PLOX: A Secure Serverless Framework for the Smart Home

by

Micheal Friesen

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2021

© Micheal Friesen 2021

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This thesis is based upon three different conference submissions to NSDI 2019, OSDI 2021 and NDSS 2022. The attached submission is therefore in part co-authored by myself, Ryan Hancock, Ali Mashtizadeh, Omid Abari, and Yousra Aafer, all authors of the submissions sent to these conferences.

The PLOX framework was designed together by myself and Ryan Hancock, under the supervision of both Ali Mashtizadeh and Omid Abari. PLOX was developed by both Ryan and I, with the source code and commit history available on the Reliable Computer Systems instance of Phabricator.

My development contributions to PLOX were focused on the manifest system, design and development of the protocol used between devices, development of the converted smart applications used in the evaluation and implementation/testing of the taint-based IFC system. I was also responsible for the implementations of Amazon IoT Greengrass, Azure IoT Edge and Home Assistant used to compare PLOX against other systems in the evaluation. The background research used to create the comparisons between the available smart hub frameworks and currently known vulnerabilities was also written and researched by myself. My development efforts were in partnership with Ryan Hancock, who led the development of the other core components of PLOX and offered valuable advice and guidance throughout.

Abstract

Smart hubs play a key role in the modern smart home in executing code on behalf of devices locally or on the cloud. Unfortunately, smart hubs are prone to security problems due to misconfigurations, device over permissioning and network mismanagement. In this work, I show the major vulnerabilities and attacks currently targeting smart hubs, and provide a brief overview of the literature that addresses these issues. After discussing the limitations found in the literature as well as the available off the shelf smart hubs, I provide an overview of PLOX, an end-to-end approach designed to combat a large number of the common vulnerabilities and security/privacy risks that impact smart hubs, while maintaining a moderate overhead.

PLOX is designed to sandbox applications on the home WiFi router. This allows for increased network controls, as well as lower latency in direct communication with devices. PLOX provides a new hybrid security model that combines a mandatory access control (MAC) system with information flow control (IFC), providing developer familiarity while addressing the overtainting issue found within taint based IFC systems through a serverless execution pattern. In our evaluations, PLOX outperforms Amazon Lambda by 500% and an open source smart hub solution, Home Assistant, by 13%, all while providing finer grained security policies and improved security guarantees. This is due to PLOX's locality and its light weight nature.

This work demonstrates that PLOX, an open source end-to-end solution for the smart home is well suited to address a large number of the security and privacy problems that the smart home suffers from. This work also highlights a number of novel approaches to smart hub designs, including the use of the home router to maintain device isolation, and combination of manifest and IFC based permission systems.

Acknowledgements

I would like to thank Ryan Hancock for his role as a mentor and friend throughout this project.

I would also like to thank Omid Abari and Ali Mashtizadeh for their guidance and mentorship as my co-supervisors.

I would also like to thank the friends and family that helped me focus and relax through the unprecedented reality of this past season.

Dedication

This work is dedicated to Rachelle Berg for her support, patience and love. Let it be known that planning a wedding while also completing a thesis is not a remarkably wise combination of activities.

Table of Contents

List of Figures ix

List of Tables x

1 Introduction 1

1.1 Smart Home threats 4

1.2 PLOX 6

2 Background 8

2.1 Security Issues 8

2.2 Security Models 11

3 Design 13

3.1 Threat Model 13

3.2 Design Overview 14

3.3 Network Isolation 17

3.4 PLOX Client 18
3.5 PLOX Service 18
3.6 PLOX Runtime 20
3.7 PLOX Shim Layer 21
3.8 PLOX Developer API 22
3.9 Scheduling 24
3.10 PLOX Security Policy 25
3.10.1 PLOX Manifests 25
3.10.2 PLOX's Sensitivity Label 27
3.10.3 Dynamic Capability Model and the Serverless Model 29

4 Evaluation **32**
4.1 Effectiveness 33
4.2 Performance 35

5 Future Work **42**
5.1 Limitations 44

6 Conclusion **46**

References **47**

List of Figures

3.1 The basic architecture of PLOX. 15

3.2 Example communication flows in a camera application, with associated manifest and code to dictate communication shown in Listing 3.2 and 3.1. Red arrows show external communication. 24

4.1 This figure shows overall overhead from running a sample benchmark within each of the three frameworks. Home Assistant was excluded from the C++ evaluation due to the need to run the C++ code through its python interface. 37

4.2 This figure shows a micro benchmark ran with and without (labeled Base) the PLOX framework. We ran networking calls that talked to three separate servers, a write server, a read server, and an echo server. We then measured time taken to do each of these calls, with the echo server representing a realistic view of the overhead of a standard API request that would be done by an IoT application. 38

List of Tables

2.1 Comparison of solutions designed to address the security and privacy concerns of the smart home. 9

4.1 A list of attacks found through the IoT ecosystem, how they were exploited and what part of PLOX protects against this attack. 36

4.2 Total round trip time of the C++ benchmark for PLOX and Amazon Lambda. Cold start up is defined after PLOX has retrieved both the manifest and code from the device. These artifacts are retrieved during the initial handshake with a device so should always be on disk. The cost of cold start up is initial setup of the C++ runtime and the forking of the lambda process. 39

Chapter 1

Introduction

Internet of Things (IoT) devices have continued to offer an increasing set of utilities and features for the modern smart home. From security cameras to door locks, smart home users have placed an increased amount of trust in the reliability and security of in home smart devices. Competing against this trust are the ever-growing challenges that arise from the variety of devices and device manufacturers. Each manufacturer has a different set security standards and practices, leaving a growing number of consumers vulnerable to data breaches and attacks. Numerous surveys have shown these risks along with privacy concerns deter users from bringing smart devices into their home [49, 51].

Consider a simple security system that includes a camera with a video feed used for face detection, a motion sensor to detect intruders, and a phone notification system. Each of these devices have different degrees of data privacy requirements and data sensitivity. Some tasks may require the use of cloud resources, while others are able to execute locally. A security camera recording the garden may have substantially different privacy concerns than a camera placed inside the home. Individual user permissions are varied between

household residents and guests. Even in this simple example, there are numerous possible devices and communication paths in which user privacy and security could become compromised. When expanded to include real-time operations such as emergency response systems, or devices with tangible security implications in failure cases such as a smart lock, the importance of addressing the litany of problems that can arise within this domain is significant.

IoT devices are diverse but typically have limited on-board resources, requiring processing to take place elsewhere. Cloud computing has been used extensively across the smart home. Smart devices leverage a number of developer tools the cloud provides such as auto scaling, serverless frameworks and remote firmware management to assist in smart home deployments. Users are left with a plug and play experience that only requires devices to be connected to the WiFi router. Despite these advantages, the cloud computing pattern has led to security and privacy issues. First, device data leaves the home, allowing cloud providers to store and process generated data, sacrificing privacy. These privacy issues are especially concerning, given the sensitive data captured by devices such as security systems and smart baby monitors.

Second, the lack of transparency in how devices are connected to each other can lead to security problems. When each device uses a separate set of cloud services, security mechanisms and privacy guarantees, attackers can take advantage of the least secured device in the home to compromise the entire system [36, 66]. Smart home devices serve a wide variety of responsibilities, each requiring a different degree of trust. Without a control mechanism in place to prevent an over permissioned device from accessing a home security system, users are left unable to proactively prevent misuse of their devices and data.

Furthermore, the reliance on cloud computation has led to total device unavailability when network connection is lost. With the ever increasing reliance on connected devices,

losing functionality of smart thermostats or smart locks can cause significant disruption to the home. One such example caused users to be unable to use their vacuum or unlock their doors due to an Amazon server failure [52].

Rather than use the cloud, many devices offload data processing to *smart hubs*, which are local devices that perform computation. Smart hubs have desirable reliability [52] and privacy properties due to their locality and continue to function in the case of an internet outage.

Samsung SmartThings [60], Apple Homekit [1] and Home Assistant [2] are a small subset of the growing number of solutions used that follow this model. A central point of computation provides a range of tools that orchestrate the connection and communication of data among smart devices, while managing device state and controlling device interactions. Smart hubs often mix the use of cloud resources and edge computing to minimize device functionality in network failure and run larger computational tasks on the cloud to minimize strain on the rest of the devices. A growing number of platforms that follow this model also allow third-party "smart applications", software that can connect and control devices within the home that can be downloaded by home users through a marketplace. These third party applications help provide extensibility for developers to use and control the smart device data and sensors in novel ways. Unfortunately, these third party smart apps provide further possibilities for poorly written or malicious applications to compromise the security and privacy of the smart home.

A number of open source solutions, including Home Assistant [2] and OpenHAB [55], have been developed to increase reliability, user privacy and user control over the smart home. Both offload to a local compute node for most tasks, only using cloud when necessary. Each have a focus on increasing the privacy of user data through ratings of smart applications and extensions to the home deployments, highlighting how properly config-

ured the smart applications are. Unfortunately, both OpenHAB and Home Assistant are developer-centric frameworks that are challenging for home users to properly configure. For example, to properly isolate devices, Home Assistant requires the configuration of local subnets within the network, a task that non-technical users are unlikely to perform. Also, to enable encryption for device communication, an extension must be enabled and configured, forcing users to provide certificates and execute numerous shell commands, rather than being enabled by default in platforms like Apple Homekit.

