 | 36 / 0 |
| Z1B/Z1W | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 | 0 / 0 |
| NOP | 0 / 0 | 0 / 0 | 0 / 0 | 9 / 0 | 0 / 0 | 0 / 0 |
| B | 852 / 0 | 396 / 0 | 341 / 1 | 324 / 0 | 0 / 0 | 0 / 0 |
| BC | 816 / 72 | 1044 / 0 | 3190 / 3 | 0 / 0 | 0 / 72 | 72 / 72 |

Table 7.2 – Fault types statistics for faulted binaries leaking sensitive data. (normally terminated / terminated by timeout).

flipping a single bit in a branch instruction regulating the encryption loop the type of branch was changed from `bls` (branch if less or equal to 30) to `blt` (branch if less than 30). Similarly, only two hardened versions of PRESENT were vulnerable, namely *CLH* and *VD* with 9 and 2 binaries respectively. The leaks were caused by the same FLP faults as in a baseline version.

Next, we analyzed the type of faults causing faulted binaries to leak sensitive data (plaintext) for a baseline and five hardened versions. The vast majority of leaks were caused by branch instructions faults (both conditional and unconditional) (see Table 7.2). This was expected since these faults aim to skip the execution of functions performing sensitive operations (e.g., encryption) or fault detection. Bit flip faults (FLP) were the second most common cause of

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Binaries** | **1314549** | **3836889** | **5639832** | **52341831** | **3580380** | **3902400** |
| Valid cipher | 16.7 % | 38.64 % | 34.07 % | 62.87 % | 16.99 % | 14.02 % |
| Invalid cipher | 17.3 % | 2 % | 1.64 % | 0.19 % | 0.57 % | 0.55 % |
| No output | 66 % | 59.36 % | 64.29 % | 36.94 % | 82.45 % | 85.43 % |

Table 7.3 – Overall execution results in presence of faults.

data leaks, followed by NOP faults with just a few binaries leaking sensitive data. Z1B/Z1W faults failed to cause data leaks in all cases.

Overall, hardening techniques *FD* and *DaP* proved to be the most effective in protecting against sensitive data leaks. The *FD* technique explores redundancy of sensitive computations while *DaP* performs an in-place results verification. In contrast to other hardening techniques, *FD* and *DaP* operate with the final encryption results instead of intermediate ones.

### 7.7.2 Fault tolerance

To analyze the general fault tolerance of the baseline and hardened versions we count the number of faulted binaries that were unaffected by any of the faults and produced a valid cipher, then those that produced an invalid cipher, and, finally, those that crashed and produced no output. Note, the binaries leaking sensitive data in their output were not considered in this experiment.

The results of the analysis are presented in Table 7.3. The percentage of faulted binaries producing a valid cipher is higher, sometimes significantly, than the baseline for most of the hardening techniques. The highest percentage was achieved with the *SC* technique (62.87%) which proved to be more resistant to faults as compared to other techniques. The hardening techniques exploring redundancy on a variable level (i.e. *CLH* and *VD*) were less efficient, while techniques exploring redundancy on a function level (i.e. *FD* and *DaP*) were the least efficient. At the same time, the vast majority of faulted binaries across all five hardening techniques crashed during the execution and provided no output. This is expected since in most of the cases the injected faults corrupt the program logic and raise exception errors. To have a better understanding of the true cause of these crashes we collect statistics on the error codes returned

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| **Crashed binaries** | **867852** | **1862129** | **4807851** | **19334285** | **3505885** | **3845470** |
| Seg. fault | 56.11 % | 35.21 % | 20.99 % | 17.36 % | 45.28 % | 60.81 % |
| Timed out | 30.65 % | 6.81 % | 43.46 % | 0.48 % | 15.33 % | 9.87 % |
| Illegal instr. | 8.83 % | 1.82 % | 0.85 % | 0.65 % | 1.83 % | 2.25 % |
| Aborted | 0.63 % | 1.7 % | 0.68 % | 0.11 % | 0.29 % | 0.29 % |
| Fault detected | n/a | 52.4 % | 33.2 % | 80.22 % | 35.3 % | 25.03 % |
| Other | 3.78 % | 2.06 % | 0.82 % | 1.18 % | 1.97 % | 1.75 % |

Table 7.4 – Statistics on failed executions.

by the crashed binaries. The results are presented in Table 7.4.

In case of a baseline non-hardened PRESENT version, the vast majority of faulted binaries crashed due to a segmentation fault, while the others were interrupted by timeout or crashed while trying to execute an illegal instruction. For the hardened binaries the situation was slightly different. While segmentation faults and timeout errors still constitute the major cause of failure, particularly for *DaP* and *VD*, a significant portion of faulted binaries detected a presence of faults in their execution logic and terminated by throwing a corresponding error. The fault detection rate varies across all five hardening techniques ranging from 25% to 80% for the *DaP* and *SC* techniques respectively.

### 7.7.3 Performance analysis

Table 7.5 features the execution times for all six binaries (baseline and five hardened versions) averaged across 10k runs. We also measure the binary size to see if hardening techniques have any significant impact on file size.

All five hardening techniques have little impact on the binary size adding on average 1 KByte to the original size, the only exception being the technique implementing statement counters (*SC*) that nearly doubles the size of the original binary. This is expected since this technique adds two additional lines of code for each line in the original non-hardened code. In terms of runtime performance, we see no significant difference in execution times.

| | **Baseline** | **CLH** | **VD** | **SC** | **FD** | **DaP** |
|---|---|---|---|---|---|---|
| File size (bytes) | 16448 | 17340 | 17652 | 24524 | 17764 | 17908 |
| Execution time (ms) | 40.8 | 39.77 | 42.16 | 42.41 | 42.36 | 42.09 |

Table 7.5 – Runtime performance and file size comparison

## 7.8 Discussion

Considering the leakage of sensitive information, we observe that none of the hardening techniques were able to prevent plaintext leakage. This is in line with the results of the previous studies on the impossibility of effective countermeasures to faults [114], but is also concerning since this kind of leakage is a dangerous vulnerability. We note, however, that the *FD* and *DaP* hardening techniques were the most effective at reducing sensitive data leaks. This is due to their checks of the final output data instead of the intermediate values, e.g. loop counters. Nevertheless, these techniques may still be vulnerable to multiple fault injections that other hardening techniques would detect locally.

The number of faulted binaries still producing a valid ciphertext is another important parameter for analysis. Our experiments showed that the majority of faulted binaries simply failed to execute correctly, either crashing or entering an infinite loop. However, the *SC* technique appeared to be the most effective for operating correctly when faulted.

When it comes to types of faults that contributed to sensitive data leaks there is an absolute leader – branch instruction faults. For this reason, the hardening techniques that add more branches to the original binary should be avoided since they create more locations that can be faulted to leak information. This was confirmed by the test results in which a *SC* technique showed the highest success rate for branch instruction faults. New branches introduced with the statement counters checks inadvertently increased the attack surface. On the contrary, alternative techniques, e.g. *FD* and *DaP*, proved to be less affected by this type of faults.

Some of the hardening techniques, namely *CLH*, *SC* and *VD* require specialized tools to annotate the source code automatically. As a result, the developers remain oblivious to the nature of the hardening modifications and their impact on program security. Other techniques, e.g., *FD* and *DaP*, are easier to implement and reason about.

In terms of performance, in all cases we see no significant impact on runtime performance, nor on a binary size. The *SC* technique was the only exception that had an impact on both the binary size and the runtime performance. However, considering its fault detection rate (the highest among all the techniques we have studied) this technique strikes a good balance between the security and performance. Overall, there is a great potential in hardening techniques exploring redundancy on a function level. This granularity is in a sweet spot between the required developer effort and a desired program safety when a single fault injection is considered.

Finally, we discuss Chaos Duck's performance and its ability to simulate various faults. Chaos Duck proved to be a useful tool in hands of a developer seeking to improve the safety properties of the software he/she develops. It allows to systematically perform a fault-tolerance analysis as part of a CI/CD cycle. We note, however, that the Chaos Duck prototype could be further optimized in order to reduce the time needed to explore all the potential fault space. This can be achieved, for instance, by exploring a low-level language for Chaos Duck implementation (e.g. C).

## 7.9 Related work

There is a large body of research on both fault injection and hardening techniques. Below we provide an overview of key related works.

**Fault Injection.** Various fault injection techniques have been described and studied previously demonstrating a wide variety of faults and ways to invoke a particular behavior of a given program [51, 165, 89, 152, 111, 150, 179, 183, 192]. These include fault injection at compile time using LLVM [152, 183] or on a binary level (like this work) [89, 112, 111]. Their approaches range from purely experimental [152] to formally verified [165] or even both [112].

**Software hardening.** Various hardware and software hardening techniques have been proposed in the past [45, 109]. The most well-known technique implements the N-version [37] approach where multiple implementations of the same algorithm are executed in parallel and their results are compared for consistency. The computation redundancy ensures the fault-tolerance, since a fault in one version will cause an inconsistency with other versions' results. Many hardening techniques implement the same approach but on the variable [71, 190], statement [43, 71], function [42], or even instruction level [46, 165].

