Secure and Practical Splitting of IoT Device Functionalities

Han Zhang

CMU-CS-23-129

August 2023

Computer Science Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Yuvraj Agarwal, Co-chair
Matt Fredrikson, Co-chair
Vyas Sekar
Alec Wolman, Microsoft

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by: Air Force Research Laboratory under award number 87501520281; Intelligent Automation, Inc., under award number 24331; National Science Foundation under award numbers 1704542, 1564009, and 1943016; Office of Naval Research under award number 000141812619; US Army Contracting Command under award number 911NF20D0002; Defense Advanced Research Projects Agency under award number 00112020006; and University of Wisconsin-Madison under award number 0000001468.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the United States Government or any other supporting entity.

Keywords: Internet-of-Things, Security, Privacy, Access Control, Computation Offloading, Verification

Abstract

Internet-of-things (IoT) devices have rapidly gained popularity in people's daily lives. While these devices provide many smart functionalities and enable new applications, they raise several security and privacy concerns and practical operational challenges for device users and vendors. With their growing adoption and sheer deployment volumes, IoT devices have become attractive targets for attackers, and many recent security incidents have had broad and serious impacts. Meanwhile, IoT devices can collect a wide range of personal data through sensors and ubiquitous placements. It is an important challenge for device vendors to protect users' privacy and manage access control properly. In addition, device vendors have to invest heftily in cloud infrastructures to mitigate the limited computation resources on devices. With more and more devices installed in the future, the demand for more computation will also increase.

We attribute these concerns and challenges of future IoT deployment partially to the predominant monolithic design of IoT devices and applications. Device vendors must take responsibility for many tasks, including managing device security, protecting user data privacy, and maintaining cloud infrastructure efficiently. However, device vendors mainly focus on building compelling applications to attract more users. Therefore, they have to prioritize certain tasks over other responsibilities, given their limited engineering resources. As a result, the current monolithic design leads to many vulnerabilities, security incidents, and inefficiencies.

In this thesis, we argue that by combining formal security analyses and performance optimizations, we can achieve a separation of concerns and offload many high-level IoT functionalities to third-party services, improving IoT devices' security and privacy while minimizing performance impacts. In particular, we design three systems — TEO, CAPTURE, and VERISPLIT — to showcase the benefits of functionality splitting. Each of these systems delivers strong security and practicality guarantees. We demonstrate their feasibility and effectiveness with prototype implementations and evaluations using various smart home applications. Overall, these systems present several novel techniques towards enabling secure and practical functionality splitting for IoT devices.

Acknowledgments

First and foremost, I would like to express my gratitude to my advisors, Professors Yuvraj Agarwal and Matt Fredrikson, for their countless guidance during my graduate studies. It's a privilege to work with both of them and learn so much from their advice and feedback over the years. This journey has helped me become a researcher and given me many valuable experiences in life.

I would like to thank other members of my thesis committee: Professor Vyas Sekar, for always being accessible for collaboration and providing constructive comments and advice on many research projects. Dr. Alec Wolman, for kindly agreeing to join my committee and providing valuable feedback on the dissertation.

I am fortunate to have worked with many researchers and faculty members on various projects: Arthur Azevedo de Amorim, Anupam Datta, Jason Hong, Limin Jia, Carlee Joe-Wong, Corina Pasareanu, and Justine Sherry. Their collaborations helped broaden my research interests and allowed me to continue learning new knowledge throughout these years.

I would like to thank the research and support staff in the Computer Science Department and Software and Societal Systems Department: Deb Cavlovich, Catherine Copetas, Jennifer Cooper, Tracy Farbacher, Jenn Landefeld, Angy Malloy, Erin Murray, and Charlotte Yano. I appreciate their help in making everything go smoothly around the departments.

I want to thank many long-time friends who made my graduate school journey really enjoyable: Matt Butrovich, Graham Gobieski, Jack Kosaian, Michael Rudow, and Giulio Zhou. Hanging out with them truly makes time fly by. Moreover, I am fortunate to have met and collaborated with many talented graduate students and friends: Abhijith Anilkumar, Nirav Atre, Krishna Bagadia, Sudershan Boovaraghavan, Emily Black, Christopher Canel, Chen Chen, Mike Czapik, Mihir Dhamankar, Clement Fung, Haojian Jin, Jeremy Lacomis, Tianshi Li, Klas Leino, Antonis Manousis, Soo-Jin Moon, Prasoon Patidar, Joseph Severini, Ke Wu, Daehyeok Kim, Dohyun Kim, Thomas Kim, Zifan Wang, Samuel Yeom, Yuhang Yao, Haozhe Zhou, Chi Zhang, and Zichao Zhang.

Moreover, I would like to thank my undergraduate mentors, who inspired me to pursue a PhD degree and conduct research: Harsha V. Madhyastha, Peter Honeyman, and Michalis Kallitsis. Their support, advice, and kind guidance lead to the beginning of this journey.

Finally, I would like to express my deepest gratitude to my family, who have supported me unconditionally in pursuing my interests and passions with confidence and freedom over the years.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Internet-of-Things (IoT) devices have rapidly gained popularity in modern smart homes, buildings, and shared spaces [30, 74, 229]. IoT devices are equipped with diverse sets of sensors and actuators, enabling many new applications and functionalities. Their prevalence also begins to raise more and more security and privacy concerns. Numerous news reports of high-profile security breaches involving these devices [12, 181, 188, 233] create a troubling outlook for the future of IoT deployments as the number of devices continues to grow [6, 229]. Due to their closed-source nature, end-users must count on the device vendors to promptly patch their device's firmware to mitigate security vulnerabilities.

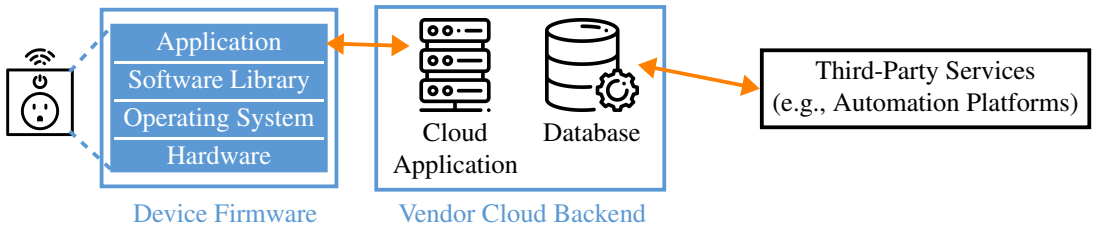
Moreover, IoT devices can collect a large variety of sensitive data from their sensors (e.g., audio, video, behavior), raising additional privacy concerns for device users [22, 64, 93]. To make matters worse, IoT devices are often heavily resource-constrained and must rely on the vendor's cloud backend to augment their limited computation capability. Offloading to the vendor's cloud not only induces extra costs to end-users [189, 224], but also exacerbates security and privacy concerns. Device users must fully trust IoT vendors with a long list of responsibilities, such as securing their devices from Internet attackers, protecting their data and managing access control properly, and meticulously managing cloud infrastructures to reduce users' cost burdens.

Currently, IoT devices have monolithic designs — device vendors must provide full-fledged solutions for the entire device software stack, including firmware, companion apps, applications, and cloud backend for various services. Given limited engineering resources, vendors must prioritize certain tasks, such as developing new compelling applications, over other responsibilities. As a result, they may sometimes fall behind in applying security patches and securing their applications, leading to the security incidents mentioned above.

Several IoT frameworks have been proposed aiming to alleviate the drawbacks of monolithic designs [8, 140, 165]. These frameworks advocate an approach to help IoT vendors with managing low-level device firmware and software libraries, liberating them to focus on building high-level applications. Although these approaches are promising, they still lack several critical features, such as supporting heterogeneous hardware platforms, third-party libraries' flexibility, and strong security and privacy guarantees.

In this thesis, we argue that by applying formal security analyses and secure system designs with appropriate optimizations, we can provide **secure and practical** solutions to help IoT devices become more secure, privacy-preserving, and cost-efficient. This dissertation introduces

Figure 1.1: Current monolithic design of IoT devices. Device vendors must implement and integrate all components in blue, including device firmware and cloud backend applications. Orange lines indicate communication over the public Internet.



several new system frameworks with the goal of relieving IoT vendors of many responsibilities. The general approach is to offload many common functionalities in IoT devices (e.g., software library updates, ownership and access management) to third parties. This concept, which we call **functionality splitting**, enables many new system designs; meanwhile, we applied a number of security analysis techniques to formally ensure the security and privacy guarantees of our new system designs.

In particular, we present the designs, implementations, and security analyses of three frameworks for functionality splitting. First, TEO helps protect smart home users' data by enabling IoT vendors to offload the task of enforcing access controls with a new ownership model. Second, CAPTURE alleviates the prevalent security challenges of mismanaged software libraries by offloading the responsibility of managing third-party libraries. Finally, VERISPLIT reduces IoT vendors' burdens and costs of maintaining cloud infrastructure by offloading machine learning inferences to local devices. All of these systems employ a combination of applied formal security analyses and application-specific optimizations to ensure a secure and practical design.

Thesis statement: *By combining formal security analyses and performance optimizations, we can enable IoT device vendors to efficiently and securely offload many functionalities to improve the overall IoT deployment's security, privacy, and cost-efficiency.*

1.1 Current Monolithic Device Design

With the proliferation of IoT devices, more and more manufacturers have developed smart devices for their product lines. Today, many IoT devices have adopted a monolithic design for their software stacks and cloud companion applications, as illustrated in Figure 1.1 (based on teardown blogs [4, 5, 54, 55, 97]). IoT vendors directly control both firmware and cloud backend, retaining high flexibility in development.

IoT vendors are responsible for many tasks and must implement and maintain multiple components in the software stack. Let's take a look at the stack from the bottom up. Device vendors have to design the physical hardware with various microcontrollers (MCUs) and operating systems (e.g., Linux, RTOS). They must procure a list of third-party software library dependencies to be flashed with the firmware. They must develop high-level application logic in the device firmware and manage communication with backend systems for data storage and functionality.

Device vendors must also manage the cloud backend applications to prevent security and privacy incidents and optimize performance. In addition, the vendor's backend needs to extend support for third-party services (e.g., home automation platforms) to expand interoperability with other devices in the smart home ecosystems. However, interfacing with third-party services further complicates the backend system design and increases the difficulty of protecting users' security and privacy.

In summary, the current monolithic design imposes an extensive list of responsibilities on IoT vendors' plates. Vendors must implement or integrate many functionalities to fulfill these responsibilities. Given the limited engineering resources, whether IoT vendors can maintain all these functionalities satisfactorily is questionable. Unfortunately, as indicated by many security incidents and our own analysis conducted in this thesis, the answer is "no".

1.2 Secure and Practical Functionality Splitting

In this thesis, we propose the designs of three novel systems facilitating *functionality splitting* in IoT devices. We want to help IoT vendors delegate selected functionalities to third parties securely and practically. We incorporate many formal analysis techniques to provide strong security guarantees during the design process. In addition, we meticulously investigate optimization opportunities to minimize performance impacts. Splitting these functionalities helps IoT vendors reduce the scope of their responsibilities and minimize workload; meanwhile, our new systems help improve the overall IoT deployment's security, privacy, and cost-efficiency. In particular, each of the three systems identifies new opportunities in splitting functionalities from different layers of the device software stack:

TEO: Application Access Control and Ownership Management. The first system, TEO, provides a solution to splitting application-level functionalities for IoT vendors. TEO addresses challenges in managing access control for users' private data and ownership of shared IoT devices. Currently, IoT vendors store users' data in their first-party backend services. Therefore, they must be responsible for properly managing user data access control. To make matters worse, more and more devices are being deployed in shared or semi-private spaces beyond users' homes, which raises new concerns on how to decide who the current users are and how to protect their data accordingly.

TEO presents a suite of protocols designed to address these emerging challenges in managing IoT device access controls. TEO achieves several high-level design goals. Specifically, it can support frequent changes in temporary device ownership efficiently through fine-grained segmentation of temporal ownership sessions. It enables variable-sized groups to collectively control and manage shared devices by leveraging Shamir Secret Sharing [194] to distribute control across stakeholders. Finally, to enable data access revocation with low communication overhead, TEO incorporates key homomorphic encryption [28, 217] for third-party data storage providers to rotate encryption keys without accessing plaintext contents.