1.1 Smart Home threats

Malicious attacks against IoT devices are becoming more prevalent and sophisticated. 32% of all observed attacks in 2020 were against IoT devices, a 13% increase from 2019 [54]. Cybercriminals are intensifying their efforts by exploiting the weak security practices in the IoT ecosystem. Manufacturers often default leave factory known passwords on devices [7, 16], and use unencrypted communications to share sensitive user credentials between devices [42]. Out of date firmware leaves devices prone to exploitation by botnets [11, 63, 5] and open communication protocols can be exploited leading to the execution of malicious code [14, 53]. Cloud providers have also mistakenly exposed sensitive user data through public endpoints, leading to data privacy concerns. [8, 9, 30, 31, 64, 26].

Addressing these security issues is difficult as the IoT ecosystem is complex. First, IoT devices are heterogeneous ranging from low powered sensors to sophisticated smart hubs.

Second, the lack of transparency into inter-device communication has previously forced users to rely on manufacturers and developers to configure and secure their devices. Unfortunately, manufacturers and developers do not posses the required context to understand how users are using each device and what data is considered sensitive. This lack of context

has been shown to leave devices over-permissioned [44, 65, 57], leading to data leaks and privacy violations. This lack of ownership of user data is listed as one of the top issues by The Open Web Application Security Project. (OWASP) [15].

Smart hubs run third party applications in the home which necessitates the use of sandboxing to provide application isolation. However, popular sandboxing tools [50, 18, 3] are complex with difficult to configure security models, which are incomprehensible to users and difficult for developers [37]. Sandbox tools also lack context on the sensitivity of user data and are unable to the prevent the leaking of this data to external sources. Further, these sandboxing tools cannot address communications occurring outside of the sandbox environment as they lack pertaining context on how devices are communicating. For example, in the IoT ecosystem, devices communication rules may be dynamic but sandbox permissions and the objects they are tied to are static on creation.

The research community has proposed various solutions to address IoT security issues. Unfortunately, current solutions fall short because of the heterogeneity of IoT devices, and the opaque and complex inter-device communication model. Particularly, monitoring solutions rely on communication and external resource monitoring [38, 22, 62] like application code updates but require knowledge about IoT protocols to correctly recognize anomalous behavior. In practice, reverse engineering closed communications protocols is challenging and brittle in the face of future updates.

Other solutions improve user comprehension by better informing the user on what resources each device needs access to [44, 65, 46]. This requires developers to modify code to include code annotations to generate better prompts to the user. While this process can help reduce over-permission problems, they do not prevent devices from being attacked in the smart home. Additionally, if these user comprehension strategies influence the design of the permission system, the permissions can become coarse grained in an attempt to be

understandable to users, leading to further over-permissioning problems [44, 37].

1.2 PLOX

We present PLOX, a novel smart hub IoT framework that exposes a fine grained permission API to developers with comprehensible privacy controls for users. These permissions are enforced across IoT devices and applications running on PLOX. PLOX runs on the home router and isolates each IoT device while tracking sensitive data as it moves throughout the home to ensure it is never leaked to an untrusted external entity. The router is already a trusted device that arbitrates all communication between IoT devices, making it a natural place for operating on sensitive data.

PLOX introduces a novel hybrid security model that combines capability-based sandboxing (based on Capsicum [67]) with information flow control (IFC) [33, 59, 68, 28, 34]. Capabilities provide the fine-grained security policies for developers to limit and secure their application on the smart hub, while IFC is used to define user policy and dynamically remove permissions as an application accesses sensitive data.

A key insight is our observation that IFC and capability systems share much in common, and IFC systems only require the ability to revoke capabilities. Both models require similar interposition into system calls and control over file descriptors and other OS resources. We can apply both capability and IFC policies to our sandbox framework with a small modification to our system. IFC adds a modest overhead to only the resources protected by IFC.

PLOX developers use a serverless microservices architecture to achieve privilege separation [47, 56]. Applications are constructed as a combination of short stateless functions

that that can invoke one another. PLOX avoids overtainting problems common to IFC systems by restarting each stateless function after each invocation, allowing the runtime to reset all IFC labels and clear all process data.

This thesis makes the following contributions:

- A novel hybrid IFC/capability security model, called the Dynamic Capability model, that allows fine grained control over applications and user data. By using a serverless architecture for applications, we avoid the overtainting of processes due to IFC.
- PLOX's use of the router as a smart hub device, allowing for the proper restriction of device to device communication and isolation of devices.
- We evaluate the PLOX IoT framework and give a detailed outline of how PLOX defends against attack in the IoT ecosystem and show the low overhead of the hybrid security model on resource constrained devices.

We implemented a prototype of PLOX on FreeBSD 12.1, and evaluated it from several metrics. First, we simulated popular attacks in the IoT ecosystem (i.e., remote botnet and sensitive data leaks). Our experiments show that PLOX is effective in thwarting them in practice. Second, we evaluated PLOX's performance against Amazon Lambda and Home Assistant, a popular open source smart-hub framework. Our results show that PLOX outperforms both frameworks in round-trip time, demonstrating the feasibility of deploying PLOX on the home router.

Chapter 2

Background

In this chapter, we examine the causes behind the security and privacy vulnerabilities within the IoT ecosystem which motivates PLOX's design and give an overview of capability systems and IFC systems.

2.1 Security Issues

Table 2.1 shows a comparison of current state-of-the-art research solutions and the security vulnerabilities they protect against. We classify the problems broadly under three categories: configuration, open communication, and untrusted third-party application code for smart hubs.

Configuration Configuration vulnerabilities in IoT devices occur when manufacturers adopt insecure *default configurations* (e.g., static factory device credentials). Permission-based systems on IoT devices are also prone to misconfigurations when developers cannot

Security Concern	SmartThings	Home Assistant	HomeKit	Flowfence	ContexIot	HAWatcher	PLOX
Configuration							
Weak Authentications			✓				✓
Default Configurations			✓				✓
Open Communications							
Encrypted Communication	✓	✓	✓	✓			✓
Event Eavesdropping			✓	✓		✓	✓
Event Spoofing			✓	✓		✓	✓
Third Party Applications							
Coarse Grained Permissions		✓		✓	✓		✓
Data Locality		✓		✓			✓
Data Ownership		✓	✓				✓

Table 2.1: Comparison of solutions designed to address the security and privacy concerns of the smart home.

determine the minimal set of capabilities/permissions required to run correctly, often leading to overpermissioning.

Fundamentally, overpermissioning is caused by inherent limitations in current permission/capability systems; they require specific knowledge around systems calls and network interfaces [50, 3, 18], and often lack the granularity required for a correct execution of code (e.g., AppArmor [50] lacks fine-grained network controls). These systems also lack context around user policy and are far too complex for an average (non-technical) user to configure.

Open Communication Open protocols like MQTT [20] allow an adversary to interact with all devices (using *event eavesdropping/spoofing*) within the home, as many of these event systems are completely open to all connected devices. This enables adversaries to use a weakly secured device (using the configuration issues outlined above) to attack the entire system. For example, attackers were able to use a fish tank monitor to gain access

to a casino's database [13] and steal sensitive data through the fish monitor itself.

Other attacks like botnets [11] focus on exploiting the configuration issues outlined above to compromise any of the IoT devices that are externally reachable. Using weak authentication as a result from default configurations, attackers turn victim devices into remotely controlled bots with the purpose of later using them to perform various attacks, including distributed denial of service (DDoS) attacks, email spamming [53], and power grid attacks [63]. All these attacks exploit a lack of *monitored communication* between devices within the home. Other attacks such as those done on smart locks [42] exploits *unencrypted communications* to steal user credentials.

Third-Party Application Code The diversity of IoT devices on the market has lead to a variety of applications being run on smart hub, as well as a range of firmware versions on each device. Proper sandboxing of third-party applications is difficult due to the need to enforce policy on sensitive data (*data ownership*) while also restricting the functionality in the sandbox itself. Sandboxing techniques that have been used [4, 18, 3, 50] are too *coarse-grained*, thus often lead to overpermissioning.

Legislation has been proposed or passed in the UK [32] and California [21] to regulate default configurations and unpatched firmware issues, requiring manufactures to abide by a specified level of security standards. However, there are still major issues in properly executing third-party applications while upholding user data policies. Capture [22] is a novel solution used to handle the deployment and management of device firmware and third-party libraries. However, it only focuses on this issue and as such does not defend against poorly configured devices or overpermission issues.

2.2 Security Models

Capability/Permission Systems Capability systems are rule-based security systems that outline what a program is allowed to do. Often these systems require the use of manifests to outline these permissions [40]. Singularity [43] is an example system that uses manifest based programs to enforce the rule set created by the developer. As such these manifest-based systems are often used to create sandboxes to isolate each program [4, 50, 18, 3, 45]. Current sandbox systems suffer from coarse granularity of these permissions. Even the fine-grained permission systems are prone to misconfiguration which can lead to applications being exploited [41].