Another classic countermeasure against software faults relies on using 'canary' words strategically placed in the program's memory stack by a compiler to prevent buffer overflow attacks [76]. Other techniques suggested encrypting the pointer addresses instead [75] or randomizing the address space [52, 100]. Alternative techniques propose countermeasures based on hardware and software checksums or randomization of execution order [42].

## 7.10 Summary

Faults can have a devastating effect on IoT software security, especially those that target components implementing encryption algorithms. In this chapter we described our evaluation of several common hardening techniques applied to an implementation of PRESENT cipher. We compared the impact of these techniques on security and performance, and analyzed their general resistance to different types of faults. We determined that techniques exploring redundancy on a function level strike a good balance between software security and performance properties. We also found out that some of the techniques made the software more vulnerable to faults resulting in a plaintext being leaked.

To evaluate the efficiency of hardening techniques we developed Chaos Duck – a framework that strategically injects faults into a given software and collects statistics on the impact of each fault type. Chaos Duck being completely automated can be a useful tool in hands of the developers seeking to improve the security properties of their software. We envision Chaos Duck to be used to systematically perform a fault-tolerance analysis as part of a continuous integration and development (CI/CD) cycle.

Chapter 8

# Analysis and limitations

In this chapter, we identify common properties across all proposed systems described in the previous chapters and point out their current limitations. To this end, we first present a unified trust model of private-by-design IoT systems. Using this model, we then discuss the limitations of these systems and potential ways to address those.

## 8.1 A trust model of private-by-design systems

Private-by-design IoT systems presented in the previous chapters while being used in different domains all share the same high-level design that is based on three key components stacked on top of each other forming a so-called *pyramid of trust* as shown at Figure 8.1. These components are *trusted hardware*, *trusted software*, and, finally, *trusted third party*. We will now describe each of these components.

**Trusted hardware**: The trusted hardware at the bottom of the pyramid of trust lays a solid foundation for all other components. It provides an essential secure environment and confidentiality and integrity protection for all computations on sensitive sensor data. In HomePad, we rely on a trusted hub that is fully controlled by the user and is typically deployed locally within the smart home to eliminate the risks of unauthorized access. In PatrIoT we rely on TEE provided by Intel SGX technology, which allows us to perform secure computations in the untrusted cloud environment. With SGX enclaves we can leverage the computational resources of a given cloud host without

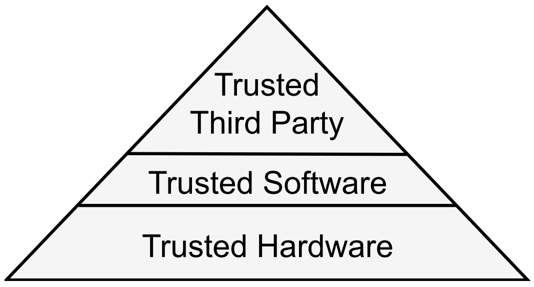


Figure 8.1 – Trust model of private-by-design IoT systems (pyramid of trust)

necessarily trusting its operating system, software stack, connected hardware, and even administrators with physical access to the host – so long as the SGX enclaves are implemented correctly. To verify that, we perform an attestation procedure which validates the integrity and confidentiality properties of the enclave each time PatrIoT starts. Finally, in Flowverine we relax the trusted hardware requirement slightly by relying on a general mobile hardware to preserve compatibility with the stock Android OS. As an alternative, we could use the TEE provided by the ARM TrustZone technology available on all modern smarthphones. With it we could provide similar confidentiality and integrity protection as the one offered by Intel SGX enclaves.

**Trusted software**: The trusted software represents the core component of private-by-design systems. It provides a software stack that ensures a secure and private sensitive data collection and processing by the untrusted third-party IoT apps. This trusted software implements the dataflow programming model, an interface for the users to securely connect their IoT devices, install apps, and specify their privacy policy rules, and, finally, a mechanism for app verification and policy enforcement. In HomePad, the trusted software consists of the hub controller and runtime code, as well as the code of device drivers and trusted elements' stubs. In PatrIoT, the trusted software consists of the TSAR service code which implements a runtime manager, a flowwall security monitor, an elements' API and device drivers, and a remote attestation procedure for SGX enclaves. In Flowverine, the trusted software comprises the app development toolchain with the code weaving service, the certification service which ensures app authenticity, a manager app to install Flowverine apps on user devices, and, finally, the Flowverine middleware which provides a runtime environment for these apps. Finally, in all of the mentioned systems we assume the IoT device software to be trusted and configured to interact exclusively with HomePad hub, PatrIoT cloud instance or a smartphone.

**Trusted third party**: The trusted third parties play an important role in the design and continuous operation of private-by-design IoT systems. The dataflow programming model relies on a set of trusted elements that are provided as part of an API for app developers to use. These elements are embedded in all of the proposed systems but are originally implemented and maintained by their respective third-party developers. The core software components of HomePad, PatrIoT and Flowverine are also envisioned to be developed and maintained by a single entity or a consortium of trusted third-party developers in an open-source and fully transparent fashion. Finally, in PatrIoT we rely on a trusted third-party represented by the SCONE's configuration and attestation service (CAS) to attest the secure state of the SGX enclaves. Similarly, in Flowverine the app certification service is maintained by a trusted third-party which can be a single individual or an organization.

## 8.2 Limitations of private-by-design IoT systems

In the previous chapters we described how secure and private-by-design IoT systems can be build. We use various security mechanisms to make sure that sensitive sensor data is processed in a way that is in line with user expectations and preferences. Any attempts to circumvent these security mechanisms must be blocked by the system. The latter is a fundamental requirement for the users when deciding whether to trust a given system with their data or not. Despite our best effort to fully comply with such a requirement, some of the technologies and trusted parties we rely on have their own flaws and limitations that may have a negative impact on overall system security. While some of these flaws are related to the assumptions we have made in the design of our systems, others are related to the current limitations of the technologies we use. Next, we discuss these flaws and limitations in detail.

### 8.2.1 Limitations related to trusted hardware

As stated previously, the trusted hardware plays a fundamental role in building private-by-design systems. If the integrity of the trusted hardware is violated the whole system will be compromised. It is thus important to recognize and take into account current limitations and potential flaws of the trusted hardware components we rely on.

In HomePad, we rely on a Linux-based local device represented by a personal computer or a dedicated smart home hub with a specialized hardware to connect various IoT devices. In both cases, one of the potential attack vectors is a network connection. A skillful attacker could try to exploit a vulnerability in the Linux OS or any of the software components installed at the hub to gain unauthorized access to the sensor data by sending a malicious HTTP request to the HomePad server. The risks and the consequences of such an attack could be substantially minimized if not eliminated completely by keeping the hub up-to-date with the latest security patches and software updates, and by using a network firewall (e.g. iptables, Fail2ban, or FirewallD) to limit external connections to trusted devices and/or users only. Another potential attack vector is through a physical access to the trusted hardware. An attacker could compromise the security by directly accessing sensitive data stored in the volatile or persistent memory of the device (e.g., via a memory dump). To prevent this type of attack, the hub should be placed outside the reach of any unauthorized party, e.g., at the locked closet.

In PatrIoT we rely on Intel SGX enclaves to provide integrity and confidentiality protection for sensitive data processing at the untrusted cloud environment. However, various side channel attacks have been demonstrated over the last few years allowing the attackers with physical access to the host machine to compromise the enclave security and eavesdrop on its content [26, 72, 141]. To address these attacks Intel has released patched versions of processor microcode and BIOS updates. While Intel actively works with academia and open source partners to help mitigate the threats, we cannot be certain about the protection level provided by the latest security patches and their ability to prevent new types of attacks. Nevertheless, PatrIoT can take advantage of various alternative mitigation techniques [176, 199, 200] that address specific SGX side channel threats.

In Flowverine, the trusted hardware is represented by the mobile device, i.e., smartphone. While we do not address the security concerns on the hardware level, we rely on built-in access control mechanisms available in the latest versions of Android, such as hardware-backed full phone encryption and biometric-based authentication, to prevent unauthorized access to Flowverine middleware and sensor data it manages. However, some attacks are still possible even with these security measures in place [216]. An attacker with physical access to a given smartphone could use a USB connection to interact with ADB (Android Developer Bridge) interface to add and execute malicious scripts and exploit vulnerabilities in the device. Attackers can also exploit Android's recovery mode to gain privileged access to user data. To mitigate these attacks, the latest Android security patches need to be installed and activated on the device. This, however, proved to be a weak point due to the infrequent patch release cycle and an abundance of outdated devices that receive no further updates from their respective manufacturers. Using the newest smartphone models with the latest security patches seems to be the only way to mitigate potential security compromises.

### 8.2.2 Limitations related to trusted software

The trusted software is an essential part of private-by-design IoT systems. It implements a middleware for sensitive data processing in compliance with user defined preferences. This software is expected to operate correctly and have mechanisms in place to prevent potential attacks. Next, we describe some of the limitations of the trusted software we use that could be potentially exploited by the attackers.