To ensure TEO's security guarantees and mitigate potential design-level vulnerabilities, we utilize a protocol verification tool ProVerif [26] to formally analyze TEO protocol specification against our well-stated security goals. We address multiple modeling challenges and extend

ProVerif with a new template language for easy encoding. After an iterative process of verification and revisions, we eliminated many design vulnerabilities. We concluded with a final protocol suite satisfying security properties, including secrecy, mutual authentication, resilience to data spoofing, and effective revocation for variable group sizes.

CAPTURE: Third-Party Software Library Maintenance. The second system, CAPTURE, dives deeper underneath the software stack and addresses issues in mismanaged third-party library dependencies in many IoT devices. CAPTURE proposes a new framework for device firmware development that enables IoT vendors to delegate the responsibility of maintaining and updating libraries to a trusted third party (a local hub). Currently, the monolithic device design entrusts IoT vendors always to keep their devices (along with all dependent libraries) up-to-date to eliminate potential attacks leveraging publicly-known exploits. Unfortunately, as we demonstrate in our analysis results, this is a tall order, and even the most well-known IoT vendors often fall short of satisfactorily fulfilling this responsibility.

CAPTURE enables IoT vendors to offload library management tasks to a trusted entity, reducing the IoT attack surfaces stemming from vulnerable, outdated third-party libraries. CAPTURE centralizes library management responsibilities by introducing a trusted local hub. CAPTURE-enabled devices partition their software across the device and a corresponding driver on the hub. The hub maintains common third-party libraries and keeps them updated. In addition, the hub enforces strong network and process isolation to prevent compromised devices from escalating attacks.

We formalize reasons about CAPTURE's system design against both internal and external threats for our security analysis. We want to ensure CAPTURE is secure by construction and can uphold various security goals, such as strong isolation and minimal attack surfaces.

VERISPLIT: Efficient Use of Compute Infrastructure. Finally, the third system, VERISPLIT, explores functionality splitting opportunities at the hardware and computation infrastructure level. In this case, VERISPLIT relieves IoT vendors from managing individual cloud backends for transitory computation demands by offloading them to local devices. Currently, IoT devices with limited local computing power must utilize first-party cloud services for heavyweight computation demands. In VERISPLIT, they can seek helps from other devices deployed in the same user's home. Although computation offloading has been a long-standing research topic and widely adopted in practice, there lacks a solution for secure and efficient offloading for IoT devices, given their limited computing and communication capability. By focusing on the emerging applications of machine learning models, we provide specialized solutions for secure and private inference offloading with practical performance overheads.

VERISPLIT presents solutions to address various security and practical concerns for offloading inferences to third-party hardware. First, the offloading device may want to protect the privacy of the inference data, which could contain smart home users' personal information, before sharing them with third-party workers. Second, the offloading device could be hesitant about sharing proprietary machine-learning models with third parties, which is necessary for workers to perform the computation. Third, the offloading device may need to verify the offloaded computation's integrity and the reported results' correctness. Finally, VERISPLIT provides an effi-

cient and practical solution so that, unlike other heavyweight cryptographic solutions, offloading inferences with all security options enabled still outperforms the local inference baselines.

We perform security analysis with formal proofs to justify VERISPLIT's security and privacy guarantees. VERISPLIT consists of several algorithms, each addressing specific goals (data privacy, model confidentiality, and inference integrity). Therefore, we individually prove their correctness and security properties. Combining all together, VERISPLIT presents a comprehensive solution for secure and private inference offloading for IoT devices.

1.3 Thesis Outline

This thesis proceeds as follows. In Chapter 2, we discuss the background and related work in securing IoT devices, protecting users' privacy, and facilitating mobile device computation offloading. In Chapter 3, we discuss TEO, a new device ownership and access management architecture to help devices protect user data and provide access control. In Chapter 4, we present CAPTURE, which helps IoT devices manage third-party libraries to improve system security. In Chapter 5, we discuss VERISPLIT, a new offloading framework that facilitates secure and private machine learning inference offloading across local IoT devices. Finally, in Chapter 6, we discuss lessons learned and future research directions and conclude the thesis.

Chapter 2

Background

This chapter provides background information and an overview of related work in the broad Internet-of-Things (IoT) research space. We split this chapter into three parts, covering various discussion topics including 1) high-level challenges and user concerns in IoT security and privacy, 2) current and experimental system and networking designs of IoT devices, and 3) novel offloading solutions for emerging IoT applications (specifically, machine learning model inferences in IoT).

2.1 IoT Security and Privacy Concerns

2.1.1 Stakeholder and Bystander Privacy

Recent work has identified the challenges due to the discrepancy between decision makers and device users [41, 93, 235], partially motivating the need for stakeholder privacy. Some of this work examines privacy issues from the perspective of bystanders [22, 228] and incidental users [40]. Moreover, research on preventing intimate partner violence (IPV) for smart devices [75, 76, 92, 161, 210] also echoes our goals for stakeholders' privacy.

2.1.2 Smart Device Access Control

Motivated by real-world security incidents and research results that highlight the risks stemming from mismanaged IoT delegation chains [230], researchers have proposed many improvements and novel access control systems for smart devices, as surveyed by He et al. [94]. Prior work have proposed improving access control policy language with expressive authorization logic [20] and contextual information about the home environment [71, 105, 187]. Specifically, Kratos [196] designs an access control system over multi-user multi-device-aware smart homes. Complementary to providing granular access control from a temporal aspect, I-Pic [1] proposes a novel approach to enforce privacy policies for pictures owned by groups, while Hivemind [114] introduces mechanisms to enable collectively control of shared public IoT actuators. Moreover, many decentralized authorization frameworks, including Wave [10] and several blockchain-based approaches [18, 61, 168, 191], further eliminate the need for central trusted entities. Finally, the use

of novel cryptographic constructions to facilitate access control and delegation has been present in several instances of recent work [101, 118, 123, 158, 192, 193].

2.2 New System Designs for IoT Security and Privacy

2.2.1 IoT Network Security

Several prior efforts have looked at IoT security issues [229], and proposed augmenting current network designs to address them. Dreamcatcher [65] uses a network attribution method to prevent link-layer spoofing attacks. Simpson et al. [198], DeadBolt [115], and SecWIR [125] propose adding features and components on network routers to secure unencrypted traffic. HoMonit [242], Bark [98], and HanGuard [50] propose finer-grained network filtering rules and context-rich firewall designs.

2.2.2 IoT Software Security

Several projects address vulnerabilities in various aspects of current IoT software development. Vigilia [209] introduces capability-based network access control to protect devices while supporting home automation applications. Each device has one *driver* program, which provides public APIs accessible by home automation programs. Other efforts [129, 151, 222] address security challenges in the application-layer of devices, such as operation logging, cloud backend services, and automation apps, which are complementary to our work.

2.2.3 IoT Frameworks and OSes

Both academia and industry have looked at the challenges of IoT software stacks for smart homes with heterogeneous IoT devices. HomeOS [56] proposes a unified PC-like platform to manage all local devices. Commercial IoT frameworks emphasize their security offerings and ease of management for third-party developers. Microsoft Azure Sphere [140], Particle OS [165], and AWS Greengrass [8] all provide services to manage device library updates on behalf of developers. These frameworks also include native support for application-level over-the-air upgrades, reducing the barrier for developers to patch bugs. Samsung SmartThings Device SDK [184] reduces the developer burden of managing library updates by directly offering high-level APIs in the SDK (e.g., MQTT services). Developers do not need to worry about patching libraries, as long as they regularly update the SDK runtime.

2.3 Secure Offloading Designs for Emerging IoT Applications

2.3.1 Trusted Hardware

To facilitate secure and verifiable computation offload, several prior works leverage hardware support using trusted execution environments (TEEs). Slalom [208] utilizes a CPU TEE on the worker device (e.g., Intel secure enclaves) to ensure the correctness of ML inferences on GPUs.

Similarly, DeepAttest [34] relies on TEEs to attest the integrity of DNN models. TrustFL [244] and PPFL [143] also leverage TEEs (on untrusted devices) to ensure the integrity of federated learning based training.

Several recent efforts propose new secure accelerators [100, 215, 249] to expand the trusted computing base beyond the CPU. Notably, Graviton [215] brings trusted execution environments to GPUs, enabling new applications such as privacy-preserving, high-performance video analysis [169]. Efforts like SecureTF [171] extend ML frameworks with TEE support to help with developer adoption. Commercial efforts [154] have started adding support for TEEs on datacenter GPUs.

With a trusted computing base on workers, it should be possible to meet our goals by running model inferences inside TEEs. However, several practical challenges and additional security implications hinder its adoption for IoT devices. First, executing inferences with TEEs inside CPUs incurs high performance overhead [208]. In addition, TEE-enabled accelerators are too costly to be widely deployed in users' smart homes (e.g., datacenter GPUs cost over $10,000 [38]). Moreover, TEE systems may be vulnerable to side-channel attacks [33, 172, 212] and other attack vectors [2], potentially affecting our security goals and requiring additional research and security enhancement efforts.

2.3.2 Efficient Verification

Various techniques for general-purpose verifiable computation [77] and proof techniques for practical verification[45, 164] have been used to support verifiable inferences using ML models. For example, SafetyNets [78] proposes interactive proof (IP) protocols and VeriML [245] leverages succinct non-interactive arguments of knowledge (SNARK) proofs to make the inference execution verifiable. Unfortunately, these proof generation techniques require computing over finite fields (instead of real numbers) and do not support non-linear activation functions (and hence must use quadratic activation approximation). Therefore, these approaches do not scale to large neural networks while preserving high accuracy of the original, unmodified models.

In addition to generating succinct proofs, prior works have explored other techniques for efficient verification. Fiore et al. [72] propose the approach of "hash-first, verify-later" to verify computations on outsourced data. Zhang and Muhr [243] introduce a selective testing mechanism for federated learning that also use Merkle tree hashes to record intermediate results. GOAT [17] performs probabilistic verification of the intermediate results of machine learning inferences.

2.3.3 Cryptography for Machine Learning Applications

Many recent works propose novel cryptographic protocols for secure ML inferences. These solutions ensure the integrity of the computation since unfaithful executions can be easily detected by the client (e.g., resulting in corrupted ciphertext). One popular approach is by using partial or fully homomorphic encryption (FHE) for matrix multiplication used for inferences [13, 79, 106, 227]. In addition to providing inference integrity, homomorphic encryption often preserves data privacy from untrusted workers because it enables computation over encrypted data. Despite recent advances in making FHE more efficient [178] and using accel-

erators [183], homomorphic encryption still imposes a significant performance penalty, slowing down inferences by orders of magnitudes.

To alleviate FHE's overhead, recent works have explored incorporating secure multi-party computation into the offloading protocol design [144] or with homomorphic encryption [110, 142]. In addition, systems like MiniONN [133] leverage oblivious transfers to prevent private data from leaving the device while preserving model secrets on the worker machine.

2.4 Summary

Although many researchers have looked into various challenges in IoT devices, a few important problems remain unanswered. The rest of this thesis will present our contributions in these areas: protecting stakeholder privacy, enhancing device security from mismanaged library updates, and improving computation efficiency through local offloading. More importantly, this thesis proposes novel solutions with practicality and strong security guarantees.

Chapter 3

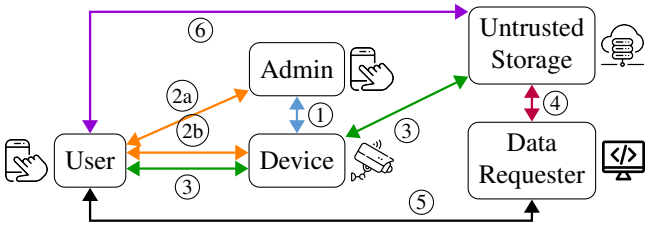
TEO: Protecting IoT Device Users by Offloading Ownership Management and Access Control

Many IoT device users worry about the potential privacy implications of the ubiquitous IoT device deployment and the breadth of the sensitive data they can collect [22, 64, 93]. Although IoT device vendors already have basic access controls in their existing applications, those systems are insufficient to handle complex use cases in future IoT deployments. Specifically, many prior works have looked into how to improve IoT device authorization and delegation systems' expressiveness [20, 71, 105, 187], decentralized storage models [18, 61, 168, 191], and cryptographically enforced access policies [101, 118, 123, 158, 192, 193, 217]. One common pitfall of these approaches is that they primarily focus on addressing the privacy concerns of the device owners and fail to consider concerns from other stakeholders. Those whose data may be captured by the IoT device must trust the device owners or the small group of device administrators.