FreeBSD's Capsicum [67] is a capability sandbox that ties capabilities to file descriptors. Once a process enters capsicum mode it may only use the permissions given on the current file descriptors open. Processes are limited in the ways they can create new file descriptors. For example to open a file it requires the use of the `openat` system call with a relative parent file descriptor given the `CAP_LOOKUP` capability. The opened file descriptor can only have a subset of the rights currently given to the parent used to create it.

System calls that don't require a relative file descriptor are prohibited such as `open` or `getcwd`. This makes creating sandboxes for third party applications difficult due to the need to support these system calls in safe ways. To do this, the need for a system call interposition layer is required to translate system calls to a secure version. When additional file descriptors are required, a secure daemon process is needed to pass new file descriptors to this process. This sandboxing technique allows us to avoid many of the security flaws that can occur with system call interposition. However, once these file descriptors are given its difficult to revoke these file descriptors or permissions on them.

Information Flow Control IFC systems are a restrictive based security model which tracks the communication between processes, "tainting" them based off of how they read or write as this occurs. Tainting objects is the process of labeling an object with a unique identifier and category of taint. Operations that cause data to flow between two tainted objects causes the processes reading the data to gain the label of the writer.

This taint is accumulated and spread to other processes as it runs, restricting functionality as it moves through the system. Systems that use an information flow control model [34, 68, 28, 35, 37, 69, 48] often suffer from "overtainting" in which devices or applications become so tainted that they can no longer communicate with other entities on the network, effectively halting communication.

Chapter 3

Design

3.1 Threat Model

We assume that an attacker may use any external means to gain complete control of an IoT device. This can be through adversarial code deployment, using poorly configured security permissions, or attempting to communicate through weakly secured devices. This device cannot be the router itself as we assume this is a fully trusted device that has been properly configured. Once a device on the home network has become compromised, the attacker should be unable to:

1. Communicate with any device or service within the home that it is not specifically allowed in the manifest. This communication cannot be sent through the provided PLOX event system or directly through the network.
2. Send sensitive data outside the IoT network, even if the network endpoints themselves are allowed. If an endpoint is given trust to handle a device's sensitive data by an

administrator, only that device's sensitive data can be leaked. Further, an attack should be unable to pipe sensitive data to another device to be sent to an external source.

Out of scope of our threat model is the leakage of sensitive data or usage through external endpoints already deemed trustful by the user, or sensitive data located outside the network. Also attacks that involve overpermissioning by the developer to DDoS devices within the home using accepted events (this can be mitigated however). Side channel attacks that focus on timings throughout the PLOX system or attacks requiring physical acquisition or close proximity to the device [58] are also out of scope.

3.2 Design Overview

PLOX is a secure smart hub framework that enforces user policy around sensitive data through its use of the serverless paradigm to support privilege separation of IoT applications. PLOX uses its full view of the IoT ecosystem (due to its placement on the router) to mitigate or completely stop various attacks in it. This full view also allows PLOX to have context of data normally not seen by other solutions, allowing to display risk style questions [57, 44] to its users to configure devices which has been shown to be more effective for non-technical users.

Processes within the restricted *capability mode* of Capsicum are restricted from doing most system calls, and can only make calls using the file descriptors they have open. Capsicum can be challenging to use for 3rd party, off the shelf software, as such software assumes access to standard system calls. For example, the Python interpreter heavily relies on `getcwd` and `dlopen` when importing new modules.

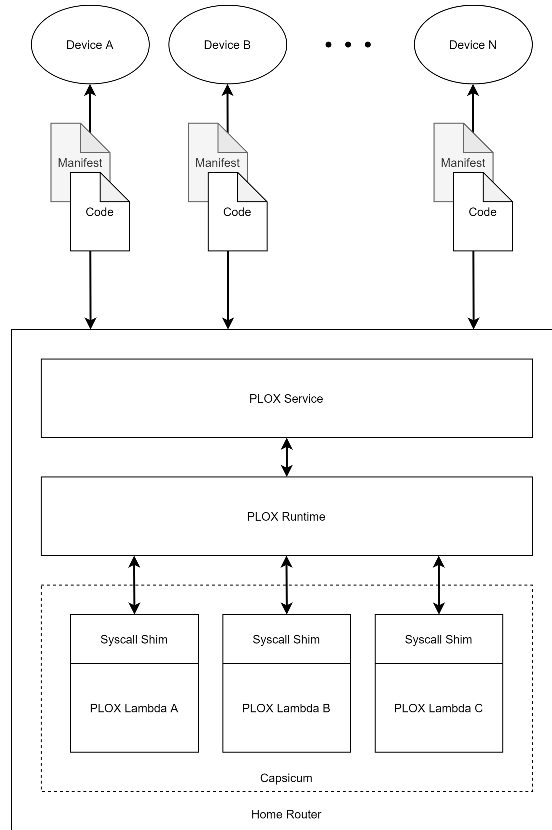


Figure 3.1: The basic architecture of PLOX.

PLOX overcomes this using its system call interposition layer, which also acts as the main abstraction to enforce IFC. This abstraction allows PLOX to run nearly unmodified third party C/C++ or Python code. We define the functions provided and their execution runtimes using a commonly used term – "lambdas".

PLOX uses a manifest [40, 43] to allow developers to describe the resources needed by the application. The manifest outlines the functionality of each lambda through the files, network endpoints and general system call usage needed during the execution of the lambda(§ 3.10.1). PLOX provides a "yell" mode for developers that identifies all

permissions required when testing the application and generates a manifest.

As shown in Figure 3.1 PLOX consists of four main components: the PLOX service, PLOX runtime, PLOX interposition layer, and the PLOX lambda.

- **PLOX Service:** The PLOX service interfaces directly with devices, handling all requests and messages. It is responsible for instantiating, scheduling and caching lambdas. Users interact with the service to manage access control for each device.
- **PLOX Runtime:** Manages the sandbox and provides runtime services for the PLOX interposition layer. This includes system calls that open new resources, e.g. open and connect. The runtime creates capabilities as needed that are allowed in the manifest and enforces user policy on sensitive data.
- **PLOX Interposition Layer:** The interposition layer (or shim) that redirects system calls and some libc functions. This layer translates system call requests to IPC and directly communicates with the PLOX runtime and uses the runtime to complete specific system calls.
- **PLOX Lambda:** Runs code belonging to an IoT device as defined by its previously approved and installed manifest. It runs on top of the shim which translates system calls to runtime requests to provide a more compatibility for existing third party code. Lambdas are able to publish and subscribe to events within PLOX as allowed by the manifest.

3.3 Network Isolation

PLOX enables wireless client isolation to restrict communication exclusively between the device and the WiFi router, disabling all other communications. Adding a devices requires user approval to pair the device by generating and installing TLS certificates on the device. The user also chooses a sensitivity label for device's data. PLOX uses TLS to encrypt communications between PLOX and device. It uses TLS client authentication to allow both PLOX and device verify each other's identify on all subsequent connections.

In this section we outline the main components of the PLOX system and its client. We introduce the interposition layer (shim), which is used to translate system calls to inter-process communications (IPC) to the PLOX runtime. The PLOX runtime administers the policy dictated by both the manifest of the application, as well as enforces user IFC policy throughout the system. We also describe the PLOX service, responsible for orchestrating the system, scheduling resources, providing an interface for external control, and providing end users controls.

As an example of a simple application when illustrating how our components work, we illustrate a security camera which notifies a user when a face is detected, the camera also allows for a user to rotate it even when they are outside the network. Listing 3.2 and 3.1 shows an example of the manifest and code in reference to this application, while Figure 3.2 diagrams how communication is delegated in the system. The overall view of how the components interact can be seen in Figure 3.1.

3.4 PLOX Client

The client library is used by IoT devices to communicate with the PLOX service to perform the initial device pairing, and send requests to be run on the server itself. Once the connection is created with the server, the device and PLOX service mutually authenticate one another and the client begins by firstly ensuring associated code (Listing 3.1) and manifest (Listing 3.2) are present on the system, one done the device listen for the events it is subscribed to along this channel. It is through these events that devices both receive information or requests and can become tainted.

Device taint can be mitigated through two options: proper restriction of device capabilities via the manifest, avoiding events (and data) that would cause tainting through the use of a "cloud needed" option. This option restricts sensitive data flowing to this component outlined in the manifest. On device installation PLOX prompts the user on whether this device should be considered sensitive before any communication or execution is allowed.

3.5 PLOX Service

The PLOX service runs on the router and can restrict the lambdas' memory usage through calls to limits and rctl or pin lambdas to specific cores through calls to cpuset. This can be useful, as PLOX may want to keep a dedicated core open for router traffic. The PLOX services manifest dictates the root policy that will be taken by future lambdas that use the PLOX Service, allowing for coarser grain policy over these lambdas if needed.

Once the initial handshake and necessary files are transferred to the router, the PLOX service begins the initialization of the lambda. To allow for communication and remote procedural calls (RPCs) between the lambda process and the main service a socket pair is

created. This channel acts as the main way for delivering and servicing requests from the PLOX Runtime.

A working directory is created for the child to act as its root directory. This directory will act as the only location a lambda may read or write to unless specified and approved in its manifest. Specific directories can be allowed by the PLOX service (e.g. library directories for dynamic linking) when required.