The core modules of HomePad, PatrIoT and Flowverine have direct access to sensitive sensor data, so the attackers can try to exploit vulnerabilities in those modules or in any of their dependencies. With respect to core modules, an attacker could try to add a malicious code snippet into one of the modules which when activated will forward the sensor data samples to the external host controlled by the attacker. For instance, a device driver or its element stub code could be compromised this way. Considering the open source nature of the proposed systems, one way to mitigate such attacks would be to utilize the power of community to perform extensive code review and detect potential deviations from the modules' desired and advertised specifications. Similar approach is used in development of Linux kernel and in other open source projects. Alternatively, static code analysis techniques could be applied whose goal is to search for particular attack patterns every time a new software update is submitted. Such an analysis may be performed as part of a continuous integration and delivery (CI/CD) cycle along with the unit testing procedures.

The same kind of attacks could be carried on any of the third party Java libraries or npm packages the core modules depend on. An attacker can compromise the security of the core modules by injecting a malicious code into one of their dependencies. Unfortunately, such attacks are quite common and have been reported previously [237]. The mitigation strategies include a vetting process that yields trusted maintainers, and a code review and vetting for new releases of certain sensitive packages. If a given dependency passes both vetting processes, we can substantially minimize the risks of it being malicious. The same vetting processes can be effectively used to mitigate attacks that exploit known and still not patched vulnerabilities in certain packages.

Both HomePad and PatrIoT, as well as Flowverine, all have a relatively large trusted computing base (TCB). This can be viewed as a limitation, since a large TCB exposes a greater attack surface which can be exploited by the attacker. By shrinking the TCB size we can reduce the exposure of potential vulnerabilities and improve overall software robustness. To some extent, we reduce the TCB size by stripping off non-essential modules and system features, leaving those available only to advanced users as part of the custom system package. We also limit the number and size of the libraries in the dependency list, and use slim Docker images where possible. While all of these measures are not sufficient on their own, they help to minimize the number and consequences of the potential attacks.

Next, we address the limitations of the operating systems all of our proposed systems rely on. In both HomePad and PatrIoT we rely on a Linux-based OS (Debian and Alpine Linux respectively), while Flowverine runs on top of Android OS. In all of these cases, the security level depends on the availability of the latest security updates, with the highest level when all of such updates are installed and activated. For Linux-based OSes the security patches release cycle is rather short resulting in all of the newly discovered vulnerabilities being patched within few days after reporting. This is, however, not always a case with Android OS which, as described previously in Section 8.2.1, suffers from infrequent patch release cycle or absence of patches for discontinued devices. Nevertheless, we assume that latest software updates and security patches are applied in a timely manner to ensure the maximum level of protection.

Finally, the existing IoT device software represents another limitation of our approach. In all of our proposed systems, we assume that IoT devices are configured to send sensor data exclusively to the HomePad hub, PatrioT cloud instance or a smartphone that are controlled by the user – the owner of these devices. This is however not the case for existing IoT devices that are currently incompatible with the private-by-design model. They are configured to stream sensor data to the cloud servers that are usually maintained and fully controlled by the device manufacturer. This model constitutes the original problem of sensor data privacy which motivated our research. We showed that our proposed private-by-design system model benefits both users and device manufacturers without sacrificing data privacy, however, it might still take time for this approach to get adopted by major manufacturers and service providers. Nevertheless, we see a constant shift towards privacy-friendly IoT systems design [64, 174, 156] and general interest of public in privacy-oriented technologies [127].

### 8.2.3 Limitations related to trusted third parties

There are several trusted third parties we rely on. All of those could potentially act maliciously and by doing so compromise the security of the whole system.

The developers of trusted API elements can introduce a malicious behavior into the element's code. This can be done either intentionally or unintentionally by adding a bug in otherwise legitimate procedure. The consequence of such a behavior can range from denial of service (DoS) attacks preventing normal element's operation, to sensitive data leaks. In Chapter 6 we studied the feasibility of N-version programming to bootstrap trust in these trusted elements and prevent potential malicious behavior. By running several element instances provided by different developers we can minimize the risks of security breaches, so long as these developers do not collude. Additionally, code review and a vetting process can be effectively used in this case. The same mitigation techniques can be used to prevent core module developers from acting maliciously by producing a malicious version of the module. In general, the open source nature of the project and transparency of the developer actions make it easy to track individual changes in the trusted software stack and prevent changes that scrutinize security.

We rely on Intel developers to correctly implement SGX hardware and software components. However, this trust is compulsory. Intel is a single entity that has exclusive rights and control over SGX technology and all of the supporting services, e.g., device registration, remote attestation or security updates. As a consequence, Intel constitutes a single point of failure: if, for instance, Intel's attestation service fails or, worse, gets compromised, the trusted system which relies on this service can become unavailable or insecure. We face similar challenges with the SCONE's CAS service which PatrIoT uses to attest the state of the SCONE container. The CAS service is maintained by a single entity and can become unavailable and thus prevent the deployment of new PatrIoT instances, or act maliciously by falsely attesting the authenticity of compromised PatrIoT Docker images. In both cases, the only way to mitigate such risks is to have a distributed chain of trust maintained by several independent entities, similar to certificate authorities (CAs) that verify the authenticity of issued TLS certificates on the web. Making software components of SGX technology and SCONE library OS open source besides allowing self-hosted attestation services, can also improve security, since a detailed and thorough security audit will then be possible.

Chapter 9

# Conclusions and future work

In this chapter, we summarize the main contributions of this thesis, and outline the directions for future work.

## 9.1 Conclusions

One of the biggest barriers to the widespread adoption of IoT devices involves concerns over privacy of the sensor data these devices collect. Numerous cases of sensitive data leaks and abuse have been reported, but no adequate measures have been taken so far to prevent such situations from happening in the future. In fact, existing IoT services and platforms continue to harvest and monetize sensitive sensor data from the connected devices without providing any control to the end users – the owners of those devices. To protect against such threats, in this thesis, we revisited the design of IoT services and platforms so as to provide the end-users with greater transparency and control over how their data is collected and used. To this end, we proposed a model for building private-by-design IoT systems that span across local (home), cloud and mobile domains. In all of these systems the end-users retain full control over their device data and can benefit from various third-party IoT services and applications without sacrificing their privacy.

Within the local domain, we introduced HomePad, a privacy-aware hub for smart homes. HomePad aims to empower users with the ability to determine how various IoT applications (apps) access and process sensitive data collected by smart devices (e.g., web cams) and to prevent these apps from executing unless they abide by the privacy restrictions specified by the users. To achieve this goal, HomePad implements a dataflow programming model in which apps are implemented as directed graphs of *elements*, and each element is represented by an instance of a function that processes data in isolation. By modeling the behavior of graph elements and their interactions with other elements using Prolog rules, HomePad allows for automatic verification of the app's data flows against user-defined privacy policies. We implemented a prototype of HomePad and performed a thorough performance and security analysis. Homepad incurs a negligible performance overhead, requires a modest programming effort, and provides a flexible policy support to address the privacy concerns most commonly expressed by potential smart home users.

For the cloud domain, we proposed PatrIoT, a private-by-design IoT platform for smart homes. PatrIoT revisits the typical architecture of existing cloud-based IoT platforms, and provides an alternative design which allows end-users to obtain fine-grained control of data flows generated by their IoT devices. It leverages Intel SGX to prevent unauthorized access to the data by untrusted IoT cloud providers, and offers users an intuitive security abstraction named *flowwall* to specify easy-to-use policies for controlling sensitive sensor data flows within their smart homes. We have built and evaluated a PatrIoT prototype on several fronts focusing primarily on performance, policy expressiveness, usability. Performance-wise we saw no significant differences when running IoT apps inside and outside PatrIoT. In terms of available throughput, despite a significant overhead introduced by the SGX technology, PatrIoT can sustain a typical smart home traffic load. Finally, most of the participants in a field study considered PatrIoT to be easy to use, and its supported policies to be highly expressive and flexible.

For the mobile domain, we introduced Flowverine, a system for building privacy-sensitive mobile apps for unmodified Android platforms. Flowverine exposes an API based on a dataflow programming model which allows for efficient taint tracking of sensitive data flows within each app. By checking such flows against a security policy, Flowverine can prevent potential privacy violations. We implemented a prototype of Flowverine and evaluated it on several fronts. Our evaluation showed that Flowverine can be used to implement mobile apps that handle security-sensitive information flows while preserving compatibility with Android OS and incurring small performance overheads.

Finally, we introduced additional techniques that aim to enhance the security and privacy properties of all three systems. First, we studied the feasibility of applying N-version programming (NVP) to bootstrap trust in software components provided by third party developers. Such components can be used as part of the trusted software stack within the dataflow programming model, hence, ensuring their correct implementation and behavior is essential for overall system security. Our results showed that NVP can be a viable option to securing these software components. We then study additional ways to ensure secure data handling on the device level, by comparing the impact of various hardening techniques on IoT device software security and performance. As a result, we offered a guideline for IoT developers seeking to make their software robust to fault-injection attacks, and a tool for automatic fault-tolerance analysis and evaluation.