In this thesis chapter, we describe our work on the design of a new IoT device ownership model (TEO — IoT Ephemeral Ownership) giving direct controls to the device stakeholders — who may be impacted by the presence of the device — and the implementation of a new system architecture demonstrating the ownership model. This architecture splits the role of maintaining secure data storage and managing access controls of users' private data from the list of responsibilities of current IoT device vendors into a third-party service. More importantly, we limit the trusted computing base of handling private data and avoid any trusted requirements for the cloud and storage service providers. At the same time, the data owners retain control and ultimately decide who can access their data. To provide strong security guarantees, we apply formal security analysis and verify that the proposed protocol design satisfies all our security goals.

The rest of this section describes TEO in detail, based on our published paper [240]. Moreover, we presented a working prototype of TEO as a demonstration [241] and released the source code of our prototype [238]. In Appendix A, we provide the formal modeling code of TEO's protocol used for verification in ProVerif and the additional template language we developed.

Figure 3.1: Overall TEO workflow. An admin initializes the device (①). Next, the user claim device ownership with the admin's pre-approval (②a and ②b). During normal operation (③), the device encrypts users' data and uploads it to storage. A requester can download the data (④), but needs the owner's approval to decrypt it (⑤). To revoke access, the user can directly issue a request to the storage provider (⑥).



3.1 Motivation: Importance of Stakeholder Privacy

Internet-of-Things (IoT) devices are rapidly gaining popularity in both private settings (e.g., homes, offices) and public spaces (e.g., conference rooms) [30, 74]. However, their growing ubiquity has led to privacy concerns that often stem from the breadth of sensitive data (e.g. audio, video, images) that they can sense [22, 64, 93].

To address growing IoT privacy concerns, prior work proposes expressive access control mechanisms for authorization and delegation [20, 71, 105, 187], decentralized storage solutions based on blockchain [18, 61, 168, 191], and cryptographic access control schemes [10, 101, 118, 123, 158, 192, 193]. One key limitation of these approaches is that they lack support for exclusive user control. They focus primarily on the privacy concerns relevant to the device owners and not other stakeholders who may have legitimate concerns about these devices since the data they capture may relate to their activities. These approaches often grant the sole device owner, or a small group of administrators, full authority over the data that the device generates, assuming these entities are trusted by those who may be impacted by the devices.

As pointed out by a growing body of research, these assumptions are not always consistent with the emerging ubiquity of smart devices in shared spaces. Passive bystanders may nonetheless be impacted by a smart device simply by, for example, visiting someone's home [22, 228], or by incidental exposure in public areas [40]. A related but distinct scenario is illustrated by the presence of IoT devices in short-term rentals (e.g., Airbnb), where guests may wish to use whatever smart devices are in their units but have concerns about the host's ability to invade their privacy [53, 135]. Both cases highlight the lack of support for *stakeholders' privacy* in existing access management systems. In situations like these, where there is an implied expectation of privacy, any user in a device's vicinity who is impacted by its sensors should have a say when it comes to the device's functionality and the data it generates. A key challenge arises as to how to give stakeholders control over devices that impact them so that they can decide how these devices operate, and who has access to any data that emanates from their sensors.

3.2 System Overview

To address stakeholders' privacy and security concerns, we envision a new model of device ownership that protects the interests of both the device users and its administrators. We propose TEO — IoT Ephemeral Ownership – that grant users, as ephemeral owners, full control over the device's operations. Historical data collected by the device will always belong to the ephemeral owners. When someone wants to access the data, they have to get permission from all the data owners (stakeholders). While the device administrators decide who can claim the device to become an ephemeral owner, they cannot interfere with the device's operation or access private data captured by it without the owners' approval. To help contextualize TEO's workflow, we use a running example of a group of friends renting an Airbnb house for the rest of this section.

The high-level TEO workflow is illustrated in Figure 3.1. The admin (e.g., Airbnb host) first installs the IoT device and initializes it. (Step ①). Afterward, the device is ready to be claimed by new owners only if they are authorized by the device admin. The potential users (e.g., Airbnb guests) all have a user agent program running on their phones. After booking their reservation, they need to ask the host to issue "pre-auth tokens" with everyone's public key (Step ②ⓐ). Pre-auth tokens prevent unauthorized people such as malicious neighbors from accessing the device. When users arrive at the Airbnb rental, the user agent on their phones initiates a process to claim the device (Step ②ⓑ). As the group membership changes (users join and leave), the device dynamically adjusts the set of owners. During normal operation, the device protects its stakeholders in two ways. First, if the device receives commands to perform actions (e.g., open the door, adjust the temperature), it needs to ensure the command is authorized by the current owners (omitted from the figure). Every command includes a certificate with user-generated signatures, which the device can verify using the current owners' public keys. Second, the device preserves users' data ownership with a series of encryption operations and distributes individual data keys to the set of owners (Step ③). To facilitate future data access and reduce users' storage overhead, TEO-enabled devices directly upload the encrypted data to any untrusted cloud storage provider. When any entity wants to request access to this encrypted data (Step ④), they need to seek the permissions of the original owner(s) to decrypt it (Step ⑤). Later on, if a user decides to revoke the access, they can directly contact the storage service provider and provide re-key tokens (Step ⑥) generated with key homomorphic encryption. This special cryptography primitive allows the storage provider to switch encryption keys of the ciphertext by directly applying user-provided tokens. In other words, key revocation can be performed by the storage provider on the encrypted data itself without having to decrypt it first.

3.2.1 Target Use Cases

TEO aims to provide ephemeral owners with full control over who can access their data and when regardless of data types. Therefore, TEO is well suited for applications and IoT devices that store operational data that could contain sensitive information about their users, such as camera and speaker recordings or sensor readings. For data accesses, we primarily focus on scenarios where data requesters want to use historical data for analysis, such as to train machine learning models or to recall past events.

Since TEO devices maintain an up-to-date list of current owners, we can extend TEO to

enforce real-time access control of the device as well. Consider, for example, a smart door lock. After being claimed by a new user, this lock should reject commands issued by previous owners. To achieve this, the device can require that all incoming commands include an authorization certificate signed by the owner, and we implemented a simple application as part of our case studies. Although not the current focus of TEO, we believe it would be a useful future direction to enrich certificate designs with proof-carrying authorizations [15, 20, 23, 119] for more expressive policy specification languages.

TEO targets a variety of deployment scenarios such as rental homes and shared offices. Specifically, they have different design requirements and considerations. In rental homes (e.g., Airbnb), the host sets up smart devices and lets guests use them. Guests have lower churn rates (stay a few days at least) and smaller group sizes. Sometimes, a single owner would suffice as the group implicitly trusts each other if they stay at the same place. In contrast, smart devices in shared offices and conference rooms have more frequent changes in owners and group members would prefer an equal role in decision making. These devices would be managed by the building managers, and they may have more insight on users' daily routine (e.g., which floors and rooms they are likely to occupy). Building managers can potentially use this information to improve the practicality of TEO. For example, they can selectively issue pre-auth tokens to conference rooms based on a user's calendar events.

3.2.2 Design Goals

Flexible Association of Devices and Users. We expect frequent ownership changes in physical spaces with smart devices. An Airbnb may see ownership changes that span days, while shared spaces in smart buildings (e.g. conference rooms) may see ownership changes even hourly. Moreover, multiple users can share an office and hence be collectively impacted by the devices. Ideally, all stakeholders should have a say in controlling the device and accessing the data it collects. Unfortunately, existing smart home access control systems often assume a static group of user(s) make all these decisions.

Preserving Data Ownership. Data collected by smart devices should always belong to the group of users present at the time of capture. Anyone trying to access the data should request the data owners' permission. Most importantly, dynamically changing ownership of the device's users and administrators should not affect historical data ownership. This requirement ensures that users preserve their control over the private data even if they no longer own the original device in the future.

Decentralized Trust. Users should be able to manage access requests without relying on third parties. Centralized access control systems, managed by individual companies and building owners, require complete trust in these entities and in their ability to protect users' data and enforce access policies. In return, centralized systems provide an efficient solution for processing access requests and sharing data. On the contrary, we want to empower users to decide who should have access to their data themselves while benefiting from the performance and availability of cloud services.

Formally Verified Security. Our goal is to provide formally verified security guarantees for our proposed TEO system. The main components of TEO are a series of complex communication protocols designed for multiple entities in IoT deployments. Therefore, we encode our protocol specifications into models and verify their security and correctness under our stated threat model in Section 3.4. After several rounds of refinement, our streamlined TEO design provides assurances of security and correctness.

3.2.3 Threat Model

We designed TEO under the assumption of a powerful attacker, who can monitor all communications between users, devices, and the third-party storage provider, attempts to undermine its goals by either *(1)* controlling the device without the active consent of the ephemeral owner, *(2)* accessing the data generated by the device, which should only be accessible by the ephemeral owner, or *(3)* impersonating one of these parties. This follows the Dolev-Yao network attacker model [57] used in our formal analysis (Section 3.4). Concretely, such an attacker might correspond to someone in the vicinity of the device, a malicious admin who wishes to violate the privacy of an ephemeral owner, or a previous ephemeral owner who aims to extend their control of the device and its data past the agreed-upon terms.

We assume that local devices are trusted to correctly execute the protocol, i.e. the TEO-enabled device has not been backdoored or rooted, and will not leak data, encryption keys, or bypass authorization checks. We assume that the third-party storage providers may be *passively* malicious (i.e. honest-but-curious): they might attempt to extract private information from the data they receive, but will faithfully execute the TEO protocol as specified. This is consistent with using reputable cloud services, with whom users may not trust storing cleartext data, but for whom the reputational risk stemming from actively-malicious behavior is too great.

Our formal analysis (Section 3.4) is in the symbolic model, so we must also assume that the cryptographic primitives used by the protocol are secure against computational attacks. Likewise, our analysis does not consider information that might be leaked from metadata (e.g., the identity of users who participate in the protocol, from their public key certificates), nor semantic information that might leak through traffic analysis (e.g., inferring user behaviors by observing the timing and sizes of encrypted network packets). While these may provide opportunities for attackers to learn unwanted information in certain settings, we leave careful consideration of these risks to future work.

3.3 TEO Protocol

We now describe the TEO protocol and the workflow between device administrators, users, storage providers, and devices. We start by introducing the key challenges that we sought to address in our design and describing our notation.

Granular Ownership & Data-Sharing. First, we envision TEO's use in settings where device ownership changes frequently, and the duration of ownership varies drastically (e.g., using a conference room for tens of minutes, or renting a house for several days). We accommodate these

in TEO by partitioning the ownership period into relatively small segments, with each segment tracking its set of owners (potentially distinct) from adjacent segments. A new user wishing to claim (potentially shared) ownership of a device takes effect at the start of the next available segment. This approach facilitates fine-grained data sharing of selected time windows of data rather than all-or-nothing sharing. However, to do so requires generating fresh keys even when ownership remains the same across segments. Configuring a relatively small segment interval affords flexible and responsive transitive ownership, at the cost of the corresponding overhead of managing cryptographic state for each segment, and additional rounds of TEO communication.

The storage overhead introduced by this scheme may be significant for mobile user agents. However, for many of the envisioned use scenarios for TEO, data sharing requests are likely to access several contiguous segments, e.g., in the case of streaming video or sensor readings. Thus, to mitigate the storage overhead, we designed TEO to securely store groups of keys from a single session on the untrusted cloud storage, so that the user agent is only responsible for maintaining a single key for the entire session. This mechanism also enables efficient group ownership, using Shamir Secret Sharing [194] to split data block keys across group members.