Before forking, the library of the lambda is opened (but not read) so that the lambda may have access to the open file descriptor. The PLOX service then forks and cleans up memory and restricts previously opened file descriptors (e.g. library file descriptors). The lambda enters capsicum mode and performs necessary bootstrapping for the language runtime and blocks on the channel to wait for its first event.

Reverse HTTP Service To handle external services that interact with internal devices, we use the devices manifest and keep a long running connection open to the outlined endpoints. These services send requests through this channel and trigger functions within the PLOX framework through allowed events. Using the example of Figure 3.2, a user wishing to move a camera through their cloud provider sends an event to the long running connection previously opened by the PLOX service. The endpoint sends a ”camera/move” event to the system which is passed through to the camera.

External endpoints are not limited to interacting with just devices as they are able to publish events to the event bus. Through the publication of these events, external endpoints can spawn lambdas defined in their application manifest (as long as they are properly subscribed to them). This is important when working with sensitive data, as it allows for an external entity to ask for computation to be done on sensitive internal data without becoming tainted itself (as communication flows one way).

Event Queues All lambdas have the ability to publish and subscribe to custom events of the PLOX service. When a device first registers itself with the PLOX service, its identifier is placed into the event queues that it wishes to be subscribed to. The event queues are organized by device and event type. When an event is fired by a running lambda, any relevant payload data is copied out, and placed within an event structure, which holds identifying information of the publisher. Events can be executed in two domains, either on the physical IoT device itself or through its associated serverless function. We allow physical devices to run their own events as this is crucial in getting the functionality that is required by them (such as taking a picture).

3.6 PLOX Runtime

The PLOX runtime supports the needs of the running lambda and provides APIs to receive and respond to events while also acting as the filter for received system call requests. We avoid the major pitfalls of security systems based on system call interposition, as described by Garfinkel [39], by following two design rules. First, all lambdas are single threaded and disallow forking. This eliminates many race conditions present with using other techniques including ptrace() and Seccomp [17].

Second, we use Capsicum's limiting of file descriptor privileges to implement finer grain policy and hand descriptors back to the sandboxed process. This avoids the need to replicate internal operating system state that is challenging to do and source of security problems, which is needed when interposing on all operations.

3.7 PLOX Shim Layer

Each lambda converts system calls and other API calls to RPC to the PLOX runtime using the interposition layer. These requests are made over a socket pair that is established between the two processes on creation of the lambda. The runtime replies with the results, and may transfer file descriptors that adds capabilities to the sandbox. The PLOX runtime allows or denies calls based on the manifest file and the list of user approved permissions.

The user is responsible for approving the specified permissions outlined in the manifest, but this process can be seen similarly to how users approve applications on their phone. If a lambda ever attempts to execute outside of their specified permissions the operation fails. As described earlier, one of the major pitfalls of Capsicum is its inability to run unmodified code. The PLOX shim is what supports this conversion, for example it supports turning open calls into openat calls with the dedicated root directory for the lambda being used as its argument. If absolute paths are used then an RPC is sent and the PLOX runtime filters this request based on the manifest sent with the application.

Curl as an example, must be given access to not only the file that describes nameservers, but also be given permission to access the specific IPs of these nameservers. Curl also requires cryptography libraries which must be defined and loaded by the service itself. PLOX exposes much of the inner workings of these libraries by showing exactly what files, connections and resources these applications require.

Enforcing Information Flow Control To enforce that data remains within the household, PLOX specially handles sockets that are created within the lambda. We cannot hand real connected sockets to lambdas as there is a possibility that a lambda opens a connection to an allowed host or IP, then opens a tainted file or attempts to communicate with

a sensitive service, this opens up the opportunity sensitive data to be sent through the opened channel. With the standard usage of capsicum we would be unable to stop the running lambda from using the connected socket as we had already handed it over.

To handle these cases PLOX has to take special care of sockets by firstly interposing on the socket based system calls like `socket`, `bind`, `listen`, `accept`, `getsockopt`, `setsockopt`, etc. PLOX starts by intercepting the socket creation call, registering that socket to a structure similar to a file descriptor table associated to that specific lambda. PLOX hands one end of a socket pair to the lambda rather than the socket itself.

When specified operations are called (e.g. `connect`) these operations are performed on the internally registered file descriptor that the runtime holds. This allows PLOX to have full control over this socket and the data that is sent between the lambda and the other connected service. PLOX can completely block communication upon seeing the device becoming tainted. Applications that try to bypass this (e.g., using the syscall function) service are never allowed to connect to anything as these permissions are never given to the process.

3.8 PLOX Developer API

We expose a special function sendEvent, which allows devices or lambdas to publish events to the system (Listing 3.1), it was important to minimally expose functionality in code to keep the code simple and closely resemble a deployment on Amazon Lambda [61].

Listing 3.1: Sample Python lambda which makes a call to an external API

```
from requests import put
import json

# Internal Camera Application Code
def notify(arguments, sendEvent):
    request = put("http://api.notification.com",
                  data={arg: arguments})
    return request.status_code != 200

# On Device code
def event(device_type, event_type, data):
    ...
```

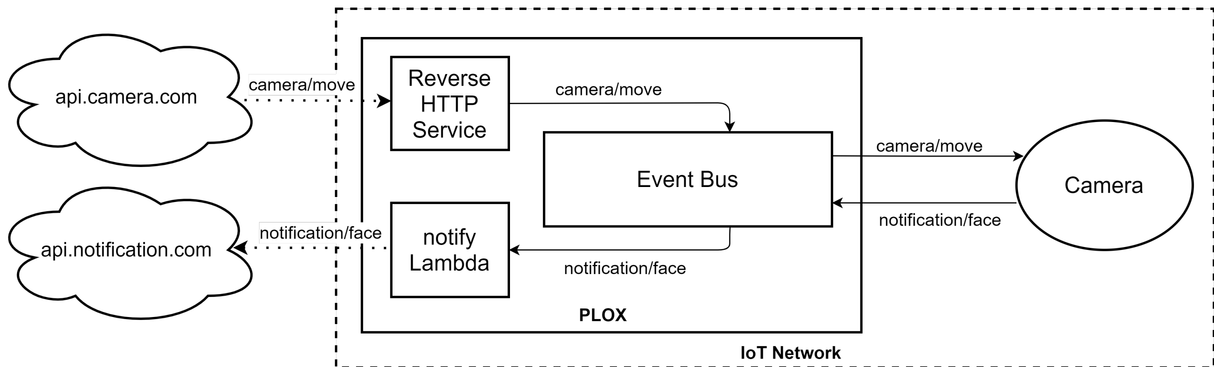


Figure 3.2: Example communication flows in a camera application, with associated manifest and code to dictate communication shown in Listing 3.2 and 3.1. Red arrows show external communication.

3.9 Scheduling

In order to ensure fairness and minimize the overhead of our caching algorithm, we use a simple least recently used (LRU) caching strategy. To minimize latency, lambdas are cached until a new request is received or if forcefully restarted due to becoming too tainted. These lambdas, do not consume CPU resources, as these processes are always blocked waiting for arguments which gives the PLOX service the ability to schedule lambdas as it sees fit. When a new request is received, the manifest is checked to identify what the required resources on the system will be (ie, memory, threads). The PLOX service then follows the LRU algorithm to evict lambdas that currently not running from cache until enough space is available. When lambdas are evicted, a special construction function is placed on the specific event queues that the lambda wished to be notified about.

The event scheduling follows the exact same pattern. Placed into a queue, events are executed and potentially evict lambdas if no space is currently available to execute. Once

reaching the front of the queue, the lambdas subscribed to the event are executed. Physical devices are notified on the channels currently connected with the PLOX service, allowing them to execute whatever functions are defined in their libraries. Given the wide variety of tasks that are offloaded, in the future it may be valuable to allow users to prioritize tasks. This might allow for a higher degree of control over the specified scheduling algorithm, and prevent large batch like workloads from blocking the short, higher priority tasks.

3.10 PLOX Security Policy

PLOX's dynamic capability model mitigates overtainting issues in IFC and simplifies the policy interface for users. Capabilities alone are insufficient as they are difficult for home users to parse leading to overpermissioning. Furthermore, they lack the context of the user's preferences about data sensitivity. PLOX's simplified IFC model relies on the capability model to reduce tainting and data-agnostic attack vectors. Used together, a hybrid solution solves the weaknesses of each by supporting the different needs of both developers and users.

3.10.1 PLOX Manifests

The PLOX manifest separates privileges based on the individual functions an application wishes to run. Listing 3.2 shows a simplified manifest from our PLOX-enabled IoT camera device. Each top-level key in the manifest represents either the device, an external endpoint, or a function name in the provided code. This way PLOX separates the privileges of internal functions and the privileges granted to external communication endpoints which require approval/trust by the user to access the home network.