## 9.2 Directions for future work

Certain aspects of the proposed systems could be further explored and/or improved. Below we highlight some of the potential directions for future research and exploration.

The dataflow programming model would benefit from a browser-based visual interface allowing app developers or even end-users to create app flow graphs by selecting necessary elements from the palette and connecting them together without necessarily writing any code. Such app graphs could then be automatically verified against the privacy policy rules and deployed to the app store or to the runtime system with a single-click. Node-RED programming tool for event-driven applications is a great example of such an interface [19]. Node-RED can complement the dataflow programming model and take advantage of the flows verification mechanisms this model provides.

Exploring the ways to apply our data flows verification mechanism in the context of cloud security is another interesting research direction. For instance, graphs can be used to visualize and reason about potential security problems within complex cloud services deployments. Verifying the graph structure and potential data flows between different service instances could help to expose otherwise hidden flows and dependency relationships between service's assets and validate assumptions about privacy and security risks. Existing tools that are widely used by cloud developers [157, 138, 169, 54, 149] can only analyze access rights and exposure of various cloud instances, but not sensitive data flows between those. We see a great potential in applying our data flows modeling and verification techniques in this context. As the IoT apps, service clusters could be modeled with Prolog rules and facts describing each service's generated data types and connections.

There are certain aspects related to dataflow programming model and its verification mechanism that could be further improved. For instance, the detection and prevention of implicit data flows generated by the app graphs. While our programming model makes sensitive data flows explicit and subject to verification, a determined app developer can still leak sensitive sensor data through implicit low-bandwidth covert channels, for instance, based on communication patterns to authorized network destinations. Devising methods for shaping traffic and reducing bandwidth of such channels is an interesting topic for further studies.

Reducing the size of the trusted computing base (TCB) of the proposed systems is another interesting research direction. This is especially important for PatrIoT which due to its dependency on third-party libraries (e.g., SCONE library) has a rather large TCB. While reducing the TCB size without sacrificing the performance and security is generally a challenging task, one of the potential directions could be exploring other available *library operating systems* that might have smaller TCB size, e.g., Graphene-SGX [214]. Furthermore, SCONE library proved to be not suitable for network-intensive applications, like PatrIoT's TSAR service. The latter had a significant performance loss when running inside SGX enclaves provided by SCONE library. Other library OSes might offer better performance in the same environment and under the same conditions.

Lately ARM processors have been making remarkable inroads into the cloud environment. Naturally, multiple researchers have explored the adoption of ARM TrustZone technology in order to provide an isolated environment for sensitive data processing securely in the cloud [60, 59]. Considering the encouraging results of the previous studies we argue that TrustZone technology could be a viable option for hosting PatrIoT software stack within the untrusted cloud environment. Studying its performance impact and security guarantees constitutes an exciting direction for future research.

# Bibliography

[1] A Creepy Website Is Streaming From 73,000 Private Security Cameras. https://gizmodo.com/a-creepy-website-is-streaming-from-73-000-private-secur-1655653510. Accessed August 2021.

[2] A Leak Suggests That Google Employees May Be Listening In On Your Conversations With Google Home. https://www.pastemagazine.com/articles/2019/07/a-leak-proves-google-is-listening-in-on-your-conve.html. Accessed August 2021.

[3] Advanced vm/sandbox for Node.js. https://github.com/patriksimek/vm2. Accessed August 2021.

[4] Alexa Skills Kit. https://developer.amazon.com/alexa-skills-kit. Accessed August 2021.

[5] Amazon Alexa. https://developer.amazon.com/en-US/alexa. Accessed August 2021.

[6] Amazon Echo. https://www.amazon.com/Amazon-Echo-Bluetooth-Speaker-with-WiFi-Alexa/dp/B00X4WHP5E. Accessed August 2021.

[7] Amazon employees listen in to your conversations with Alexa. https://www.zdnet.com/article/amazon-employees-are-listening-in-to-your-conversations-with-alexa/. Accessed August 2021.

[8] Amazon Sends 1,700 Alexa Voice Recordings to a Random Person. https://threatpost.com/amazon-1700-alexa-voice-recordings/140201/. Accessed August 2021.

[9] Apple HomeKit. https://developer.apple.com/homekit/. Accessed August 2021.

[10] EU General Data Protection Regulation (GDPR). https://www.eugdpr.org/. Accessed August 2021.

[11] German parents told to destroy Cayla dolls over hacking fears. http://www.bbc.com/news/world-europe-39002142. Accessed August 2021.

[12] Goodbye privacy, hello 'Alexa': Amazon Echo, the home robot who hears it all. https://www.theguardian.com/technology/2015/nov/21/amazon-echo-alexa-home-robot-privacy-cloud. Accessed August 2021.

[13] Hacks to turn your wireless IP surveillance cameras against you. http://www.networkworld.com/article/2224469/microsoft-subnet/hacks-to-turn-your-wireless-ip-surveillance-cameras-against-you.html. Accessed August 2021.

[14] How Alexa, Siri, and Google Assistant Will Make Money Off You. https://www.technologyreview.com/s/601583/how-alexa-siri-and-google-assistant-will-make-money-off-you/. Accessed August 2021.

[15] How Amazon Echo Users Can Control Privacy. https://www.forbes.com/sites/tonybradley/2017/01/05/alexa-is-listening-but-amazon-values-privacy-and-gives-you-control. Accessed August 2021.

[16] How private is Amazon Echo? https://www.slashgear.com/how-private-is-amazon-echo-07354486/. Accessed August 2021.

[17] IFTTT. https://ifttt.com. Accessed August 2021.

[18] Kaldi ASR. `http://www.kaldi-asr.org/`. Accessed August 2021.

[19] Node-RED: Low-code programming for event-driven applications. `https://nodered.org/`. Accessed August 2021.

[20] Privacy and Information Sharing. `http://www.pewinternet.org/2016/01/14/privacy-and-information-sharing/`. Accessed August 2021.

[21] Samsung SmartThings. `https://www.smartthings.com`. Accessed August 2021.

[22] Speech API - Speech Recognition. `https://cloud.google.com/speech/`. Accessed August 2021.

[23] The FBI Can Neither Confirm Nor Deny Wiretapping Your Amazon Echo. `http://paleofuture.gizmodo.com/the-fbi-can-neither-confirm-nor-deny-wiretapping-your-a-1776092971`. Accessed August 2021.

[24] Webcam Maker Takes FTC's Heat for Internet-of-Things Security Failure. `http://www.technewsworld.com/story/78891.html`. Accessed August 2021.

[25] What did she say?! Talking doll Cayla is hacked. `http://www.bbc.com/news/av/technology-31059893/what-did-she-say-talking-doll-cayla-is-hacked`. Accessed August 2021.

[26] Voltpillager: Hardware-based fault injection attacks against intel SGX enclaves using the SVID voltage scaling interface. In *Proc. of USENIX Security*, Vancouver, B.C., Aug. 2021. USENIX Association.

[27] Z. Alkhalifa, V. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, 1999.

[28] T. Allard, N. Anciaux, L. Bouganim, Y. Guo, L. Le Folgoc, B. Nguyen, P. Pucheral, I. Ray, I. Ray, and S. Yin. Secure Personal Data Servers: a Vision Paper. In *Proc. of VLDB*, 2010.

[29] B. Amos, B. Ludwiczuk, and M. Satyanarayanan. Openface: A general-purpose face recognition library with mobile applications. Technical report, CMU-CS-16-118, CMU School of Computer Science, 2016.

[30] B. Amos, B. Ludwiczuk, M. Satyanarayanan, et al. Openface: A general-purpose face recognition library with mobile applications. *CMU School of Computer Science*, 6(2), 2016.

[31] I. Analytics. State of the IoT 2018: Number of IoT devices now at 7B – Market accelerating. `https://iot-analytics.com/state-of-the-iot-update-q1-q2-2018-number-of-iot-devices-now-7b/`. Accessed August 2021.

[32] N. Anciaux, L. Bouganim, B. Nquyen, I. S. Popa, P. Pucheral, and P. Bonnet. Trusted cells: A sea change for personal data services. In *Proc. of CIDR*, 2013.

[33] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar. New Privacy Issues in Mobile Telephony: Fix and Verification. In *Proc. of CCS*, 2012.

[34] S. Arnautov, B. Trach, F. Gregor, et al. SCONE: Secure Linux Containers with Intel SGX. In *Proc. of OSDI*, 2016.

[35] C. D. Arvind and D. Culler. Dataflow architectures. *Annual review of computer science*, 1(1):225–253, 1986.

[36] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *Proc. of PLDI*, 2014.

[37] A. Avizienis. The n-version approach to fault-tolerant software. *IEEE Transactions on software engineering*, (12):1491–1501, 1985.

[38] G. S. Babil, O. Mehani, R. Boreli, and M. Kaafar. On the effectiveness of dynamic taint analysis for protecting against private information leaks on android-based devices. In *Proc. of SECRYPT*, 2013.

[39] L. Babun, A. K. Sikder, A. Acar, and A. S. Uluagac. IoTDots: A Digital Forensics Framework for Smart Environments. *arXiv preprint arXiv:1809.00745*, 2018.