Efficient Revocation. Finally, efficient key revocation is challenging, as it is infeasible for resource-constrained user agents to download, locally re-encrypt, and re-upload ciphertexts to the storage. We leverage key-homomorphic encryption [28, 217] to facilitate re-encryption directly on the untrusted storage provider, requiring users to only generate fresh rekey-tokens.

3.3.1 Notation

We use three main encryption primitives in our protocol. Each is denoted by $\mathrm{Enc}(\cdot)$ for encryption and $\mathrm{Dec}(\cdot)$ for decryption, but vary by the set of keys that they take. Symmetric-key encryption is represented as $\mathrm{Enc}_k(n, m)$ with key k, nonce n, and message m. Long messages are concatenated with multiple parts $(m_1|\cdots|m_n)$. Encrypting a message with a recipient's public key is denoted by $\mathrm{Enc}_{pk_R}(m)$ where pk_R is the receiver's public key. We use public-key authenticated encryption to protect the message's confidentiality and integrity when the public keys of both parties are known, denoted by $\mathrm{Enc}_{\langle sk_S, pk_R\rangle}(n, m)$ where sk_S is the sender's secret key. Finally, we use key homomorphic encryption from Sieve [217] to support revocation, and denote this operation by $\mathrm{SieveEnc}_{K_s}(n_s, m)$, where K_s is the Sieve key.

3.3.2 Device Initialization

New devices need to go through a one-time initialization process and obtain a valid device proof (DP), as illustrated in Figure 3.2, before they can start regular operation and accept new owners. The device proof is necessary for the device to demonstrate its authenticity to potential owners in later steps. To facilitate mutual authentication, devices and administrators need to establish an out-of-band communication channel to share setup keys. This is common with smart devices using existing solutions including QR codes printed on devices, physically-printed passkeys packaged with the device, and short-range wireless communication [159]. Our prototype implementation uses QR codes to share setup keys in this phase of the protocol. If the device needs

Figure 3.2: Protocol workflow for device initialization.

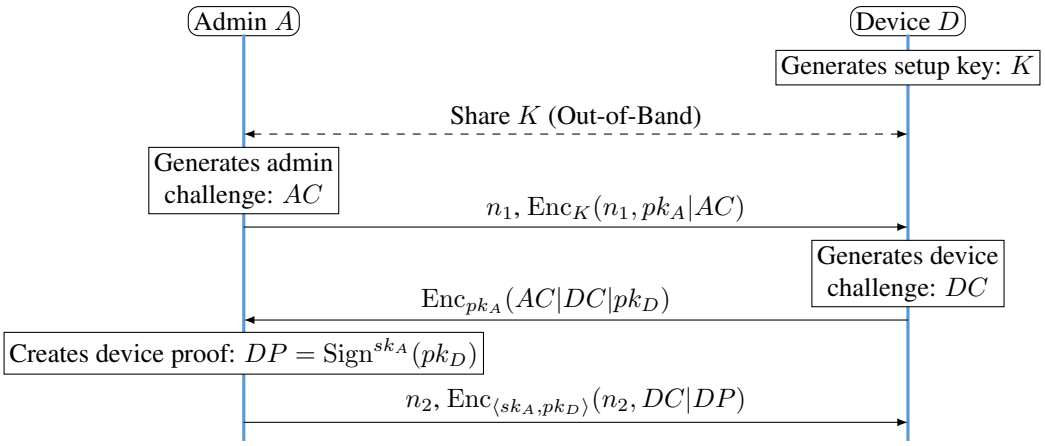
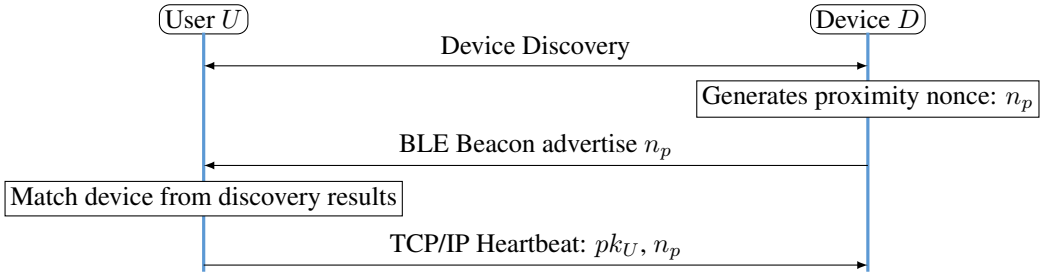


Figure 3.3: Protocol workflow for proximity detection.



to change its administrator, it must be reset, and the initialization process needs to be completed again with fresh out-of-band key material. Note that this is different from changing *ephemeral* owners, which we describe next.

3.3.3 Device Ownership Management

Device Discovery. When users enter a new environment, they can discover TEO-enabled devices to obtain public information about the device, such as its public key and admin information. This can be achieved in several ways. First, they can locate the device and scan the QR code dynamically generated and displayed on the device. However, many IoT devices do not have displays to provide such functionalities. Alternatively, the device can advertise its presence and communicate this information over a local network, similar to service discovery functions used in existing protocols [16, 155, 204, 236]. Users broadcast discovery requests to all local hosts and TEO devices will respond with their information. Since this request is broadcast, users may receive responses from devices located in nearby rooms and offices. To reduce false positives, we envision several potential extensions to improve TEO's usefulness and practicality in future deployments. First, the reply messages can include additional information about the device's location, so users can choose the correct device based on their own location. In addition, admins can decide who are capable of claiming certain devices by restricting pre-auth tokens to users. For example, Airbnb hosts can give tokens based on guests' emails, and building managers can

Figure 3.4: Protocol workflow for acquiring pre-auth tokens.

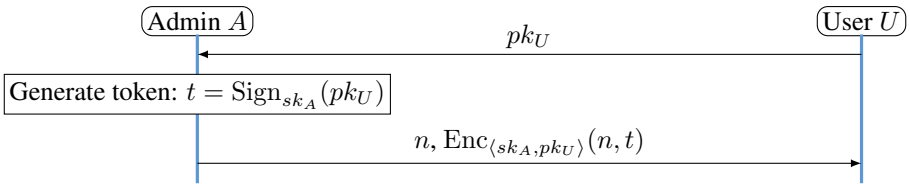
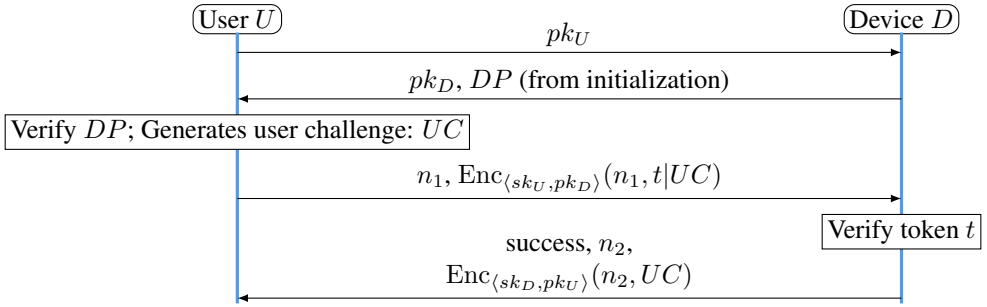


Figure 3.5: Protocol workflow for claiming ownership.



provision tokens based on users' locations (buildings, floors, etc.). Finally, the device can ask its current occupants to moderate new users' join requests since they can physically confirm whether the new users are in the room.

Proximity Detection and Duration Setting. We propose a Bluetooth Low Energy (BLE) based proximity detection mechanism for the TEO enabled device to detect when ephemeral owners leave its vicinity. For simpler scenarios such as home rentals, this might be unnecessary since the owners' stay has a fixed duration. However, this mechanism would alleviate challenges in shared spaces such as offices and conference rooms where users frequently enter and leave. Figure 3.3 shows the workflow of TEO's BLE proximity detection. The device periodically generates new proximity nonce n_p and broadcasts it through BLE. We do not require a high frequency of new proximity nonce generation, since the device only needs to know if the owner is part of the current data block recording (e.g. one data block per minute). Since multiple users can collectively own the same device, we set the device in BLE beacon mode so it continuously advertises information without requiring an explicit connection from the user's phone. On the other side, the TEO app on the user's phone periodically sends out heartbeat messages with the latest nonce sent within the device advertisements over BLE. Once the device stops receiving the correct nonce from a particular user's phone for an extended period, it can infer that the user must no longer be in BLE range and thus remove them from the list of ephemeral owners. The device should be configured with proper transmission power so only nearby users likely in the same space can receive the BLE advertisement while reducing false positives. This automated process is executed by the user's phone app to minimize user burden. However, as a fallback mechanism, users can manually specify the duration of their occupancy in a space (or use default values) to be included in the ownership claiming process.

This mechanism is designed to ensure current ephemeral owners are still within the device's

vicinity. Consequently, the device can quickly remove stale owners that have already left the room. On the other hand, this mechanism is not intended to limit who *can* claim devices (the responsibility of pre-auth tokens) because BLE could have high false positive rates (i.e., everyone within the range will receive this message).

Claiming Ownership. Figures 3.4 and 3.5 illustrate the steps to claim ephemeral ownership. First, the user acquires a "pre-authorization token" from the administrator (Figure 3.4). TEO uses these tokens to prevent unauthorized device access and enforce more granular, device-specific usage policies. For example, rental hosts can generate pre-auth tokens for guests with upcoming reservations, and building administrators can give pre-auth tokens for the devices in a specific office to those who can use them.

Figure 3.5 illustrates the next step in establishing ownership. Just as the pre-auth token establishes the authenticity of the ownership claim to the device, the device proves its authenticity to the new owner by issuing its device proof (DP) obtained during initialization. The user also generates a fresh random challenge (UC) to prevent replay-enabled device spoofing. Extending this phase to support group ownership is straightforward: each user performs the steps in Figure 3.5 independently, and the device tracks the list of ephemeral owners accordingly. The device is then responsible for synchronizing control between its current owners.

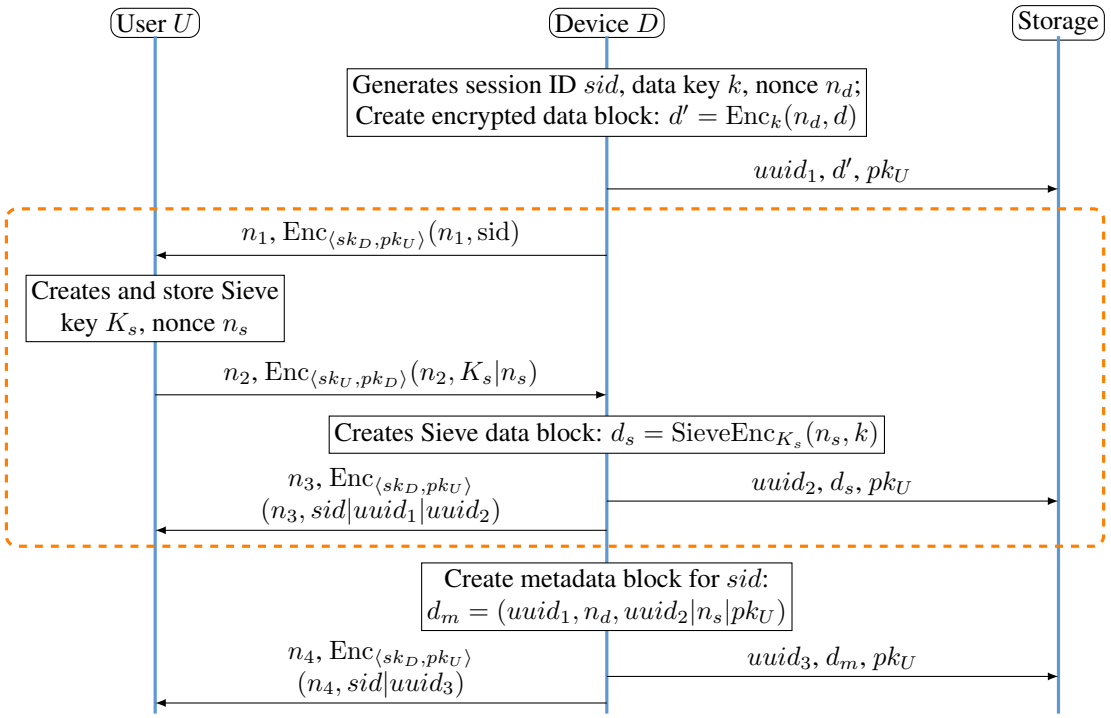
The device needs to verify the integrity of the token t before accepting the owner. Optionally, this token can include expiration times and other constraints (signed by the admin when creating the token), and the device can enforce these constraints during token verification. We can enhance the token design with JSON web tokens (JWT) [109] for this use case, although our current implementation does not support it yet.