Inside each object the developer lists the permissions required for full functionality.

```
device {
    publish [ notification/face ]
    subscribe [ camera/move ]
}
notify {
    subscribe [ camera/move ]
    connect {
        api.notification.com {
            socket {
                family: AF_INET
                type: SOCK_STREAM
                protocol: IPPROTO_TCP
            }
            ops: rw
            sockopts {
                SO_DOMAIN, rw
            }
        }
    }
    required_resources {
        memory: 10MB,
        timeout: 30s,
    }
    options {
        cloud_needed,
    }
}
api.camera.com {
    publish [ camera/move ]
}
```

Listing 3.2: PLOX Manifest for our security camera application

Devices list the events they can publish or subscribe to. External endpoints only accept specified events restricting sensitive data flowing from an event to an external API. Functions can specify resources (e.g. memory and CPU time), files, system calls they require and connections to external resources (e.g. a notification API).

No direct communications between an IoT device and the outside world is allowed. All sensitive data have to traverse a defined function. Furthermore, users will be presented with a list of all required external resources to select the ones they trust with sensitive data. The manifests can be automatically generated by PLOX under a developer "yell" mode which uses the PLOX interposition layer to capture all system calls and arguments invoked within a lambda. To determine the minimal capabilities required to run the functions, PLOX uses this captured data to map system calls and their arguments to a capability within the manifest.

3.10.2 PLOX's Sensitivity Label

PLOX users can associate a sensitivity label with each device joining the PLOX network. This label uniquely tags the sensitivity of the data produced by the device and restricts the functionality of lambdas that gain this label, restricting all forms of communication to untrusted external endpoints. Similar to devices, external application endpoints can be marked as trusted (by the users) to allow sensitive data to flow to them.

When a lambda is tainted by multiple labels, any endpoint it wishes to communicate with needs to be approved to access all labels. If a label is missing, the data flow will be prohibited. Sensitivity labels can be similarly added to all other objects on the network; allowing PLOX services to interact with internal services in the network (e.g. databases and network file systems) while enforcing IFC labeling. This results in PLOX tracking all

sensitive sources data and restricting flows to only trusted external endpoints.

User Comprehension PLOX's dynamic capability system presents two separate policy descriptions allowing PLOX to use a fine granularity capability system without the pitfalls of presenting this system to non-technical users. PLOX uses a minimal IFC policy to avoid common problems associated with describing application capabilities to non-technical users. PLOX's user interface presents a simplified set of choices to users only requiring them to label the sensitive sources of data (devices) in their home and the trusted sinks (endpoints).

Through the IFC system, PLOX presents a risk style policy to its users which has been shown to be more comprehensible in user studies [57]. For example, when a new device is installed in the PLOX network, the system presents the user with the following question:

"Are you comfortable with any information produced by this device to leave the home? Is this device sensitive to you?"

Further due to PLOX's unique location on the home router and its running of lambda sandboxes, PLOX has full context of how data is being used within the home allowing to easily present policy decisions to the user when sensitive data is being accessed. In this way PLOX shows what data could be possibly leaked to these endpoints using the risk category placed on the device and the context in which the device is acting. For example, suppose a security application wishes to send a notification to a user through a cloud service when a face has been detected by a camera deemed sensitive to the user. The following question is presented to the user:

"Application Face Detector wishes to use https://www.big-internet-company.com to function, but has access to Camera-Bedroom's sensitive data? Do you trust this website with getting this data?"

3.10.3 Dynamic Capability Model and the Serverless Model

Capability and IFC systems benefit from the a common use of a system call interposition layer as both require context of the arguments and type of system calls being called; this allows for capability systems to restrict the usage of these calls based of arguments passed while IFC uses them to propagate taint tracking to objects within the system and restrict the flow of data when needed.

PLOX uses a simplified IFC system that only tracks data produced by applications or devices, unlike prior systems that require more complicated labels and track all forms of communications. Systems like HiStar [68] use more sophisticated labeling schemes to better restrict the action space of processes as they execute, however our simplification of this system to only handle our sensitivity label can open this system to potential attack vectors. By supplementing our IFC system with capabilities, we can bound the functionality possible within these applications.

For example both capability and IFC systems interact closely with sockets. Capability systems limit the remote endpoints that a functions can communicate with, while IFC tracks connections and propagates IFC labels when connections are read or written to. Connections that violate *a can flow to* relation will be stopped by revoking the underlying capability. Socket handling is is explained more thoroughly in Section 3.7.

IFC requires the interposition layer to gain context on how system calls are used within a lambda. These systems calls such as `open`, can be used as a way to gain access to sensitive data, as such all functionality that could possibly taint a lambda must be accessible by our IFC system.

The PLOX interposition layer copies system call arguments through an inter-process communications (IPC) connection to the privileged PLOX daemon to avoid time-of-check

to time-of-use (TOCTOU) attacks that often occur with system call interposition [39]. Inspecting arguments allows our system to to grant or revoke resources before any operation. For example our ability to tie labels to the URL of a socket. This is a key feature at giving our capabilities context, as now arguments themselves are objects within PLOX.

Short Lived Processes PLOX uses the serverless paradigm in two ways: Firstly, the paradigm allows PLOX to easily restart lambdas once they become too tainted as these functions are often shorted lived and stateless. Functions restarted in this way, lose access to any sensitive data it could have accessed and taint is removed. Secondly it forces developers to separate their application into smaller functions allowing PLOX to have a finer grained manifest that can define the minimal functionality for each of these functions as to properly enforce privilege separation.

Communication From the Cloud Much of the communication previously mentioned addresses internal communication and how PLOX facilitates data flowing from the devices throughout the home, but also to external endpoints. PLOX uses a separation of privilege when handling external connections wishing to interact with internal devices. For example suppose a user wishes to move a camera through one of their cloud services.

To handle this case, developers express through the manifest an endpoint to accept requests and run associated functions within the PLOX environment. In the example manifest of Listing 3.2, the application developer has separated the privileges between the internal application of the "camera" that runs the function "block", and the function "move_camera" that runs when a request is received through an external connection to "api.camera.com".

When data is required to leave the home, developers expose endpoints in which data

can be sent through the associated application to this allowed endpoint. However, an interface is required to allow external endpoints to interact with internal devices. For example, a user may wish to notify a camera to turn or rotate from outside the network from their cloud provider. PLOX disallows direct access to these devices, so to handle this, developers express and register functions to the event system, and users are able to approve such external connections.

Chapter 4

Evaluation

We implement PLOX in ∼5200 lines of C++ on FreeBSD 12.1. We use FreeBSD's Capsicum capability system to implement the sandboxes.

Our evaluation aims to answer three primary questions.

- RQ1: Is PLOX effective in thwarting popular IoT attacks?
- RQ2: What is the performance impact and resource consumption of PLOX?
- RQ3: What is the overall overheard of PLOX?

Our experiments were conducted in a laboratory setting. We use a Raspberry Pi 3 Model B+ as the PLOX home router; The Raspberry Pi comes equipped with a quad core Cortex-A53 1.4GHz ARMv8 CPU with 1 GiB RAM. We use another Raspberry Pi for victim device prototype (See Section 4.1) and for the Home Assistant evaluation (see Section 4.2). To simulate a malicious station, we used another identical Raspberry Pi. Our evaluation shows that PLOX is effective in thwarting popular attacks in the IoT ecosystem.

Our performance evaluation shows that PLOX functions are comparable in execution time to Amazon Lambda and overall faster thanks to lower overhead and zero latency.

4.1 Effectiveness

To answer RQ1, we pick popular attacks in the IoT ecosystem and describe how PLOX thwarts them in practice. The attacks are summarized in Table 4.1; Column two reports the flaw exploited by the attackers, and column three shows that feature(s) provided by PLOX to defend against it.

Network Isolation As shown in Table 2, PLOX's network isolation is a key defense mechanism for mitigating exploits due to default configurations and firmware bugs (CVE-2015-2884, CVE-2019-12920, CVE-2018-8531, CVE-2020-7461). Adversaries outside PLOX's network cannot reach IoT devices within the network and must go through a triggered function in PLOX and only through an accepted endpoint – thus eliminating the chance for remote botnet exploitation. Attacks initiated and triggered within the PLOX network (e.g., a malicious app is deployed within an IoT device in PLOX) can be similarly thwarted; Any attack that requires to connect dynamically to different endpoints (e.g., email spamming) would require an update to the manifest, notifying the user again of this change.

To demonstrate how PLOX's network isolation can defend practically against popular attacks, we conducted a PoC attack on a secondary Raspberry Pi device (denoting our target victim device): we performed the following two experiments:

(1) We connected the target IoT device to our "Vanilla" IoT network (without deploying PLOX), and further set up the device's credentials to the manufacturer's default one and

configured it to be open to Telnet traffic. Next, we set up a "malicious" station outside of the network that mimics the behavior of a Mirai botnet server; it sends TCP SYN probes to the victim device (hardcoded), on the open Telnet port and then uses the default credentials to login to the device.

(2) In the second experiment, we deployed PLOX on the network, and setup the target IoT device to use the same configuration from experiment (1). As expected, the malicious station could login into the IoT device in the Vanilla network. With PLOX enabled, the server could not reach the victim device thanks to its network isolation feature. This confirms the effectiveness of the isolation feature enforced by PLOX to thwart popular botnet attacks.