[40] M. Backes, S. Bugiel, and E. Derr. Reliable third-party library detection in android and its security applications. In *Proc. of CCS*, 2016.

[41] M. Backes, S. Gerling, C. Hammer, M. Maffei, and P. von Styp-Rekowsky. Appguard: Enforcing user requirements on android apps. In *Proc. of TACAS*, 2013.

[42] H. Bar-El, H. Choukri, D. Naccache, M. Tunstall, and C. Whelan. The sorcerer's apprentice guide to fault attacks. *Proc. of the IEEE*, 94(2):370–382, 2006.

[43] G. Barbu, P. Andouard, and C. Giraud. Dynamic fault injection countermeasure. In *Proc. of CARDIS*, pages 16–30. Springer, 2012.

[44] A. Barenghi, L. Breveglieri, I. Koren, and D. Naccache. Fault injection attacks on cryptographic devices: Theory, practice, and countermeasures. *Proc. of the IEEE*, 100(11):3056–3076, 2012.

[45] A. Barenghi, L. Breveglieri, I. Koren, G. Pelosi, and F. Regazzoni. Countermeasures against fault attacks on software implemented aes: effectiveness and cost. In *Proc. of WESS*, pages 1–10, 2010.

[46] T. Barry, D. Couroussé, and B. Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proc. of CS2*, pages 1–6, 2016.

[47] A. Baumann, M. Peinado, and G. Hunt. Shielding Applications from an Untrusted Cloud with Haven. *ACM Trans. Comput. Syst.*, 33(3):8:1–8:26, Aug. 2015.

[48] J. Bell and G. Kaiser. Phosphor: Illuminating Dynamic Data Flow in Commodity JVMs. In *Proc. of OOPSLA*, 2014.

[49] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. Mockdroid: Trading privacy for application functionality on smartphones. In *Proc. of MCSA*, 2011.

[50] E. D. Berger and B. G. Zorn. Diehard: probabilistic memory safety for unsafe languages. In *Proc. of PLDI*, 2006.

[51] M. Berthier, J. Bringer, H. Chabanne, T.-H. Le, L. Rivière, and V. Servant. Idea: embedded fault injection simulator on smartcard. In *Proc. of ESSOS*, pages 222–229. Springer, 2014.

[52] S. Bhatkar, D. C. DuVarney, and R. Sekar. Address obfuscation: An efficient approach to combat a broad range of memory error exploits. In *Proc. of USENIX Security*, volume 12, pages 291–301, 2003.

[53] C. Blondeau and K. Nyberg. Links between truncated differential and multidimensional linear properties of block ciphers and underlying attack complexities. In *Proc. of EUROCRYPT*, pages 165–182. Springer, 2014.

[54] BloodHoundAD. Six Degrees of Domain Admin. `https://github.com/BloodHoundAD/BloodHound`. Accessed August 2021.

[55] A. Bogdanov, L. R. Knudsen, G. Leander, C. Paar, A. Poschmann, M. J. Robshaw, Y. Seurin, and C. Vikkelsoe. Present: An ultra-lightweight block cipher. In *Proc. of CHES*, pages 450–466. Springer, 2007.

[56] B. Bonné, S. T. Peddinti, I. Bilogrevic, and N. Taft. Exploring decision making with android's runtime permission dialogs using in-context surveys. In *Proc. of SOUPS*, 2017.

[57] G. Bouffard, B. N. Thampi, and J.-L. Lanet. Detecting laser fault injection for smart cards using security automata. In *Proc. of SSCC*, pages 18–29. Springer, 2013.

[58] G. Bradski et al. The opencv library. *Doctor Dobbs Journal*, 25(11):120–126, 2000.

[59] S. Brenner, C. Wulf, and R. Kapitza. Running zookeeper coordination services in untrusted clouds. In *Proc. of HotDep*, 2014.

[60] T. Brito, N. O. Duarte, and N. Santos. Arm trustzone for secure image processing on the cloud. In *Proc. of SRDSW*, pages 37–42. IEEE, 2016.

[61] M. Brusó, K. Chatzikokolakis, and J. Den Hartog. Formal Verification of Privacy for RFID Systems. In *Proc. of CSF*, 2010.

[62] C. Busold, S. Heuser, J. Rios, A.-R. Sadeghi, and N. Asokan. Smart and Secure Cross-Device Apps for the Internet of Advanced Things. In *Proc. of FC*, 2015.

[63] C. Cadar and P. Hosek. Multi-version software updates. In *Proc. of ICSE*, 2012.

[64] G. Campagna, R. Ramesh, S. Xu, M. Fischer, and M. S. Lam. Almond: The architecture of an open, crowdsourced, privacy-preserving, programmable virtual assistant. In *Proc. of WWW*, pages 341–350, 2017.

[65] R. Capizzi, A. Longo, V. Venkatakrishnan, and A. P. Sistla. Preventing information leaks through shadow executions. In *Proc. of ACSAC*, 2008.

[66] Z. B. Celik, P. McDaniel, and G. Tan. SOTERIA: Automated IoT safety and security analysis. In *Proc. of USENIX ATC*, 2018.

[67] Z. B. Celik, G. Tan, and P. McDaniel. IoTGuard: Dynamic enforcement of security and safety policy in commodity IoT. In *Proc. of NDSS*, 2019.

[68] S. Chakraborty, C. Shen, K. R. Raghavan, Y. Shoukry, M. Millar, and M. Srivastava. ipshield: A framework for enforcing context-aware privacy. In *Proc. of NSDI*, 2014.

[69] A. Chaudhry, J. Crowcroft, H. Howard, A. Madhavapeddy, R. Mortier, H. Haddadi, and D. McAuley. Personal data: thinking inside the box. In *Proc. of Aarhus*, 2015.

[70] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. of FTCS-8*, 1978.

[71] P. Cheynet, B. Nicolescu, R. Velazco, M. Rebaudengo, M. S. Reorda, and M. Violante. Experimentally evaluating an automatic approach for generating safety-critical software with respect to transient errors. *IEEE Transactions on Nuclear Science*, 47(6):2231–2236, 2000.

[72] T. Cloosters, M. Rodler, and L. Davi. Teerex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *Proc. of USENIX Security*, pages 841–858. USENIX Association, Aug. 2020.

[73] CNET. FTC and TrendNet settle claim over hacked security cameras, 2019. `https://www.cnet.com/news/ftc-and-trendnet-settle-claim-over-hacked-security-cameras/`. Accessed August 2021.

[74] M. . Company. Growing opportunities in the Internet of Things, 2019. `https://www.mckinsey.com/industries/private-equity-and-principal-investors/our-insights/growing-opportunities-in-the-internet-of-things`. Accessed August 2021.

[75] C. Cowan, S. Beattie, J. Johansen, and P. Wagle. Pointguard tm: protecting pointers from buffer overflow vulnerabilities. In *Proc. of USENIX Security*, pages 7–7. USENIX Association, 2003.

[76] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. Stackguard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proc. of USENIX Security*, volume 98, pages 63–78. San Antonio, TX, 1998.

[77] B. Cox, D. Evans, A. Filipi, J. Rowanhill, W. Hu, J. Davidson, J. Knight, A. Nguyen-Tuong, and J. Hiser. N-variant systems: A secretless framework for security through diversity. In *Proc. of Usenix Security*, 2006.

[78] N. Davies, N. Taft, M. Satyanarayanan, S. Clinch, and B. Amos. Privacy Mediators: Helping IoT Cross the Chasm. In *Proc. of HotMobile*, 2016.

[79] A. L. Davis and R. M. Keller. Data flow program graphs. 1982.

[80] R. De Keulenaer, J. Maebe, K. De Bosschere, and B. De Sutter. Link-time smart card code hardening. *International Journal of Information Security*, 15(2):111–130, 2016.

[81] J. B. Dennis. First version of a data flow procedure language. In *Programming Symposium*, pages 362–376. Springer, 1974.

[82] J. B. Dennis and D. P. Misunas. A preliminary architecture for a basic data-flow processor. In *Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, 1974.

[83] L. Deshotels, R. Deaconescu, M. Chiroiu, L. Davi, W. Enck, and A.-R. Sadeghi. Sandscout: Automatic detection of flaws in ios sandbox profiles. In *Proc. of ACM CCS*, 2016.

[84] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Proc. of S&P*, 2010.

[85] W. Ding and H. Hu. On the Safety of IoT Device Physical Interaction Control. In *Proc. of CCS*, 2018.

[86] C. Dixon, R. Mahajan, S. Agarwal, A. Brush, B. Lee, S. Saroiu, and P. Bahl. An Operating System for the Home. In *Proc. of NSDI*, 2012.

[87] V. D'silva, D. Kroening, and G. Weissenbacher. A survey of automated techniques for formal software verification. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[88] L. Dureuil, G. Petiot, M.-L. Potet, T.-H. Le, A. Crohen, and P. de Choudens. Fissc: A fault injection and simulation secure collection. In *Proc. of SAFECOMP*, pages 3–11. Springer, 2016.