After a user becomes an ephemeral owner, the device can enforce access checks for incoming commands and instructions from a cloud back-end or home automation platform (e.g., IFTTT) to prevent previous owners from retaining controls. To support this check, every automation applet or cloud service should obtain authorization in the form of signed certificates from current owners.

3.3.4 Data Storage and Access

To preserve users' ownership of data generated by the device, one approach would be to transmit any such data directly to the user, and let them manage it independently. This is too demanding for mobile user agents, and even if it were not, would impose an unnecessary burden on them. Our protocol instead uses a third-party cloud storage provider that is honest but curious: we expect it to correctly store data from the device and respond faithfully to users' requests, but do not trust it to refrain from attempting to inspect the confidential data. To protect the confidentiality of the device data, which may contain sensor readings or video recordings that users consider private, the device could encrypt the data before sending it to the cloud provider, and provide the user with all of the keys necessary to access it in the future. This poses several challenges, including how to support group ownership while managing data-sharing and subsequent revocation requests.

Figure 3.6: Example data storage workflow. Nonces with numerical subscripts are local variables, only used within the corresponding protocol flow. The orange box indicates user-specific actions in group ownership.



Sieve Encryption. In addition to standard cryptography operations, we incorporate a key homomorphic cipher proposed in Sieve [217] in TEO's protocol design. Key homomorphism [28] allows an entity to change the encryption key of a ciphertext without seeing the underlying plaintext. This characteristic is well-suited for untrusted TEO storage providers to assist in the process of revoking data access without ever being able to decrypt the data itself.

Storing Data. Here we explain the data storage process for a single owner, as we will discuss group modes later. Figure 3.6 illustrates how TEO addresses these challenges and the steps taken to store a user's data in the cloud. First, the device encrypts the data with a freshly generated session key k for the current time segment. We use symmetric encryption for computational efficiency. The device uploads the encrypted data block to the storage provider, with an identifier $uuid_1$. Next, the device obtains Sieve credentials from current owners (Sieve keys K_s and Sieve nonce n_s). Afterwards, the device constructs the Sieve data block for this session and encrypts the value of session key k with Sieve cipher. This Sieve data block is then uploaded to the storage provider with an identifier $uuid_2$. Finally, the device uploads a metadata block to the storage provider, containing all the information needed to locate the encrypted data, as well as the Sieve blocks and the nonces used for the Sieve cipher. Meanwhile, the owner can collect bookkeeping information (such as session IDs and block UUIDs) from the device asynchronously.

To grant access to an encrypted block, the user can distribute the Sieve key to the requester. The symmetric key used to decrypt the data is stored in the Sieve block on the storage provider.

Figure 3.7: Example data access workflow. The requester wants to access the data associated with $uuid_3$ from the previous case. For brevity, the steps involved with sending download requests for UUID to the storage are omitted.

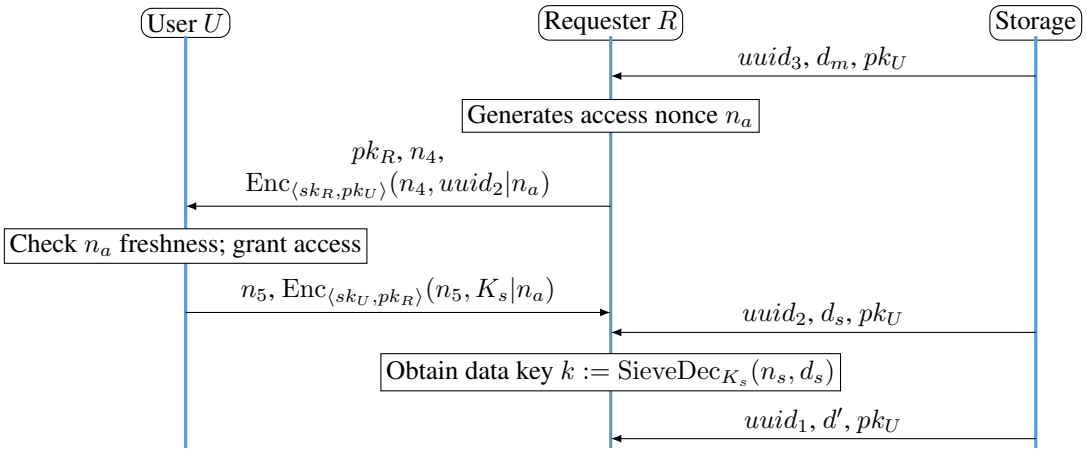


Figure 3.7 shows the process for someone to request data access. After receiving the Sieve key, the requester can gather all the information necessary to decrypt the requested data.

Threshold Encryption. One building block to enable group ownership is the well-established threshold encryption; specifically, we use Shamir Secret Sharing [194]. At a high level, the t-of-n threshold encryption allows protecting a secret message with n key shares. To decrypt the message, someone only needs to collect t shares ($t \leq n$). The values of t and n must be set statically before the encryption process begins.

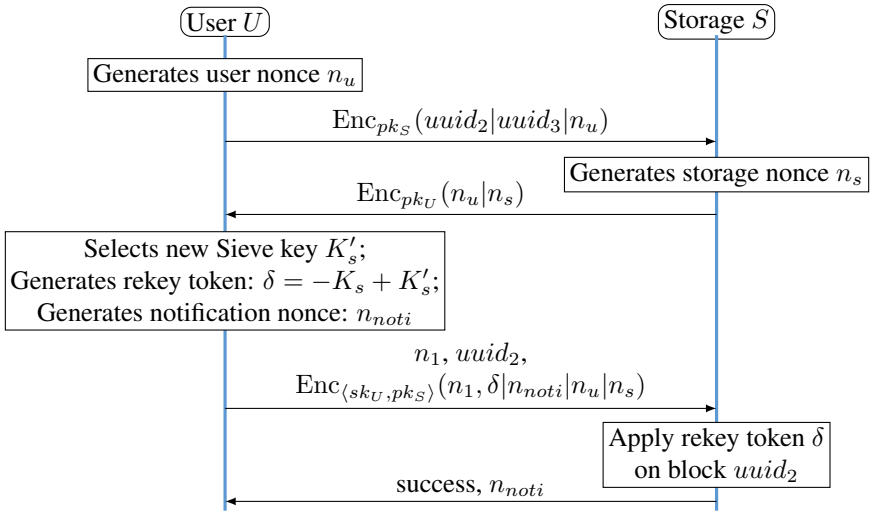
Group Ownership. To extend the protocol to support group ownership, the device needs to collect Sieve key information from each owner in the group by repeating the steps in the orange box of Figure 3.6. Assuming a group of owners with public keys $pk_G = [pk_{U_1}, \cdots, pk_{U_N}]$, the device still encrypts the data with the session key k the same way as before, and then uploads the result to the storage provider as $(uuid_1, d', pk_G)$. The device then splits the data key k into N key shares $k_1, \ldots, k_N$. It constructs one Sieve data block for each owner, $d_{s_i} = \text{SieveEnc}^{K_{s_i}}(n_{s_i}, k_i)$, and sends them to the storage provider as $(uuid_{2_i}, d_{s_i}, pk_{U_i})$. Finally, the device constructs a metadata block $uuid_3$ that refers to each of the owners and their corresponding Sieve blocks, and stores it on the storage provider:

$$d_m = (uuid_1, n_d, uuid_{2_1} | d_{s_1} | pk_{U_1} | \cdots | uuid_{2_N} | d_{s_N} | pk_{U_N})$$

To access a shared data block, the requester needs to seek permission of each owner and obtain the Sieve key for their share of the data key.

We conclude by noting that threshold encryption can support several data access policies by adjusting the threshold value t. Currently, TEO requires that accessors have the approval of all group members (by setting $t == n$) because we want to give everyone the right to veto. It is straightforward to extend TEO with alternative policies (e.g., requiring majority approval

Figure 3.8: Revocation workflow.



by choosing $t > n/2$). In addition, a future extension of TEO can include another layer of threshold encryption for individual users. Each user can save multiple key shares on different agents (laptops, phones, backup codes) and have a threshold value $t == 1$ in case they lose devices.

3.3.5 Revocation

TEO uses three blocks to encrypt each segment of data uploaded by the device and to support efficient revocation of access to a given block — the encrypted payload, a Sieve block, and the metadata block in plaintext. If the revocation was not needed, then the design could be simplified and storage overhead mitigated by dropping the Sieve block, and granting access by sharing the data key directly. Instead, our approach manages access by treating the Sieve key as a credential so that revocation can be accomplished by having the storage provider re-encrypt only the Sieve block, which is small in size relative to the actual ciphertext.

Figure 3.8 illustrates revocation in TEO. The user generates a rekey-token δ, and sends it to the storage provider along with identifiers for the appropriate Sieve blocks. The storage provider re-encrypts these blocks using the rekey-token. A benefit of this design is the consolidation of multiple encrypted data blocks, each with its own key, into a single Sieve block while maintaining low overhead on the client's side for revocation. The client can generate a fixed size rekey token to change the Sieve data block, thereby avoiding the need to download and re-encrypt arbitrary sized Sieve data blocks containing multiple data keys. Unifying encrypted data blocks in this way is especially helpful when a session contains a series of smaller data chunks, for example, an hour's worth of video recording may be stored as one-minute chunks to accommodate frequent membership changes and granular sharing, but the user is not burdened with managing credentials for each of these chunks individually. Additionally, each owner in a group can make access control decisions independently by rekeying their corresponding Sieve block.

3.3.6 Partial Availability

To process access requests, data owners (users in Figure 3.7) need to be online. This requirement in TEO is intentional to give users direct control over their data, as all access requests must seek their direct approval. However, the limitation is that even if a single user is unreachable, no one can access the original content even if they have everyone else's permission already. To strike a balance between data availability and users' access control, the group of users can modify the access policies at the time of data recording to choose different values of t for the threshold encryption (as discussed in Section 3.3.4).

Moreover, individual users may lose their mobile devices and thus lose their key shares. Aside from periodic backups, one popular solution for implementing recovery mechanisms is to leverage threshold encryption (as mentioned in Section 3.3.4 and demonstrated by prior work [217]). We acknowledge that such a mechanism is important for future TEO deployment in the real world.

Finally, users may be temporarily unreachable when the device executes the data storage operation. Our protocol design ensures that unresponsive users will not block the main data encryption and upload functions (first step in Figure 3.6). If the device cannot reach a user, it can store the user's key shares locally and retry later. Eventually, the device deletes its local copy once the user is back online. Meanwhile, since the private data have already been uploaded, the device does not need to keep the data while waiting for the unresponsive users.

3.4 Security Analysis

We formally model TEO using a well-known protocol verifier, ProVerif [26], and encode several key security properties to verify TEO's security and correctness. In modeling TEO we address several challenges, particularly in formalizing group ownership, key splitting, and revocation.

3.4.1 Security Goals

We aim to achieve the following security goals with our TEO protocol design.

- **Secrecy.** A user's private data, once encrypted by a TEO supported device, should not be accessible by anyone without the explicit authorization of the user. For group ownership, the policy requires that only entities with the consent of all owners are able to access the data.
- **Mutual Authentication.** After the device is initialized and claimed (Section 3.3.3), all parties must mutually authenticate and agree on each other's roles (i.e., device and admin, device and owner must acknowledge each other).
- **Prevent Data Spoofing.** Attackers should not be able to spoof data, potentially overwhelming users' local storage space with keys for non-existent data blocks. If the user concludes a data store operation, then the device must have indeed stored the user's private data for the corresponding session.
- **Effectiveness of Revocation.** If the data requester's access is revoked by the owner, the requester should not be able to decrypt the data block if they download it again from the

storage provider. Conversely, revocation should only happen when the owner requests it, and the new key should be able to decrypt the data in the future. For groups, an individual's decision should not affect others (i.e., others' keys should still work since they did not revoke their keys). Note that TEO does not preclude a requester from storing the already decrypted data offline perpetually.