Dynamic Capability Model As further shown in Table 4.1, PLOX's dynamic capability model defends against sensitive data leaks from the home network (e.g., CVE-2015-2884, fish monitor attack [13]). Adversaries cannot send information obtained from an infected IoT device like in the case of the fish monitor hack where an attacker infiltrated a casino's network through an overpermissioned and open fish monitor to access and leak sensitive client information. In conjunction with its network isolation feature, PLOX's dynamic capability model defends against sophisticated variants of botnet attacks which would attempt to deploy adversarial code. However, these botnets attacks require the ability to connect freely to external connections which would be prohibited by the manifest.

To illustrate how PLOX's dynamic capability model works to defend against information leak, we develop a PoC attack that aims to steal camera data on the network. We repeated our PoCs with and without enabling IFC in PLOX. Specifically, we performed the following two experiments:

(1) We deployed an overpermissioned malicious camera lambda (as seen in Figure 3.2)

on a target IoT device (a secondary Raspberry Pi device). The lambda subscribes to all events within the network and communicates with an external malicious endpoint to leak received event data (including sensitive camera data).

(2) In the second experiment, we enable IFC and set up the network identically to (1). We further taint the camera data as ”sensitive” using our model. As expected, the malicious endpoint could receive the camera events in the first experiment. With IFC enabled, the malicious lambda could not send out tainted sensitive data; external connections are dynamically disconnected thanks to the dynamic capability model – hence, disallowing any sensitive data leak.

We note that PLOX would prevent more complex data leak scenarios, i.e., where two (or more) devices co-operate to send out information. Device isolation prevents direct inter-device communication forcing all communication through the PLOX event system (which can be controlled and limited by the developer).

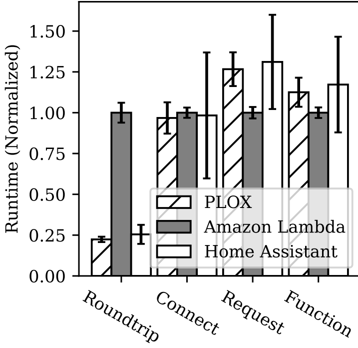
Exploited Public Endpoints PLOX cannot prevent the leakage of device data to an exploited endpoint, explicitly trusted by the user (e.g., CVE-2015-2884). However, PLOX does limit the sensitive data that can be leaked to the extent the endpoint has been trusted with device data. If a lambda were to acquire sensitive data from a separate device that the endpoint had not been given trust with, this would further taint the lambda past what was trusted, causing PLOX to cut off external communication.

4.2 Performance

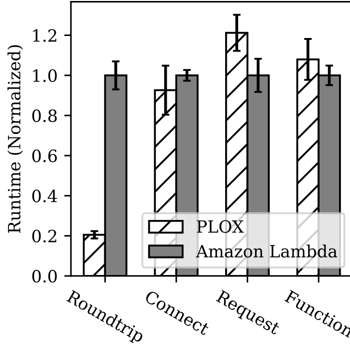
Figure 4.2 (b), (c) shows the overhead of the socket interposition. For the networking component of the micro benchmark, we had three servers running. One server acted as a

Attack	**Exploitation**	**Defense**
CVE-2015-2884 [7]	Default Password	Network Isolation
CVE-2015-2880 [6]	Default Password	Network Isolation
CVE-2019-12920 [16]	Default Password	Network Isolation
CVE-2018-8531 [14]	Firmware	Network Isolation
CVE-2020-7461 [10]	Firmware	Network Isolation
CVE-2020-11896 [23]	Firmware	Dynamic Capability Model
CVE-2020-11898 [19]	Firmware	Dynamic Capability Model
Mirai BotNet [11]	Configuration	Network Isolation, Dynamic Capability Model
CVE-2021-27561 [24]	Configuration	Network Isolation, Dynamic Capability Model
CVE-2021-27562 [25]	Configuration	Network Isolation, Dynamic Capability Model
Smart TV Botnet [5]	Configuration	Network Isolation, Dynamic Capability Model
CVE-2017-8867 [12]	No Encryption	Network Isolation, Dynamic Capability Model
CVE-2015-2884 [8]	Endpoint Vulnerability	Dynamic Capability Model
Fish Monitor Attack [13]	Overpermissioning	Dynamic Capability Model

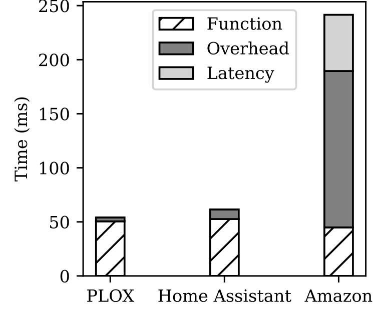
Table 4.1: A list of attacks found through the IoT ecosystem, how they were exploited and what part of PLOX protects against this attack.



(a) Python Runtime



(b) C++ Runtime

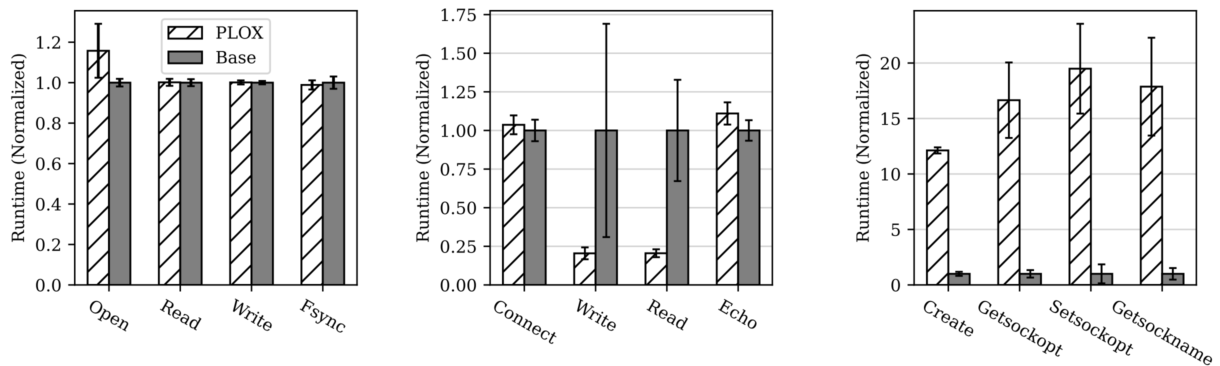


(c) Overhead costs of Evaluated Frameworks

Figure 4.1: This figure shows overall overhead from running a sample benchmark within each of the three frameworks. Home Assistant was excluded from the C++ evaluation due to the need to run the C++ code through its python interface.

sink for writes (similar to /dev/null). Another acted as a supplier of data by writing data to any of its connected clients, allowing for the client to always be able to read from its socket. The third server was a simple echo server allowing us to evaluate the full round trip cost of a mock request. For each of the servers, we wrote and read in 4 KiB chunks as this is far larger than most HTTP API GET requests that would be sent out, as these request are usually limited by the maximum URL size specific browsers will support (2 KiB on IE9 [29]).

We see that `create` and the meta-data operations in Figure 4.2 (c) are costly for PLOX as these operations require many arguments. We consistently see that, as the number of arguments or return values grow within a system call, the overhead for that call is increased due to the increased cost of IPC between the PLOX lambda and PLOX runtime. These



(a) File System Operations (4 kB) (b) Common Communication Operations (4 kB) (c) Construction and Meta-Data Operations

Figure 4.2: This figure shows a micro benchmark ran with and without (labeled Base) the PLOX framework. We ran networking calls that talked to three separate servers, a write server, a read server, and an echo server. We then measured time taken to do each of these calls, with the echo server representing a realistic view of the overhead of a standard API request that would be done by an IoT application.

functions are typically fast (10 μs) and the PLOX interposition turns these faster operations into expensive read and write socket calls. These operations however are typically more rare and sockets once fully constructed and connected will mainly be reading or writing. The `connect` system call in Figure 4.2 (b) sees a much lower decrease in performance (5%) as connect is a far more expensive call (3 ms) and the cost of argument passing becomes less of a relative overhead.

Another pattern we see is the reduced cost for both `read` and `write` and increased cost for the echo. This occurs because the process itself is not actually sending data to an externally connected destination, but rather to a local UNIX domain socket. This means

Service	**Average (ms)**
PLOX Cold*	89.6
PLOX Warm	55.5
Amazon Cold	803.3
Amazon Warm	259.6

Table 4.2: Total round trip time of the C++ benchmark for PLOX and Amazon Lambda. Cold start up is defined after PLOX has retrieved both the manifest and code from the device. These artifacts are retrieved during the initial handshake with a device so should always be on disk. The cost of cold start up is initial setup of the C++ runtime and the forking of the lambda process.

the write and read stay completely within the OS, never needing to interact with a device like the network interface.

The true cost of piping occurs with interactions with the echo server, as once the process is expecting a response an overhead is seen. This overhead is due to data having to be read from the PLOX runtime end of the socket pair and written to the real connected socket. The PLOX lambda must wait for these actions to occur, but also wait on the polling thread to wake-up and read the data from the other end of its pipe.