[89] L. Dureuil, M.-L. Potet, P. de Choudens, C. Dumas, and J. Clédière. From code review to fault injection attacks: Filling the gap using fault model inference. In *Proc. of CARDIS*, pages 107–124. Springer, 2015.

[90] P. Emami-Naeini, H. Dixon, Y. Agarwal, et al. Exploring how privacy and security factor into iot device purchase behavior. In *Proc. of CHI*, 2019.

[91] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for real-time privacy monitoring on smartphones. In *Proc. of OSDI*, 2010.

[92] A. C. Estes. Yes, Your Amazon Echo Is an Ad Machine. `https://gizmodo.com/yes-your-amazon-echo-is-an-ad-machine-1821712916`. Accessed August 2021.

[93] L. Fair. What Vizio was doing behind the TV screen. `https://www.ftc.gov/news-events/blogs/business-blog/2017/02/what-vizio-was-doing-behind-tv-screen`. Accessed August 2021.

[94] A. P. Felt, K. Greenwood, and D. Wagner. The effectiveness of application permissions. In *Proceedings of the 2nd USENIX conference on Web application development*, pages 7–7. USENIX Association, 2011.

[95] A. P. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pages 1–14, 2012.

[96] E. Fernandes, J. Jung, and A. Prakash. Security analysis of emerging smart home applications. In *Proc. of IEEE S&P*, 2016.

[97] E. Fernandes, J. Paupore, et al. Flowfence: Practical data protection for emerging iot application frameworks. In *Proc. of USENIX Security*, 2016.

[98] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proc. of NDSS*, 2018.

[99] Forbes. When 'Smart Homes' Get Hacked: I Haunted A Complete Stranger's House Via The Internet. `http://www.forbes.com/sites/kashmirhill/2013/07/26/smart-homes-hack`. Accessed August 2021.

[100] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Proc. of HotOS)*, pages 67–72. IEEE, 1997.

[101] J. Fruhlinger. Consumers are losing interest in smart home tech according to sales-rank data. `https://media.thinknum.com/articles/consumers-are-losing-interest-in-smart-home-tech-according-to-sales-rank-data/`. Accessed August 2021.

[102] G. J. Gaeth and J. Shanteau. Reducing the influence of irrelevant information on experienced decision makers. *Organizational Behavior and Human Performance*, 33(2):263–282, 1984.

[103] N. Garun. Almost half a million pacemakers need a firmware update to avoid getting hacked. `https://www.theverge.com/2017/8/30/16230048/fda-abbott-pacemakers-firmware-update-cybersecurity-hack`. Accessed August 2021.

[104] H.-T. Geek. The Smarthome Industry Has Reached a Plateau. Here's What's Holding It Back. `https://www.howtogeek.com/359365/3-things-holding-back-the-smarthome-industry/`. Accessed August 2021.

[105] N. F. Ghalaty, B. Yuce, and P. Schaumont. Differential fault intensity analysis on present and led block ciphers. In *Proc. of COSADE*, pages 174–188. Springer, 2015.

[106] S. Gibbs. Hackers can hijack Wi-Fi Hello Barbie to spy on your children. `https://www.theguardian.com/technology/2015/nov/26/hackers-can-hijack-wi-fi-hello-barbie-to-spy-on-your-children`. Accessed August 2021.

[107] B. Gierlichs, J.-M. Schmidt, and M. Tunstall. Infective computation and dummy rounds: Fault protection for block ciphers without check-before-output. In *Proc. of LATINCRYPT*, pages 305–321. Springer, 2012.

[108] K. Gilles. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[109] G. C. Gilley, L. Bearnson, C. Carroll, W. Bouricius, E. Hsieh, G. Putzolu, J. Roth, P. Schneider, C. Tan, M. Hsiao, et al. International symposium on fault-tolerant computing, digest of papers. Pasadena, California, March 1-3 1971.

[110] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum. Back to the future: Fault-tolerant live update with time-traveling state transfer. In *Proc. of LISA*, 2013.

[111] T. Given-Wilson, A. Heuser, N. Jafri, and A. Legay. An automated and scalable formal process for detecting fault injection vulnerabilities in binaries. *Concurr. Comput. Pract. Exp.*, 31(23), 2019.

[112] T. Given-Wilson, N. Jafri, and A. Legay. The state of fault injection vulnerability detection. In M. F. Atig, S. Bensalem, S. Bliudze, and B. Monsuez, editors, *Proc. of VECoS*, volume 11181 of *Lecture Notes in Computer Science*, pages 3–21. Springer, 2018.

[113] T. Given-Wilson, N. Jafri, and A. Legay. Combined software and hardware fault injection vulnerability detection. *Innovations in Systems and Software Engineering*, 2020.

[114] T. Given-Wilson and A. Legay. Formalising fault injection and countermeasures. In *Proc. of ARES*, 2020.

[115] I. Goirizelaia, T. Selker, M. Huarte, and J. Unzilla. An Optical Scan E-Voting System Based on N-Version Programming. *Proc. of IEEE S&P*, 6(3):47–53, 2008.

[116] E. Gomes. Dataflow-based Framework for Privacy-aware Android Apps. `https://github.com/GreenStage/homepad_android`. Accessed August 2021.

[117] E. Gomes, I. Zavalyshyn, N. Santos, J. Silva, and A. Legay. Flowverine: Leveraging dataflow programming for building privacy-sensitive android applications. In *Proc. of TrustCom*, 2020.

[118] S. Govindavajhala and A. W. Appel. Using memory errors to attack a virtual machine. In *Proc. of S&P*, pages 154–165. IEEE, 2003.

[119] C. S. Group. The CMU Audio Databases, 2017. `http://www.speech.cs.cmu.edu/databases/an4/`. Accessed August 2021.

[120] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[121] S. Hao, D. Li, W. G. Halfond, and R. Govindan. SIF: a Selective Instrumentation Framework for Mobile Applications. In *Proc. of MobiSys*, 2013.

[122] C. Hauser. Police Use Fitbit Data to Charge 90-Year-Old Man in Stepdaughter's Killing. `https://www.nytimes.com/2018/10/03/us/fitbit-murder-arrest.html`. Accessed August 2021.

[123] Y.-i. Hayashi, N. Homma, T. Sugawara, T. Mizuki, T. Aoki, and H. Sone. Non-invasive emi-based fault injection attack against cryptographic modules. In *Proc. of EMC EUROPE*, pages 763–767. IEEE, 2011.

[124] R. Herbster, S. DellaTorre, P. Druschel, and B. Bhattacharjee. Privacy capsules: Preventing information leaks by mobile apps. In *Proc. of MobiSys*, 2016.

[125] A. Hern. Fitness tracking app Strava gives away location of secret US army bases . `https://www.theguardian.com/world/2018/jan/28/fitness-tracking-app-gives-away-location-of-secret-us-army-bases`. Accessed August 2021.

[126] A. Hern. Hacking risk leads to recall of 500,000 pacemakers due to patient death fears. `https://www.theguardian.com/technology/2017/aug/31/hacking-risk-recall-pacemakers-patient-death-fears-fda-firmware-update`. Accessed August 2021.

[127] R. Hindi. Why you should bet big on privacy. `https://techcrunch.com/2016/05/17/why-you-should-bet-big-on-privacy/`. Accessed August 2021.

[128] G. J. Holzmann. The model checker spin. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[129] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These aren't the droids you're looking for: Retrofitting android to protect data from imperious applications. In *Proc. of CCS*, 2011.

[130] T. Hunt, Z. Zhu, Y. Xu, et al. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. of OSDI*, 2016.

[131] K. Imamura, R. B. Heckendorn, T. Soule, and J. A. Foster. N-Version Genetic Programming via Fault Masking. In *Proc. of EUROGP*, 2002.

[132] B. Insider. Wisconsin couple describe the chilling moment that a hacker cranked up their heat and started talking to them through a Google Nest camera in their kitchen, 2019. `https://www.businessinsider.com/hacker-breaks-into-smart-home-google-nest-devices-terrorizes-couple-2019-9`. Accessed August 2021.

[133] Y. Jaradin et al. Scoll–a language for safe capability based collaboration. 2005.

[134] E. Jenn, J. Arlat, M. Rimen, J. Ohlsson, and J. Karlsson. Fault injection into vhdl models: the mefisto tool. *Predictably Dependable Computing Systems*, pages 329–346, 1995.

[135] 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 *Proc. of NDSS*, 2017.

[136] 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 *Proc. of NDSS*, 2017.

[137] C. Johnston. What Google can really do with Nest, or really, Nest's data. `https://arstechnica.com/information-technology/2014/01/what-google-can-really-do-with-nest-or-really-nests-data/`. Accessed August 2021.

[138] D. Jones. Dow Jones Hammer: Protect the cloud with the power of the cloud(AWS). `https://github.com/dowjones/hammer`. Accessed August 2021.

[139] K. Kafle, K. Moran, S. Manandhar, A. Nadkarni, and D. Poshyvanyk. A study of data store-based home automation. In *Proc. of CODASPY*, 2019.