3.4.2 Modeling Protocol Workflow

We aim to identify protocol design-level bugs that may compromise TEO's security and privacy protections. We assume that the cryptographic primitives (e.g., encryption algorithms) are secure. Hence, we choose a *symbolic* verifier (ProVerif [26]) since it requires lower human guidance and is better suited for automated analysis compared to *computational* ones. Interested readers can refer to prior work for a more detailed discussion of different types of protocol verifiers [19, 25].

In ProVerif, protocols are modeled as sets of *processes*. Each process can generate fresh internal variables and local secrets, such as private keys that are hidden from the attacker. We represent each entity (e.g., admin, user, device) as its own process that can spawn and execute repeatedly to conduct multiple rounds of communication.

We symbolically encode all cryptographic operations so that incorrect credentials (e.g., decryption keys and invalid signatures) will terminate the process' execution. Processes communicate over *channels*. Attackers can intercept, drop, or fabricate any message over the channel. Since attackers can obtain a complete history of all network messages, we can conservatively model the untrusted storage provider with this general network attacker.

3.4.3 Modeling Security Goals

We encode every security goal from Section 3.4.1 with concrete ProVerif queries to ensure TEO's protocol design satisfies all security properties. We create unique *events* as checkpoints for the execution of each process. We construct *correspondence* queries to encode properties such as "*if A happens, B must have already happened*". Correspondence queries can be *injective*, which means that the verifier will check that there is a strong one-to-one mapping between events. To ensure correctness, we also add *reachability* queries to verify that all events are reachable during process execution; unreachable events can vacuously satisfy correspondence queries, leading to a false conclusion of TEO's correctness.

Secrecy. We create a private variable userPrivateData to represent the confidential information. This variable can be shared across different processes but remains hidden from the network attacker. The device encrypts this variable and uploads the ciphertext to the storage. We construct a secrecy query to verify that this variable remains secret from the network attacker and untrusted storage provider.

To verify that a data requester (requesterPK) can only decrypt the data with approval from all owners (owner_1, owner_2, $\cdots$), we construct a query ensuring that the event AccessData(requesterPK, userPrivateData) from the requester is preceded by the GrantAccess(owner_i_PK, requesterPK, dataUUID) events for all data owners $i \in [1, N]$.

Mutual Authentication. We encode several injective correspondence queries to verify this goal. For initialization, the query states that whenever the event DeviceAcceptAdmin(devicePK, adminPK) happens on the device, the predecessor AdminAcquireDevice(adminPK, devicePK) must already have occurred for the admin.

For claiming device ownership, the device needs to verify that the user has a valid pre-auth token before accepting new owners. Therefore, the final event UserFinishDevice(userPK, devicePK, adminPK) on the user process should be preceded by the device-generated DeviceAcceptUser(userPK, devicePK, adminPK, preAuthToken), which itself should be preceded by the event that admin marks this token as valid, AdminGrant(adminPK, userPK, preAuthToken).

Prevent Data Spoofing. When the user finishes the Sieve key negotiation in data store (Figure 3.6), it produces the event UserStoreFinish(userPK, devicePK, sessionID). This event should be preceded, injectively, by the device side event DeviceFinishSieve(userPK, devicePK, sessionID) issued after it finishes uploading Sieve data block to storage. Violating this query will cause users to store information for non-existed sessions.

Effectiveness of Revocation. We encode the revocation process in different *phases*, a cross-process synchronization primitive provided by ProVerif. Operations in one phase will be inactive when the model moves into a new phase. All processes start in "phase 0". The data requester obtains the owner's authorization and successfully access the data. When the user revokes access, the system transfers into "phase 1" and Sieve data blocks in the storage are updated with new keys. In "phase 2", the requester downloads the data blocks from storage but attempts to decrypt with the previously cached Sieve key. We verify that SucceedDecryptOldKey(dataBlockUUID) should be unreachable. As group size increases, we add more phases and pick one owner to revoke access at each phase. By the end of every round, the requester's cache of other owners' Sieve keys is still valid, thus we also verify that one owner's decision to revoke will not interfere with other users' keys.

Finally, we implement an injective query to ensure that the storage provider only applies to rekey tokens upon the data owner's request. This query led us to identify a bug in an earlier protocol draft that an attacker can replay rekey requests, rendering users' data inaccessible by anyone. This finding prompted us to add additional nonces to our protocol.

3.4.4 Modeling Group Ownership

To support group ownership, we use Shamir Secret Sharing to distribute data keys among co-owners. However, ProVerif currently lacks language support for this type of threshold encryption [145], particularly for encoding variable-sized sets of co-owners. To address this, we encode the size of the owner set statically in the model. All parameters of cryptographic operations must also be set statically, including the number of users and their positions. We have to create unique processes for every user in the group to handle different keys and internal states. As the group size grows, we have to expand these arguments and processes accordingly. Note that the implementation of different user processes is nearly identical, except for minor differences in user

indices. Therefore, we developed a preprocessor language that automates the construction of static models with specified group sizes. We express the protocol flow with template functions and parameterized values. In this way, we can implement a common user process and, during compilation, expand this template into a variable number of concrete user processes.

3.5 Implementation

We provide an open-source repository [238] that includes the source code of TEO and the details of the security protocol modeling. We implement the core TEO protocol as a shared cross-platform library, `libteo`, with public-facing APIs. The library is written in 8945 lines of C++, excluding third-party libraries and evaluation tests. We use libsodium [130] as the main cryptography library, containing implementation for standard secret-key and public-key cryptography operations, with X25519 key exchange, XSalsa20 stream cipher encryption, Poly1306 MAC authentication, and Ed25519 signatures. In addition, we leverage the Crypto++ [46] library to implement Shamir Secret Sharing and the key splitting functionality for group ownership. Since the authors of Sieve [217] did not release their code, we re-implement Sieve operations using the Ed448-Goldilocks elliptic curve library [90] and consulted with them over email with our implementation details to ensure correctness. We use FlatBuffers [82] to serialize TEO's protocol message in a cross-platform format.

Our TEO prototype consists of client applications for multiple platforms. We developed a prototype Android app with support for the users' and admins' functionalities in 3350 lines of Java code. We use Android Beacon Library [150] for BLE scanning. It includes `libteo` as a native C++ library and uses Java Native Interface to execute API calls. We also implement test clients for different roles on x86 Linux desktops and popular single-board computers with ARM SoCs (Rasberry Pi 4 and Pi Zero W). Moreover, we develop a storage provider daemon as a key-value store for encrypted data contents using LevelDB [83] and with support for TEO revocation. In total, we implement these agents in 1206 lines of C++ in addition to the `libteo` library. TEO's protocol model contains 940 lines of ProVerif code with the group templates. After compilation, these models include 917, 1258, and 1599 lines of ProVerif code for group of size 1–3. We observed an exponential growth in verification time and memory consumption with larger group sizes. For example, on a 16-core machine with 64 GB RAM group size=1 took 3.62s, 258 MB memory while group size=3 took 17+ hours and consumed 50 GB. While we did not verify higher group sizes, our model generalizes to any group size.

3.6 Evaluation

We evaluate TEO's design and our prototype implementation, with a suite of microbenchmarks and by integrating TEO with several real-world IoT device applications. Our evaluation results demonstrate that TEO introduces nominal communication and power consumption overhead over a baseline system without TEO's security primitives. One-time operations, such as device initiation and ownership claims, add an additional latency of up to 187 ms. Meanwhile, devices that continuously upload TEO encrypted data experience a performance overhead mostly dominated

Table 3.1: Average latency (in ms) for TEO operations, with a performance comparison of different IoT device hardware. We also measure battery usage for the TEO phone app (in μAh). Data access and revocation operations do not involve devices' participation.

Operation	User App Battery (μAh)	Average Latency $\pm$ Standard Deviation (ms)	
		RPi 4	RPi Zero
Initialize Device	20.18	44 ± 9	65 ± 35
Acquire Pre-Auth Token + Claim Device	34.43	187 ± 52	258 ± 128
Claim Device	21.19	67 ± 10	94 ± 31
Store Data, 1MB	43.03	308 ± 57	684 ± 155
Access Data, 1MB	22.25	170 ± 54	
Revocation and Re-encrypt	25.07	62 ± 15	

by the network communication speed: for larger files, TEO incurs 7–25% extra latency compared to a baseline of just uploading the same size data; for smaller files (10KB – 1MB), TEO's storage latency is 101-308 ms.

To characterize the overhead of TEO's primitives, we select representative IoT devices and client platforms, with different computational capabilities. We developed a TEO mobile app and installed it on Android phones (Nexus 5X), serving as TEO client agents for users and admins. For operations requiring human interaction (e.g., deciding whether to grant access or issue pre-auth tokens), we skip the user confirmation step to automate the tests so as to only measure the overhead of TEO's protocol communication and not the user reaction time. We chose off-the-shelf single-board computers, namely Raspberry Pi 4 (1.5 GHz 4-core, 4GB RAM,$35–$55) and Raspberry Pi Zero W (1 GHz single-core, 512MB RAM, $10), as TEO-endabled IoT devices. Both Android phones and IoT devices connect to our campus WiFi infrastructure as other devices in the building. On the other hand, we launched our prototype storage providers and agents for requesting data accesses on Linux machines (8-core, 16GB RAM) with wired connections to the same infrastructure.

3.6.1 Microbenchmarks

Latency. We first measure the overhead of each TEO operation in terms of the end-to-end latency from initiating the operation to the time it completes (Table 3.1). We repeat every operation 100 times and report the mean and standard deviation latency. Several operations such as device initialization ($\sim$44ms) and revocation ($\sim$62ms) are lightweight, while the initial claiming of the device has higher latency ($\sim$187ms). We analyze recurring TEO operations (data store) in further detail. Switching the device from Raspberry Pi 4 to Pi Zero, we observe modest slowdown (up to 2.2x for data store) but all operations finish within 65–258 ms. This is understandable since they have drastically different compute capabilities. Overall, most TEO operations are only needed once or very infrequently, so a $\leq$187 ms latency increase has a modest impact on end users.

Phone Battery Impact. Table 3.1 reports the battery consumption of TEO operations involving the user's phone app, as the average μAh over 100 iterations. We currently use the battery levels

Table 3.2: Data store operation overhead breakdown for Raspberry Pi 4, reported as mean values in *ms*.

Data Size	Data Encryption	Data Upload	Total Time (vs. Upload Time)
10KB	< 1	19	101 (5x)
100KB	2	29	116 (4x)
1MB	25	127	308 (2.42x)
10MB	168	1429	1791 (1.25x)
100MB	1577	15256	16293 (1.07x)

reported by Android since they seemed sufficient for our use case [11], leaving more precise energy measurements for future work [126, 127]. Our test Nexus 5x has a rated battery capacity of 2700 mAh. Operations such as providing Sieve key share for the device to store data (43.03 μAh) once a minute consumes just 2.2% of the battery life over a 24 hour period. Proximity detection (Section 3.3.3) for group membership supported by BLE scans also affects battery life depending on the frequency. We measure the battery drain speed over a period of 5 minutes and calculate the difference when the phone is idle. Continuous scanning quickly drains the battery at a speed of 2090 μAh per minute. However, a simple optimization (increasing the BLE scan interval to once every 10 seconds) reduces the drain to 66.2 μAh per minute, consuming 3.5% of the battery over a 24 hour period. With additional optimizations (iBeacon-based BLE entry/exit detection, reducing scan intervals), the energy impact of proximity detection can be reduced further.

Data Store and Sizes. The latency overhead of recurring operations, such as encrypting the data on the device and then transmitting it to be stored on a storage provider, depends on the size of the data, as reported in Table 3.2. We omit other overheads in the breakdown table since they do not scale significantly with data sizes. For example, Sieve data blocks (containing the data keys) are independent of the size of the data in this experiment and have the same size since we use a single key to encrypt the data.