In Figure 4.2(a) we ran a simple read/write file benchmark on FFS. The benchmark wrote to a file in a 4 KiB chunk using random data (from /dev/urandom). We synced the file to force the writes to the disk then read back the written chunks. Our main objective is to reduce cache jitter that can occur when doing these file operations and properly evaluate our open system call interposition, while showing Capsicums zero overhead for standard file reads and writes. The open system call requires minimal arguments, so the message

passing overhead is minimal. We see no overhead with Capsicum, as minimal time is taken to check the Capsicum rights struct that is apart of every kernel file object.

Amazon Lambda and Home Assistant In Figure 4.1 we compare against two frameworks used for IoT – Amazon Lambda (with default configuration) and Home Assistant. We compare against Amazon Lambda in both our C++ and Python runtime, but due to Home Assistant being built in Python, there was no way for us to call C++ libraries without first going through Python itself which would be an unfair comparison.

For our benchmark we had the process first create a socket, connect to an external service (in this case www.google.com), make a simple GET request, and wait for the response. It would then format the timing data for each call into a JSON object to be returned to the user. For PLOX, we setup two devices on the local network, one device being the Raspberry Pi which ran PLOX, and another desktop acting as the client.

For Amazon, we setup the Lambda to be triggered using their API-Gateway as we found this to be the fastest trigger. Using the AWS command line interface (CLI) to trigger functions had a substantial overhead due authentication and the CLI itself. We used CURL to send a GET request to this API endpoint and had curl output the round trip time for this request.

In order to test the connection between devices in Home Assistant, we use the event bus and built two custom components. The first component was responsible for logging the start time of the test, then firing an event to start up the second component. Upon this event firing, the benchmark would execute on the second component and log timings to home assistant. When the benchmark finished, it sent out a done event back to the first component. The first component then logged a timestamp. We used these timestamps to measure the round trip for this benchmark.

Figure 4.1 (a) we see calls within Home Assistant being highly volatile and more expensive than both PLOX and Amazon Lambda. This volatility is likely from the networking layer that Home Assistant uses. Home Assistant is bulkier due to its implementation within Python, leading to larger overheads. This can be seen in every function call and contributes to higher volatility and a higher overall round-trip cost for Home Assistant.

In Figure 4.1 (c), we see a breakdown of the overhead cost of each framework. To measure overhead cost we took the average round trip time of warmed functions, and subtracted total function time and latency. Both PLOX and Home Assistant latency's were negligible (<1 ms) so we considered this to be 0. We see PLOX is running 5 times faster than Amazon, this is from two areas: latency and Amazon's need for isolated scaling.

Amazon's framework requires the dispatching of millions of functions to hundreds of servers within the data centre. Amazon Lambda has higher than normal costs as users need kernel space isolation to eliminate users from peeking at other user workloads [27]. This framework overhead is close to 150 ms. Although these features are important, we believe features like auto-scaling and kernel space isolation are less useful within the home IoT environment.

The advantage of Amazon lambda is that this overhead is not on IoT devices themselves, but note that Amazon's functions were running only 5-10% faster then on the Raspberry Pi showing that resources given to these functions are quite limited. In Table 4.2 we see a look at our C++ benchmark in relation to cold starts. This outlines the cost for infrequent requests which is reduced by the use of PLOX due to not having to start up bulkier virtual machines. PLOX has over a 700 ms advantage in terms of latency when running infrequently used functions.

Chapter 5

Future Work

In this chapter, I will outline the directions we are taking PLOX, including some notable future research directions that could be pursued outside of this project.

At this time, a number of smart applications have been prototyped on PLOX. These are a small number of the required applications needed to test the robustness of PLOX. Furthermore, continued testing of a large scale deployment of PLOX may help to support the overtainting prevention claims, and highlight the stability of the framework as a reasonable solution for the smart home.

Qualitatively, both developers that have worked directly on PLOX smart applications have strongly preferred the development experience over other frameworks. While both developers have also designed PLOX, the manifest auditing tool, as well as the simple APIs, made building smart applications on PLOX much faster and easier than on other tested platforms, including Home Assistant, Amazon Lambda, and Azure IoT Edge. Though this is not a central goal of the project, developer experience may be worth exploring through a

user study or qualitative comparison of smart application implementations. An exploration into more complicated application deployments may also help compare the simplicity and complexity of these frameworks through analysis of the required lines of code and required configuration files.

A user study targeting smart home users could also provide more context into if the taint labeling system is intuitive. Although it has been shown that user context is essential in the smart home [49, 51], a comparison to other smart hubs or smart home frameworks could improve the design of PLOX, and help build an understanding of how misconfiguration happens within the smart home. Further research into this area could also help to improve the design of other frameworks.

Another interesting design direction for PLOX is remote patching of devices. Given the strong network segmentation and device isolation PLOX imposes, updating and controlling the firmware of physical devices is a logical extension of PLOX. This would further the completeness of PLOX as a solution for the smart home.

Another consideration to better allow PLOX to solve the security concerns of the smart home is the usage of role based taint labeling. Currently, PLOX does not support user roles beyond the administrator or trusted devices, sensitive applications and normal applications. By exploring guest labels, or allowing users to mark devices are accessible to a variety of user groups, a number of privacy concerns could be addressed. It could also help support the arguments found in research by [70] that smart homes with multi user environments require further options to prevent kids or guests from accessing sensitive or critical home functionality. This could also be a strong direction for future research, given the simplicity of configurations required by users and natural extension of data controls within the smart home.

Finally, PLOX could be extended to leverage mesh networks. Given the already distributed and concurrent nature of PLOX, using multiple nodes of computation within the smart home is a natural extension. By increasing the number of compute nodes, the home would become much less prone to failure. In the current deployment, there is a single point of failure, leading to a strong incentive for attackers to compromise the device. This redundancy would also allow for tasks to be scheduled between computation nodes, potentially increasing the types of tasks that could be performed by PLOX. Dynamic scheduling could be further investigated to account for the variety of task sizes that may be best scheduled between PLOX instances.

5.1 Limitations

Limitations on the Developer There are inherent engineering roadblocks that may make PLOX difficult to adopt. PLOX may limit the agency of developers due to our strict usage of an event system for inter-device communication and our usage of the serverless paradigm. While this limitation could impact secondary functionalities provided by the devices, it should not interfere with its main functionality.

Central Point of Computation PLOX uses the router as its only point of computation. Given the increased risk of PLOX being deployed on the home router, attackers would be heavily incentivised to exploit any security vulnerabilities found within the router or PLOX to disrupt the home network. This increased reliance on the router may lead to a larger draw to exploit said vulnerabilities.

Removing Device Taint PLOX lacks the ability to untaint devices themselves. Once a device becomes tainted by receiving an event from a sensitive device (rather than through an associated function), PLOX cannot safely deem a device untainted as this would require the ability to wipe devices. As such we cannot guarantee data is not being saved to the device itself. This is an issue outlined by OWASP as a lack of physical hardening on the devices themselves. As devices adopt this feature, PLOX would be able to use this to remove device taint.

Chapter 6

Conclusion

This paper presents PLOX, a framework that provides a serverless function programming environment for IoT device developers. PLOX introduces the *dynamic capability model*, combining a fine grain capability system (FreeBSD's Capcisum) and a simplified IFC model. This combined model addresses the weaknesses of both capability systems and IFC.

PLOX uses a serverless paradigm to allow for the separation of privileges between each of the functions that make up an IoT application. PLOX is able to provide greater context to policy divisions due to facilitating safe sandboxes for third party applications and its placement on the home router, which allows for it to fully isolate devices on the network.

In our evaluation we showed how PLOX can defend against varying exploits and vulnerabilities within the IoT ecosystem such as Botnets which exploit default configurations and old firmware. We also evaluate PLOX's ability to protect sensitive user data through the use of an adversarial application. Finally, PLOX outperformed both a cloud solution, Amazon Lambda and an open source smart hub solution, Home Assistant.

References

[1] Apple homekit. URL: https://www.apple.com/ios/home/.

[2] Home assistant. https://github.com/home-assistant.

[3] Namespaces. https://man7.org/linux/man-pages/man7/namespaces.7.html.

[4] *zones(5) Standards, Environments, and Macros*, SunOS 5.10 edition, April 2004.

[5] REFRIGERATOR HACKED: Here's The Biggest Problem Facing The Internet Of Things. https://www.businessinsider.com.au/hackers-use-a-refridgerator-to-attack-businesses-2014-1?op=1, 2014.

[6] Cve-2015-2880. https://www.cvedetails.com/cve/CVE-2015-2880/, 2015.

[7] Cve-2015-2884. https://www.cvedetails.com/cve/CVE-2015-2884/, 2015.

[8] Cve-2015-2884. https://www.cvedetails.com/cve/CVE-2015-2884/, 2015.

[9] Cve-2015-2886. https://www.cvedetails.com/cve/CVE-2015-2886/, 2015.

[10] Cve-2020-7461. https://www.cvedetails.com/cve/CVE-2020-7461/, 2015.

[11] Friday's massive ddos attack came from just 100,000 hacked iot devices, Oct 2016. URL: https://thehackernews.com/2016/10/ddos-attack-mirai-iot.html.

[12] Cve-2017-8867. https://nvd.nist.gov/vuln/detail/CVE-2017-8867, 2017.