[140] J. Karlsson, P. Folkesson, J. Arlat, Y. Crouzet, G. Leber, and J. Reisinger. Application of three physical fault injection techniques to the experimental assessment of the mars architecture. *Dependable Computing and Fault Tolerant Systems*, 10:267–288, 1998.

[141] Z. Kenjar, T. Frassetto, D. Gens, M. Franz, and A.-R. Sadeghi. V0ltpwn: Attacking x86 processor integrity from software. In *Proc. of USENIX Security*, 2020.

[142] Y. Kim, R. Daly, J. Kim, C. Fallin, J. H. Lee, D. Lee, C. Wilkerson, K. Lai, and O. Mutlu. Flipping bits in memory without accessing them:

An experimental study of dram disturbance errors. *ACM SIGARCH Computer Architecture News*, 42(3):361–372, 2014.

[143] J. C. Klontz, B. F. Klare, S. Klum, A. K. Jain, and M. J. Burge. Open source biometric recognition. In *Proc. of BTAS*, 2013.

[144] J. C. Knight and N. G. Leveson. An Experimental Evaluation of the Assumption of Independence in Multiversion Programming. *IEEE Transactions on Software Engineering*, pages 96–109, 1986.

[145] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The click modular router. *Proc. of TOCS*, 2000.

[146] P. R. Kosinski. A data flow language for operating systems programming. In *Proceeding of ACM SIGPLAN-SIGOPS interface meeting on Programming languages-operating systems*, pages 89–94, 1973.

[147] M. Kost, J.-C. Freytag, F. Kargl, and A. Kung. Privacy Verification using Ontologies. In *Proc. of ARES*, 2011.

[148] D. F. Kune, J. Backes, S. S. Clark, D. Kramer, M. Reynolds, K. Fu, Y. Kim, and W. Xu. Ghost talk: Mitigating emi signal injection attacks against analog sensors. In *Proc. of S&P*, pages 145–159. IEEE, 2013.

[149] D. Labs. CloudMapper helps you analyze your Amazon Web Services (AWS) environments. `https://github.com/duo-labs/cloudmapper`. Accessed August 2021.

[150] J.-F. Lalande, K. Heydemann, and P. Berthomé. Software countermeasures for control flow integrity of smart card c codes. In *Proc. of ESORICS*, pages 200–218. Springer, 2014.

[151] P. Lamere, P. Kwok, E. Gouvea, B. Raj, R. Singh, W. Walker, M. Warmuth, and P. Wolf. The cmu sphinx-4 speech recognition system. In *Proc. of ICASSP*, 2003.

[152] H. M. Le, V. Herdt, D. Große, and R. Drechsler. Resilience evaluation via symbolic fault injection on intermediate code. In *Proc. of DATE*, pages 845–850. IEEE, 2018.

[153] A. Lee, T. Kawahara, and K. Shikano. Julius—an open source real-time large vocabulary recognition engine. 2001.

[154] L. Lenc and P. Král. Unconstrained Facial Images: Database for face recognition under real-world conditions. In *Proc. of MICAI*, October 2015.

[155] L. Li, A. Bartel, T. F. Bissyandé, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel. Iccta: Detecting inter-component privacy leaks in android apps. In *Proc. of ICSE*, 2015.

[156] N. Lomas. The FTC Warns Internet Of Things Businesses To Bake In Privacy And Security. `https://techcrunch.com/2015/01/08/ftc-iot-privacy-warning/`. Accessed August 2021.

[157] Lyft. Cartography. `https://github.com/lyft/cartography`. Accessed August 2021.

[158] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller. Detecting information flow by mutating input data. In *Proc. of ACE*, 2017.

[159] C. Matyszczyk. Samsung's warning: Our Smart TVs record your living room chatter. `https://www.cnet.com/news/samsungs-warning-our-smart-tvs-record-your-living-room-chatter/`. Accessed August 2021.

[160] M. J. May, C. A. Gunter, and I. Lee. Privacy APIs: Access Control Techniques to Analyze and Verify Legal Privacy Policies. In *Proc. of CSFW*, 2006.

[161] D. McAuley, R. Mortier, and J. Goulding. The dataware manifesto. In *Proc. of COMSNETS*, 2011.

[162] J. McGregor. Here's How Amazon's Ring Doorbell Police Partnership Affects You. `https://www.forbes.com/sites/jaymcgregor/2019/08/06/heres-how-amazons-ring-doorbell-police-partnership-affects-you`. Accessed August 2021.

[163] Z. Meng, Y. Xiong, W. Huang, L. Qin, X. Jin, and H. Yan. Appscalpel: Combining static analysis and outlier detection to identify and prune

undesirable usage of sensitive data in android applications. *Neurocomputing*, 341, 2019.

[164] N. Moro, A. Dehbaoui, K. Heydemann, B. Robisson, and E. Encrenaz. Electromagnetic fault injection: towards a fault model on a 32-bit microcontroller. In *Proc. of FDTC*, pages 77–88. IEEE, 2013.

[165] N. Moro, K. Heydemann, E. Encrenaz, and B. Robisson. Formal verification of a software countermeasure against instruction skip attacks. *Journal of Cryptographic Engineering*, 4(3):145–156, 2014.

[166] S. S. Mukherjee, C. Weaver, J. Emer, S. K. Reinhardt, and T. Austin. A systematic methodology to compute the architectural vulnerability factors for a high-performance microprocessor. In *Proc. of MICRO*, pages 29–40. IEEE, 2003.

[167] A. C. Myers. JFlow: Practical Mostly-Static Information Flow Control. In *Proc. of POPL*, 1999.

[168] G. Nest. Nest Energy Partners. `https://nest.com/energy-partners/`. Accessed August 2021.

[169] Netflix. Security Monkey. `https://github.com/Netflix/security_monkey`. Accessed August 2021.

[170] T. D. News. Smart devices hacked in digital home invasions, 2019. `https://eu.detroitnews.com/story/business/2019/02/12/smart-home-devices-like-nest-thermostat-hacked/39049903/`. Accessed August 2021.

[171] D. T. Nguyen, C. Song, Z. Qian, S. V. Krishnamurthy, E. J. Colbert, and P. McDaniel. IotSan: fortifying the safety of IoT systems. In *Proc. of CoNEXT*, 2018.

[172] J. Oberheide, E. Cooke, and F. Jahanian. CloudAV: N-Version Antivirus in the Network Cloud. In *Proc. of USENIX Security*, 2008.

[173] 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 *Proc. of WiSec*, 2019.

[174] O. of the privacy commissioner of Canada. Privacy guidance for manufacturers of Internet of Things devices. `https://www.priv.gc.ca/en/privacy-topics/technology/gd_iot_man/`. Accessed August 2021.

[175] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE transactions on Reliability*, 51(1):111–122, 2002.

[176] O. Oleksenko, B. Trach, R. Krahn, M. Silberstein, and C. Fetzer. Varys: Protecting sgx enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*, 2018.

[177] P. Olson. Nest Gives Google Its Next Big Data Play: Energy. `https://www.forbes.com/sites/parmyolson/2014/01/13/nest-gives-google-its-next-big-data-play-energy/`. Accessed August 2021.

[178] T. Pasquier, J. Singh, D. Eyers, and J. Bacon. CamFlow: Managed Data-Sharing for Cloud Services. In *Proc. of IEEE TCC*, 2015.

[179] K. Pattabiraman, N. Nakka, Z. Kalbarczyk, and R. Iyer. SymPLFIED: Symbolic program-level fault injection and error detection framework. In *Proc. of DSN*, pages 472–481. IEEE, 2008.

[180] Philips. Hue Developer Program. `https://developers.meethue.com`. Accessed August 2021.

[181] R. Poddar, C. Lan, R. A. Popa, and S. Ratnasamy. Safebricks: Shielding network functions in the cloud. In *Proc. of NSDI*, 2018.

[182] A. Poschmann, A. Moradi, K. Khoo, C.-W. Lim, H. Wang, and S. Ling. Side-channel resistant crypto for less than 2,300 ge. *Journal of Cryptology*, 24(2):322–345, 2011.

[183] M.-L. Potet, L. Mounier, M. Puys, and L. Dureuil. Lazart: A symbolic approach for evaluation the robustness of secured codes against control flow injections. In *Proc. of ICST*, pages 213–222. IEEE, 2014.

[184] D. Povey, A. Ghoshal, G. Boulianne, L. Burget, O. Glembek, N. Goel, M. Hannemann, P. Motlicek, Y. Qian, P. Schwarz, et al. The kaldi speech recognition toolkit. In *Proc. of ASRU*, 2011.

[185] C. Priebe, K. Vaswani, and M. Costa. Enclavedb: A secure database using sgx. In *Proc. of IEEE SP*, 2018.

[186] J. Proy, K. Heydemann, A. Berzati, and A. Cohen. Compiler-assisted loop hardening against fault attacks. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(4):1–25, 2017.

[187] L. Qiu, Y. Wang, and J. Rubin. Analyzing the analyzers: Flowdroid/ic-cta, amandroid, and droidsafe. In *Proc. of ISSTA*, 2018.