As the data size increases, the overheads associated with data encryption and upload scale proportionally, dominating the latency for using TEO for large data sizes. For example, the total latency is just 1.25x compared to the time spent on uploading the 10MB files (since TEO's symmetric encryption produces ciphertext the same length as the plaintext). For larger files (100MB), the relative latency of TEO decreases further to 1.07x upload time, showing that TEO's overhead amortizes as the data size increases. For smaller data sizes (e.g. 10KB - 1MB), TEO's protocol overhead is still relatively small ($\leq$ 308 ms) given the asynchronous nature of data storage operations.

To help reduce TEO overhead for large files, we implemented an optimization to *pipeline* data encryption and upload. We split large data into fixed chunks (1MB by default) and start uploading them as soon as the encryption of that chunk completes. Therefore, for large files such as 100MB, the sum of encryption and upload time exceeds the total elapsed time.

Table 3.3: Average latency and standard deviation for storing 1MB data for different group sizes. We emulate multiple owners as different processes on a PC and have the device repeatedly store data 100 times. We also include a single user running a TEO phone agent.

Group Size	Phone	Emulation			
	1	1	5	25	50
Average Latency (ms)	308 ±57	234 ±156	316 ±30	634 ±58	1085 ±85

Table 3.4: Total changes required (lines of code) to integrate existing applications with TEO. These changes mostly focus on redirecting data storage to the co-located TEO device driver program. See more details in Section 3.6.2.

Applications	Motion [146]	Mycroft [147]	Doorlock [89]
Language	C++/Python	Python	Node.js
Lines Changed	31	73	121

Group Ownership. We measure the performance impact of variable group sizes on the device's data store operations. We emulate a large number of users in a group using a standalone Linux desktop with a wired connection to the campus network. Each user in this scenario is a separate process on this desktop. The device still performs normal TEO operations, but it needs to communicate with all users. Table 3.3 shows the average latency for such operations for groups of up to 50 members. To provide a comparison between our emulated users forming a group and a real phone client, we also include the latency for a single user on an Android phone (first column). We observe a small performance discrepancy between the emulation platform (Linux) and the real Android phones (308 ms vs. 234 ms). As the size of the group increases, the main cause of the additional latency is the resource contention on the device. For every owner, the IoT device spins off a new thread to encrypt their key share with Sieve and upload the Sieve data block to the storage. Because the Raspberry Pi 4 only has 4 cores, threads for different owners cause CPU contention. Even with a large group size of 50 users, the slowdown is just 5x compared to the single-owner case.

3.6.2 Case Studies

We integrate three real-world smart IoT applications into TEO-enabled devices (Raspberry Pi 4). We searched for popular open-source smart apps on GitHub and tutorial websites and finalized one for each of the interesting categories. In all three cases, we extend the original apps with new functionality using TEO operations and primitives. Table 3.4 shows the total changes in terms of lines of code we made to each application. In general, the integration process incurs minimal changes. We develop a TEO driver (as part of our TEO prototype) that manages the TEO runtime on the device. It opens API interfaces as REST endpoints exposed only to *localhost* so that the application can leverage the driver to store data and verify command certificates.

Motion Camera. Motion [146] is a smart camera app that records video clips whenever it detects motions. The users can later review these recorded events. To preserve privacy, all data are

saved locally. We edit Motion's configuration file and implement a post-recording *hook* program that uses TEO to encrypt and store the video recording. This integration not only protects user data but also increases the limited local storage space.

For evaluation, we set the length of event recording to be 1 minute, repeatedly triggering event detection over 100 times to measure runtime latency. We set the group size to be a single user. On average, the 1-minute clip is around 22MB and the TEO driver takes around 2879 ms to process the storage request. These performance numbers are consistent with our microbenchmark results (Table 3.2). Since it only takes $\sim$ 3 seconds to store a one-minute long video recording, we believe that TEO integration would be a useful and practical extension for the Motion app.

Speaker with Voice Assistant. Mycroft AI [147] is a smart speaker app similar to Amazon Echo and Google Home. Users trigger it with a "wake" word, followed by their instructions. A critical privacy concern with smart speakers is that they can record user interactions and upload audio clips to train better machine learning models and for internal analysis [37, 39]. We extend Mycroft so that users' private audio recording data are protected with TEO encryption and access control. Every time a wake word is detected, Mycroft starts recording the following user commands and uploads them to a TEO storage provider. Compared to video data, audio clips are much smaller and highlight the extra overhead in TEO communication. On average, the audio clips are around 72KB, and it takes the app 235 ms to finish storing these clips. This result is slightly higher than our microbenchmark, highlighting that applications storing smaller pieces of data are more sensitive to TEO's overhead and cross-app communications (between Mycroft and the TEO driver). However, we consider this overhead still acceptable since storing clips is done asynchronously and does not block the users' normal interaction with the device.

Smart Doorlock. Complementary to the previous two cases, we implement a smart doorlock application that shows how TEO can protect owners in real time and prevent unauthorized control of the device from non-stakeholders. We develop this app based on an open source door lock project [89] and utilize the Blynk IoT Library [27]. With TEO integration, the app can check for access authorizations accompanying every incoming command, and only executes valid commands from current owners. Our evaluation measures an average increase of 11 ms in latency due to certificate check on the TEO runtime. Low overhead is important in this case, since the process is now on the critical path of the device's functionality and user interactions.

3.7 Discussion and Limitations

Support for Less Capable, Lower-Power Devices. Currently, our TEO prototype supports mobile phones, Linux machines, and Raspberry Pis. These platforms are equipped with modern ARM or x86 processors. Unfortunately, porting TEO to other low-power devices (e.g. using the popular ESP32 series or the ARM Cortex M3 series) is more challenging for two reasons. First, TEO makes extensive use of several heavy-weight cryptography libraries, which are not yet supported by microcontroller-based architectures in these devices. Second, assuming that all of TEO's dependencies have been ported, TEO could encounter high performance overhead due to limited processing power and resources (e.g., ESP8266 only has 160Mhz CPU and 50KB

memory [70]). It would be an interesting future research direction to extend TEO to these lower-power class of devices and, indeed, recent works have proposed novel cryptography protocols to enable public-key cryptography on them [3].

Reducing the Trusted Computing Base. TEO assumes that IoT devices and user mobile phones are trusted to protect user data. Compromised devices can bypass TEO and directly leak users' private data, or impersonate other devices through Cuckoo attacks by relaying network traffic [162]. Malware-infected phones can steal credentials to users' data. Furthermore, TEO assumes that the cloud storage services correctly perform the computation for re-encryption to work. To reduce the trusted computing base, one promising future direction is to expand TEO with a secure hardware infrastructure. For example, we can perform security critical operations inside the Trusted Execution Environment (e.g., Intel SGX, ARM TrustZone) as inspired by many prior works [18, 143, 186, 208, 247] and leverage Trusted Platform Modules and remote attestations [9, 103, 116, 152, 163] to ensure the integrity of TEO programs and operating systems.

Specifically, to mitigate the threats of compromised smart devices, we can borrow insights from many recent works and develop a trusted TEO hub. The hub can act as a network access point for local devices and require all egress network traffic to be encrypted with TEO [65, 98, 209, 239], or redesign the IoT application programming architecture and have the trusted hub to process all user's private data [108, 113, 231].

Deployment Challenges. There are some practical challenges with large-scale use of TEO. In addition to partial availability (Section 3.3.6), identity management can be complex. Admins need to associate a user's public key with their real identity. This could be facilitated by conventional PKIs, third-party services like Keybase [112] or a trusted mediator (e.g., Airbnb holds public keys and identities for hosts and guests). Moreover, storage providers should be compensated since they will host all encrypted data with high availability. To encourage competitive pricing and avoid vendor lock-in, TEO's design does not require strong trust in the storage provider, so this role can be filled by many entities (e.g., building manager self-hosts, public cloud services, or centralized servers in Airbnb). We also provide a reference storage provider implementation in our TEO prototype that uses a simple key value store database.

Monitoring Device Ownership. To ensure their data are always protected by TEO, users should keep monitoring the list of devices they currently own. Otherwise, they might mistakenly think they still are ephemeral owners when the device is re-claimed by someone else. We envision extending TEO mobile apps with monitoring functionality for future deployment to alleviate user burdens. The app can send notifications when the user loses device ownership to help them stay informed. It can also analyze the latest data stored by the device to verify that the user is one of the owners (since metadata is publicly accessible).

3.8 Summary

This section of the thesis illustrates the added benefits for IoT device vendors to offload the role of ownership management and access control enforcement to third-party services. Specifically, device users can enjoy better privacy and security protections and directly control how their data can be accessed. By proposing the design of TEO, we materialize this vision with a set of formally verified operation protocols to ensure their correctness and security. We demonstrate TEO's practicality in terms of low performance overhead and ease of device vendor adoptions with our evaluation results from the prototype implementation.

Chapter 4

CAPTURE: Securing IoT Devices by Offloading Third-Party Library Management

This section of the thesis illustrates the security benefits for IoT device vendors to offload library management responsibilities to a central trusted service. We first present our study on real-world IoT device firmware and demonstrate the widespread issue of mismanaged third-party libraries and its security implications. We identify the opportunity and potential benefits of consistently applying library security patches in IoT devices.

To achieve our vision and reduce the attack surface of future IoT deployment, we propose a new system architecture, Capture, that centralizes library management tasks across heterogeneous IoT devices. We propose the use of a trusted central hub in the home that maintains the security upkeep of common libraries and enables local devices to share the up-to-date library runtime. Although the security benefits of updating library dependencies are fairly obvious, we need to address several challenges in enforcing isolation, preserving device integrity, and reducing adoption barriers to make our solution practical.

The rest of this chapter explains CAPTURE in detail, based on our published paper [239]. We release the source code [237] for the prototype implementation as well.

4.1 Motivation

With their growing popularity, in-home Internet-of-Things (IoT) devices are becoming ripe victims for remote attacks, leading to high-profile incidents such as the Mirai botnet [12]. Compared to traditional network hosts, IoT devices are often more vulnerable due to weak credentials [117, 181, 233], insecure protocols [125], and outdated software [174, 180]. Making matters worse, despite their home deployment, these devices may connect directly to public Internet hosts to send data and even listen for incoming connections [87, 188]. If any of them is compromised, attackers can easily wreak further havoc by moving to other devices on the same network [12, 229].

Although many current IoT exploits originate from misconfigurations, weak credentials, and

insecure applications [6, 117], the extensive use of third-party libraries in IoT devices may have security implications but remains overlooked. Vulnerabilities in common libraries, when left unpatched, can affect a massive number of devices (e.g., CallStrager [232] and Ripple20 [234]). The security impact of vulnerable libraries in *traditional* software systems is well-known [42, 49], with slow rollout of security-critical patches exacerbating the issue [62, 122, 149]. Although one may believe that similar issues of code reuse and library mismanagement are also common in IoT devices, there lacks sufficient evidence and concrete answers for the actual severity of these issues. As we will demonstrate later in the chapter by our analysis, common third-party libraries are widely used. They are often egregiously outdated, leading to many IoT devices operating with publicly-known exploits!

Recent works in IoT security may partially alleviate this challenge, but each has its own limitations. Commercial IoT frameworks and operating systems (e.g., Microsoft Azure Sphere [140], AWS Greengrass [8], and Particle Device OS[165]) all assume the burden of managing a *limited* set of shared libraries provided by the OS. However, developers may use a variety of IoT libraries for functionality [166]. These OSes provide little protection for those custom libraries imported by developers. Alternatively, several proposals attempt to isolate vulnerable devices on the network [65, 98, 209]. Network isolation offers limited flexibility when it comes to mitigating the effects of compromised devices, so these approaches present an inherent security tradeoff whenever devices need Internet access.