[13] How a fish tank helped hack a casino. https://www.washingtonpost.com/news/innovations/wp/2017/07/21/how-a-fish-tank-helped-hack-a-casino/, 2017.

[14] Cve-2018-8531. https://www.cvedetails.com/cve/CVE-2018-8531/, 2018.

[15] Owasp iot top 10. https://wiki.owasp.org/index.php/OWASP_Internet_of_Things_Project#tab=IoT_Top_10, 2018.

[16] Cve-2019-12920. https://www.cvedetails.com/cve/CVE-2019-12920/, 2019.

[17] *SECCOMP(2) Linux Programmer's Manual*, linux man-pages 5.06 edition, November 2019.

[18] *CGROUPS(7) Linux Programmer's Manual*, linux man-pages 5.06 edition, April 2020.

[19] Cve-2020-11898. https://www.cvedetails.com/cve/CVE-2020-11896/?q=CVE-2020-11898, 2020.

[20] Mqtt: Message queuing telemetry transport. http://mqtt.org/, March 2020.

[21] Sb-327 information privacy: connected devices. https://leginfo.legislature.ca.gov/faces/billNavClient.xhtml?bill_id=201720180SB327, 2020.

[22] Capture: Centralized library management for heterogeneous iot devices. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-han.

[23] Cve-2020-11896. https://www.cvedetails.com/cve/CVE-2020-11896/?q=CVE-2020-11896, 2021.

[24] Cve-2021-27561. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-27561/, 2021.

[25] Cve-2021-27562. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-27562/, 2021.

[26] Massive camera hack exposes the growing reach and intimacy of American surveillance. https://www.washingtonpost.com/technology/2021/03/10/verkada-hack-surveillance-risk/, 2021.

[27] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. Firecracker: Lightweight Virtualization for Serverless Applications. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 419–434, 2020.

[28] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. *Acm Sigplan Notices*, 49(6):259–269, 2014.

[29] Microsoft Corporation. Maximum url length. https://support.microsoft.com/en-ca/help/208427/maximum-url-length-is-2-083-characters-in-internet-explorer.

[30] Microsoft Corporation. Microsoft vunerability cve-2019-1234. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-1234, March 2020.

[31] Microsoft Corporation. Microsoft vunerability cve-2019-1372. https://portal.msrc.microsoft.com/en-US/security-guidance/advisory/CVE-2019-1372, March 2020.

[32] Jamie Davies. Uk government unveils new details for iot security standards. https://telecoms.com/505582/uk-gov-unveils-new-details-for-iot-security-standards/, 2020.

[33] Dorothy E. Denning. A lattice model of secure information flow. *Commun. ACM*, 19(5):236–243, May 1976. doi:10.1145/360051.360056.

[34] Petros Efstathopoulos, Maxwell Krohn, Steve VanDeBogart, Cliff Frey, David Ziegler, Eddie Kohler, David Mazieres, Frans Kaashoek, and Robert Morris. Labels and event processes in the Asbestos operating system. *ACM SIGOPS Operating Systems Review*, 39(5):17–30, 2005.

[35] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[36] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 636–654. IEEE, 2016.

[37] Earlence Fernandes, Justin Paupore, Amir Rahmati, Daniel Simionato, Mauro Conti, and Atul Prakash. Flowfence: Practical data protection for emerging iot application frameworks. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 531–548, 2016.

[38] Chenglong Fu, Qiang Zeng, and Xiaojiang Du. Hawatcher: Semantics-aware anomaly detection for appified smart homes. In *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[39] Tal Garfinkel. Traps and Pitfalls: Practical Problems in System Call Interposition Based Security Tools. In *NDSS*, volume 3, pages 163–176, 2003.

[40] Google. Android manifest. https://developer.android.com/guide/topics/manifest/manifest-intro, March 2020.

[41] Sudhakar Govindavajhala and Andrew W Appel. Windows access control demystified. *Princeton university*, 2006.

[42] Toms Hardware. 75 percent of bluetooth smart locks can be hacked. https://www.tomsguide.com/us/bluetooth-lock-hacks-defcon2016,news-23129.html, 2016.

[43] Galen C Hunt and James R Larus. Singularity: rethinking the software stack. *ACM SIGOPS Operating Systems Review*, 41(2):37–49, 2007.

[44] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Zhuoqing Morley Mao, Atul Prakash, and SJ Unviersity. Contexlot: towards providing contextual integrity to appified iot platforms. In *NDSS*, 2017.

[45] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *SANE 2000*, May 2000.

[46] Patrick Gage Kelley, Sunny Consolvo, Lorrie Faith Cranor, Jaeyeon Jung, Norman Sadeh, and David Wetherall. A conundrum of permissions: installing applications on an android smartphone. In *International conference on financial cryptography and data security*, pages 68–79. Springer, 2012.

[47] Maxwell Krohn. Building secure high-performance web services with okws. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '04, page 15, USA, 2004. USENIX Association.

[48] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M Frans Kaashoek, Eddie Kohler, and Robert Morris. Information flow control for standard os abstractions. *ACM SIGOPS Operating Systems Review*, 41(6):321–334, 2007.

[49] Josephine Lau, Benjamin Zimmerman, and Florian Schaub. Alexa, are you listening? privacy perceptions, concerns and privacy-seeking behaviors with smart speakers. *Proceedings of the ACM on Human-Computer Interaction*, 2(CSCW):1–31, 2018.

[50] Linux. Apparmor. https://apparmor.net/.

[51] Nathan Malkin, Julia Bernd, Maritza Johnson, and Serge Egelman. "what can't data be used for?" privacy expectations about smart tvs in the us. In *Proceedings of the 3rd European Workshop on Usable Security (EuroUSEC), London, UK*, 2018.

[52] mitchelljuanita. People Can't Vacuum Or Use Their Doorbell Because Amazon's Cloud Servers Are Down, Nov 2020. URL: https://eminetra.com.au/people-cant-vacuum-or-use-their-doorbell-because-amazons-cloud-servers-are-down/74505/.

[53] The Hacker News. Linux trojan using hacked iot devices to send spam emails. https://thehackernews.com/2017/09/linux-malware-iot-hacking.html, 2017.

[54] Nokia. Nokia threat intelligence report. https://onestore.nokia.com/asset/210088, 2020.

[55] OpenHAB. Openhab. https://github.com/openhab.

[56] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *12th USENIX Security Symposium (USENIX Security 03)*, Washington, D.C.,

August 2003. USENIX Association. URL: https://www.usenix.org/conference/12th-usenix-security-symposium/preventing-privilege-escalation.

[57] Amir Rahmati, Earlence Fernandes, Kevin Eykholt, and Atul Prakash. Tyche: A risk-based permission model for smart homes. In *2018 IEEE Cybersecurity Development (SecDev)*, pages 29–36. IEEE, 2018.

[58] Nirupam Roy, Haitham Hassanieh, and Romit Roy Choudhury. Backdoor: Making microphones hear inaudible sounds. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, page 2–14, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3081333.3081366.

[59] Andrei Sabelfeld and David Sands. A per model of secure information flow in sequential programs. In S. Doaitse Swierstra, editor, *Programming Languages and Systems*, pages 40–58, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[60] Samsung. Smart things hub. https://www.smartthings.com/gb/products/smartthings-hub, September 2020.

[61] Amazon Web Services. Amazon lambda. https://aws.amazon.com/lambda/, March 2020.

[62] Vijay Sivaraman, Hassan Habibi Gharakheili, Arun Vishwanath, Roksana Boreli, and Olivier Mehani. Network-level security and privacy control for smart-home IoT devices. In *2015 IEEE 11th International conference on wireless and mobile computing, networking and communications (WiMob)*, pages 163–167. IEEE, 2015.

[63] Saleh Soltan, Prateek Mittal, and H Vincent Poor. BlackIoT: IoT botnet of high wattage devices can disrupt the power grid. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 15–32, 2018.

[64] Keir Thomas. https://www.pcworld.com/article/214775/microsoft_cloud_data_breach_sign_of_future.html, Dec 2010.

[65] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, XianZheng Guo, and Patrick Tague. Smartauth: User-centered authorization for the internet of things. In *Proceedings of the 26th USENIX Conference on Security Symposium*, SEC'17, page 361–378, USA, 2017. USENIX Association.

[66] Xueqiang Wang, Yuqiong Sun, Susanta Nanda, and XiaoFeng Wang. Looking from the mirror: evaluating iot device security through mobile companion apps. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1151–1167, 2019.

[67] Robert NM Watson, Jonathan Anderson, Ben Laurie, and Kris Kennaway. Capsicum: Practical capabilities for unix. In *USENIX Security Symposium*, volume 46, page 2, 2010.

[68] Nickolai Zeldovich, Silas Boyd-Wickizer, Eddie Kohler, and David Mazières. Making information flow explicit in histar. *Communications of the ACM*, 54(11):93–101, 2011.

[69] Nickolai Zeldovich, Silas Boyd-Wickizer, and David Mazieres. Securing Distributed Systems with Information Flow Control. In *NSDI*, volume 8, pages 293–308, 2008.

[70] Eric Zeng and Franziska Roesner. Understanding and improving security and privacy in multi-user smart homes: a design exploration and in-home user study. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 159–176, 2019.