[188] J.-P. Queille and J. Sifakis. Specification and verification of concurrent systems in cesar. In *Proc. of ISP*, pages 337–351. Springer, 1982.

[189] K. Razavi, B. Gras, E. Bosman, B. Preneel, C. Giuffrida, and H. Bos. Flip feng shui: Hammering a needle in the software stack. In *Proc. of USENIX Security*, pages 1–18, 2016.

[190] M. Rebaudengo, M. S. Reorda, and M. Violante. A new software-based technique for low-cost fault-tolerant application. In *Proc. of RAMS*, pages 25–28. IEEE, 2003.

[191] Y. Resten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In *Proc. of I-CAV*, pages 424–435. Springer, 1997.

[192] L. Rivière, M.-L. Potet, T.-H. Le, J. Bringer, H. Chabanne, and M. Puys. Combining high-level and low-level approaches to evaluate software implementations robustness against multiple fault injection attacks. In *Proc. of FPS*, pages 92–111. Springer, 2014.

[193] O. Ruwase, P. B. Gibbons, T. C. Mowry, V. Ramachandran, S. Chen, M. Kozuch, and M. Ryan. Parallelizing dynamic information flow tracking. In *Proc. of SPAA*, 2008.

[194] B. Salamat, T. Jackson, A. Gal, and M. Franz. Orchestra: intrusion detection using parallel execution and monitoring of program variants in user-space. In *Proc. of EuroSys*, 2009.

[195] N. Santos, R. Rodrigues, K. P. Gummadi, and S. Saroiu. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proc. of USENIX Security*, 2012.

[196] H. Schirmeier, C. Borchert, and O. Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proc. of DNS*, pages 319–330. IEEE, 2015.

[197] T. Schneider, A. Moradi, and T. Güneysu. Parti–towards combined hardware countermeasures against side-channel and fault-injection attacks. In *Proc. of CRYPTO*, pages 302–332. Springer, 2016.

[198] D. Schreckling, J. Posegga, J. Köstler, and M. Schaff. Kynoid: Real-time enforcement of fine-grained, user-defined, and data-centric security policies for android. In *Proc. of ISTP*, 2012.

[199] M.-W. Shih, S. Lee, T. Kim, and M. Peinado. T-sgx: Eradicating controlled-channel attacks against enclave programs. In *Proc. of NDSS*, 2017.

[200] S. Shinde, Z. L. Chua, V. Narayanan, and P. Saxena. Preventing page faults from telling your secrets. In *Proc. of ASIACCS*, pages 317–328, 2016.

[201] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu. Quantitative analysis of control flow checking mechanisms for soft errors. In *Proc. of DAC*, pages 1–6, 2014.

[202] J. Singh, T. Pasquier, J. Bacon, J. Powles, R. Diaconu, and D. Eyers. Big Ideas Paper: Policy-driven Middleware for a Legally-compliant Internet of Things. In *Proc. of Middleware*, 2016.

[203] F. Spiessens, Y. Jaradin, and P. Van Roy. Scoll and scollar safe collaboration based on partial trust. 2005.

[204] F. Spiessens, Y. Jaradin, and P. Van Roy. Using constraints to analyze and generate safe capability patterns. *Applications of Constraint Satisfaction and Programming to Computer Security Problems*, page 61, 2005.

[205] Statista. Number of internet of things (IoT) connected devices worldwide in 2018, 2025 and 2030, 2020. `https://www.statista.com/statistics/802690/worldwide-connected-devices-by-access-technology/`. Accessed August 2021.

[206] M. Sun, T. Wei, and J. C. Lui. Taintart: A practical multi-level information-flow tracking system for android runtime. In *Proc. of CCS*, 2016.

[207] B. Sunar, G. Gaubatz, and E. Savas. Sequential circuit design for embedded cryptographic applications resilient to adversarial faults. *IEEE Transactions on Computers*, 57(1):126–138, 2007.

[208] N. Theißing, D. Merli, M. Smola, F. Stumpf, and G. Sigl. Comprehensive analysis of software countermeasures against fault attacks. In *Proc. of DATE*, pages 404–409. IEEE, 2013.

[209] 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 *Proc. of USENIX Security*, 2017.

[210] Y. Tian, N. Zhang, Y.-H. Lin, X. Wang, B. Ur, X. Z. Guo, and P. Tague. Smartauth: User-centered authorization for the internet of things. In *Proc. of USENIX Security*, pages 361–378, 2017.

[211] N. Y. Times. Somebody's Watching: Hackers Breach Ring Home Security Cameras, 2019. `https://www.nytimes.com/2019/12/15/us/Hacked-ring-home-security-cameras.html`. Accessed August 2021.

[212] K. Tiri and I. Verbauwhede. A logic level design methodology for a secure dpa resistant asic or fpga implementation. In *Proc. of DATE*, volume 1, pages 246–251. IEEE, 2004.

[213] TRUSTe. US IoT Privacy Infographics. `https://www.truste.com/resources/privacy-research/us-internet-of-things-index-2015/`. Accessed August 2021.

[214] C.-C. Tsai, D. E. Porter, and M. Vij. Graphene-sgx: A practical library OS for unmodified applications on SGX. In *Proc. of USENIX ATC*, pages 645–658, 2017.

[215] K. Veeraraghavan, P. M. Chen, J. Flinn, and S. Narayanasamy. Detecting and surviving data races using complementary schedules. In *Proc. of SOSP*, 2011.

[216] T. Vidas, D. Votipka, and N. Christin. All your droid are belong to us: A survey of current android attacks. In *Proc. of Woot*, pages 81–90, 2011.

[217] G. Wang and S. Wang. Differential fault analysis on present key schedule. In *Proc. of CIS*, pages 362–366. IEEE, 2010.

[218] Q. Wang, W. U. Hassan, A. Bates, and C. Gunter. Fear and logging in the internet of things. In *Proc. of NDSS*, 2018.

[219] F. Wei, S. Roy, X. Ou, et al. Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps. In *Proc. of CCS*, 2014.

[220] D. Wu and S. Bratus. A context-aware kernel ipc firewall for android. In *Proc. of ShmooCon*, 2017.

[221] R. Xu, H. Saidi, and R. Anderson. Aurasium: Practical policy enforcement for android applications. In *Proc. of USENIX Security*, 2012.

[222] Y. Xu and E. Witchel. Maxoid: Transparently Confining Mobile Applications With Custom Views of State. In *Proc. of EuroSys*, 2015.

[223] Z. Xu and S. Zhu. Semadroid: A privacy-aware sensor management framework for smartphones. In *Proc. of DASP*, 2015.

[224] M. Yahyazadeh, P. Podder, E. Hoque, and O. Chowdhury. Expat: Expectation-based policy analysis and enforcement for appified smart-home platforms. In *Proc. of SACMAT*, 2019.

[225] 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 *Proc. of HotNets*, 2015.

[226] A. R. Yumerefendi, B. Mickle, and L. P. Cox. Tightlip: Keeping applications from spilling the beans. In *Proc. of NSDI*, 2007.

[227] I. Zavalyshyn. An automatic fault-injection tool used to evaluate software security and fault-tolerance. `https://github.com/zavalyshyn/chaosduck`. Accessed August 2021.

[228] I. Zavalyshyn. Private-by-design Smart Home Platform. `https://github.com/zavalyshyn/patriot`. Accessed August 2021.

[229] I. Zavalyshyn. Private IoT Smart Home Hub. `https://github.com/zavalyshyn/homepad`. Accessed August 2021.

[230] I. Zavalyshyn, N. O. Duarte, and N. Santos. An extended case study about securing smart home hubs through n-version programming. In *Proc. of ICETE*, pages 289–300, 2018.

[231] I. Zavalyshyn, N. O. Duarte, and N. Santos. Homepad: A privacy-aware smart hub for home environments. In *Proc. of SEC*, pages 58–73. IEEE, 2018.

[232] I. Zavalyshyn, T. Given-Wilson, A. Legay, and R. Sadre. Brief announcement: Effectiveness of code hardening for fault-tolerant iot software. In *Proc. of SSS*, 2020.

[233] I. Zavalyshyn, N. Santos, R. Sadre, and A. Legay. My house, my rules: A private-by-design smart home platform. In *Proc. of Mobiquitous*, 2020.

[234] F. Zhang, J. Chen, H. Chen, and B. Zang. Cloudvisor: Retrofitting protection of virtual machines in multi-tenant cloud with nested virtualization. In *Proc. of SOSP*, 2011.

[235] Y. Zhang, J. Dai, X. Zhang, S. Huang, Z. Yang, M. Yang, and H. Chen. Detecting third-party libraries in android applications with high precision and recall. In *Proc. of SANER*, 2018.

[236] W. Zhou, Y. Jia, Y. Yao, et al. Discovering and understanding the security hazards in the interactions between iot devices, mobile apps, and clouds on smart home platforms. In *Proc. of USENIX Security*, 2019.

[237] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel. Small world with high risks: A study of security threats in the npm ecosystem. In *Proc. of USENIX Security*, pages 995–1010, 2019.