4.2 Third-Party Libraries in IoT

Although it is common practice for software developers to integrate mature third-party libraries into their applications, we found a lack of concrete evidence on how popular these libraries are in IoT device firmware and how well they are maintained. Therefore, in this section, we set out to address two key questions largely unanswered by previous work. Specifically,

- *How prevalent is third-party library usage among existing IoT devices?*
- *How diligent are device vendors when it comes to releasing firmware updates that patch critical security vulnerabilities?*

Previous studies [32, 149, 248] that focus mainly on network equipment report widespread vulnerabilities, some of which can be attributed to unpatched third-party libraries. A recent study focusing on smart appliances reports similar findings [6]. However, these studies do not address the state of affairs on current IoT devices, and in particular on how frequently libraries are used and updated. To fill this gap in our knowledge, we conducted a measurement study on 122 firmware releases from 26 devices and 5 popular vendors. We find that third-party library use is prevalent, and even more concerning, that security-essential libraries like OpenSSL often remain unpatched for hundreds of days.

4.2.1 Data Collection

Retrieving Library Information. A potential approach is to analyze the binary images of publicly available firmware images. However, despite the availability of analysis tools [60, 95],

Table 4.1: Summary of devices and vendors included in the measurement. We skip firmware for network equipment since our focus is on smart devices.

Vendor	Belkin	TP-Link	Ring	Nest	D-Link	Total
Devices	12	3	1	7	3	26
Firmware	12	3	1	74	32	122
Libraries	80	5	53	290	93	441
Library versions	103	5	55	400	114	654

validating the resulting information would be time-consuming and error-prone, and the number of devices with easily obtainable firmware images is limited. Instead, we collect vendor-reported information about the use of GPL open-source libraries in firmware release notes, as this disclosure is required by the license terms. While our results may thus exclude information about closed-source and non-GPL third-party libraries, we note that this will, if anything, underrepresent the true prevalence of third-party library use in IoT devices. We used this approach to collect all available data for 441unique libraries across 122firmware releases from 5IoT vendors, dating back to 2011. We manually collected library names and version numbers for 122firmware releases.

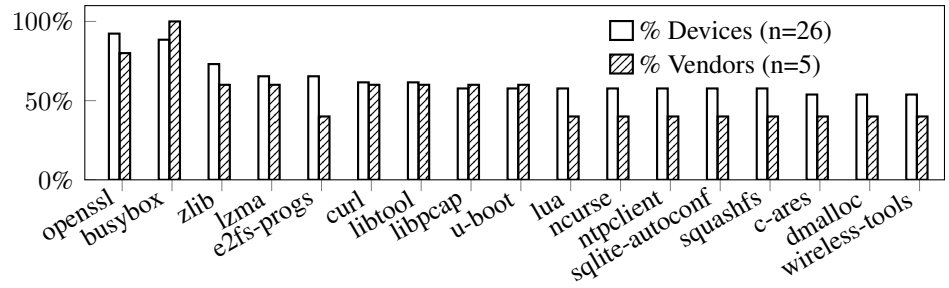
Firmware Selection. We selected 5popular device vendors (Belkin, TP-Link, Ring, Nest, and D-Link) since we were able to find consistent, detailed information about their firmware releases with the required third-party library information. Table 4.1 summarizes 122firmware releases we collected data about. Nest and D-Link provide the most comprehensive information about their firmware release history, dating back to 2011. We use these historical releases to analyze longitudinal patching behaviors. Belkin and TP-Link maintain public information for a single firmware version for each device still under support. Ring releases one summary for all open-source libraries used in their devices, which we categorize as a single generic device with a single firmware release. Table 4.2 includes individual device details.

4.2.2 Results

From the collected data, we aim to characterize two main statistics: the prevalence of third-party library usage in IoT firmware images across vendors, and the characteristics of patch release over time. In particular, our goal for the latter statistic is to understand how quickly a new firmware image is released after a third-party library is updated in response to a known CVE with a corresponding moderate or high severity.

Prevalence. As expected, we found that IoT devices use third-party libraries extensively. Table 4.1 shows that the 122 firmware releases we studied disclosed 441 unique open-source libraries. Counting libraries with different version numbers as unique, this number increases to 654. While some vendors consistently use the same version across images, others do not: for example, of the 12 Belkin devices we studied (each corresponding to one image), there are 80 unique libraries spanning 103 library versions. This finding already suggests problematic patch-

Figure 4.1: List of the most common libraries in all 26 devices across vendors. Among 26 devices, over 50% use these libraries. The most popular ones, OpenSSL and BusyBox, are used by 92.31% and 88.46% of devices. We also show the percentage of vendors who use these libraries on their devices.



ing behavior. While there is a significant variation in the range of libraries in use (441 across just 26 devices), there is a common subset across devices. Figure 4.1 shows the most popular libraries, appearing in at least 50% of the devices. OpenSSL and BusyBox are ubiquitous, used by 92.31% and 88.46% out of a total of 26 devices.

Patching Practices. To better understand the security risk of third-party library use, we examine firmware releases longitudinally, and their alignment with library patches and CVE disclosures. Since historical release data was only available for 5 devices from Nest and D-Link, we use 100 firmware releases for these devices, for a 7-year period (2011-2018).

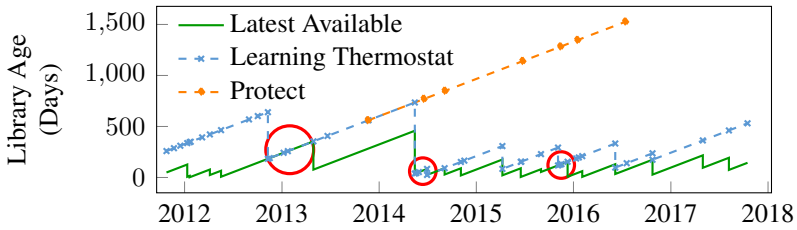
We pick OpenSSL to study library patching practices for two reasons. First, OpenSSL is a popular library used by all vendors in our dataset, except for Ring which uses GnuTLS. Second, OpenSSL is critical for software security and has a well-documented history of vulnerability discoveries and patches [157]. By examining OpenSSL versions in firmware releases and OpenSSL's update history, we analyze vendors' patching behaviors and outstanding vulnerabilities over time.

Figure 4.2 illustrates the "age" of the OpenSSL library, defined as the number of days elapsed since the release date of a particular version. The dashed lines represent the library ages used in different device firmware, while the solid green lines represent the ideal case where the devices can always use the *most up-to-date* library versions. As shown in these dashed lines, device firmware updates routinely lag behind using the latest versions of OpenSSL. In some cases, this extends for hundreds of days. For example, Nest Protect's last firmware release on 2016-07-13 used a 1525 days old OpenSSL version, while the latest available one was released on 2016-05-03 (only 71 days old). Furthermore, there are often multiple new firmware releases made by vendors without incorporating the up-to-date library version, suggesting a missed opportunity. Notably, even devices from the *same* vendor often use different library versions, highlighting the challenge of coordinating upgrades.

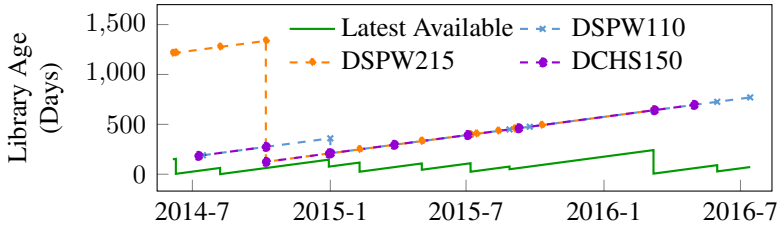
The Nest Learning Thermostat appears to have the best patching practices among devices in our study; it sometimes even used the latest OpenSSL (red circles in Figure 4.2(a)). However, a closer look at how this aligns with known vulnerabilities suggests that even this case reflects unnecessary exposure. Figure 4.3(a) depicts the number of OpenSSL CVEs and in particular

Figure 4.2: OpenSSL library ages in different devices. Dashed lines represent actual library used in the firmware. Each marker indicates a new firmware release. Solid lines indicate the expected library age **if** new firmware release always uses the latest versions, representing a best-case scenario. Red circles highlight cases in which devices *actually* use the latest version.

(a) OpenSSL in Nest Devices.



(b) OpenSSL in D-Link Devices.

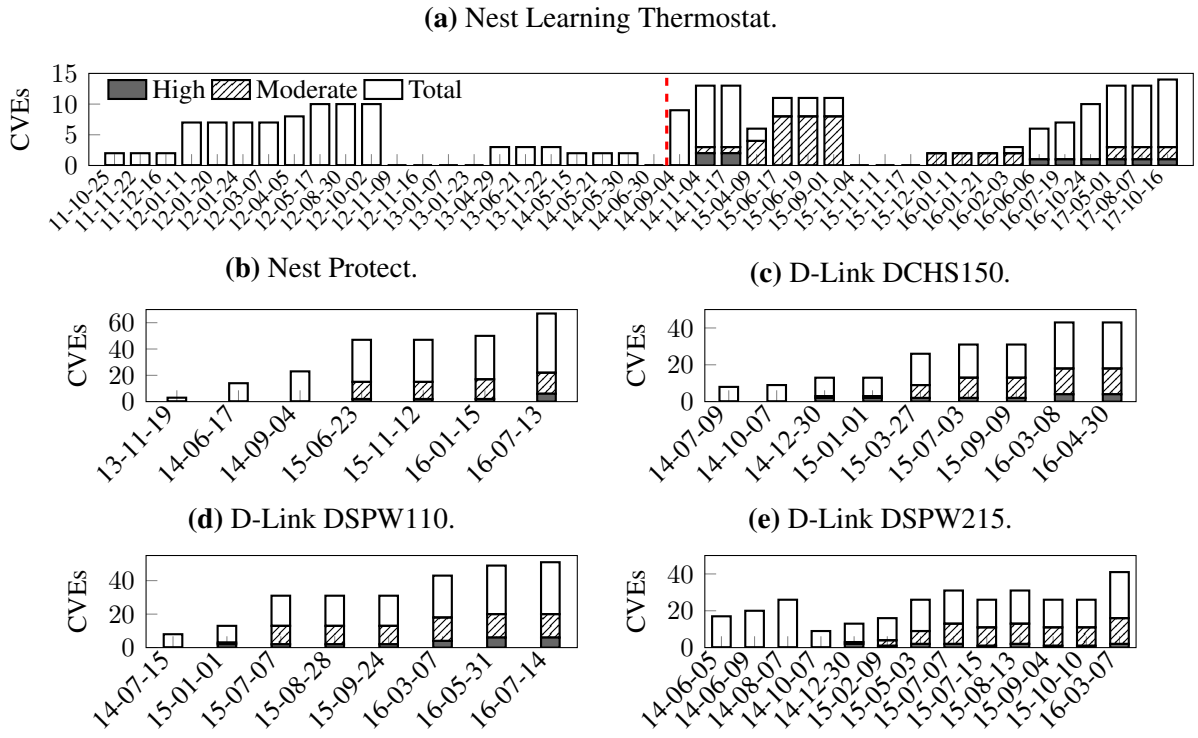


those of moderate or high severity (severity data is only available after August 2014), that apply to each version of the Nest Learning Thermostat in this time frame. Unsurprisingly, the periods corresponding to Figure 4.2(a)'s red circles are not vulnerable, but this only lasts for a few months until multiple vulnerabilities emerge. Importantly, most of these CVEs are avoidable only if the firmware uses the latest OpenSSL.

Hardware Architecture. Many devices in our dataset are Unix-based systems, as 88.46% and 46.15% of devices include BusyBox and Linux Kernel libraries. Teardowns on high-end smart devices [54, 55, 97] often find powerful ARM processors, affirming our findings. Meanwhile, budget-oriented devices may prefer alternative microcontrollers (such as ESP32 and ESP8266 in light bulbs and plugs [4, 5]). Our dataset might under-represent lower-end devices for two reasons. First, they could use libraries provided by chip maker, royalty-free [69]. Second, we had some difficulty searching for open-source compliance notices from several lesser-known vendors.

Key Takeaways and Limitations. Our measurement results reveal concerning statistics about the current state of third-party library management in IoT devices. Just by considering widely used open-source GPL libraries, we show that even market-leading vendors such as Nest and D-Link oftentimes fail to update their dependent libraries promptly. This results in unnecessary exposure to known vulnerabilities. While our data collection methodology is limited to open-source GPL libraries, we aim to shed light on the existing state of IoT library mismanagement using these libraries as indicators.

Figure 4.3: Number of publicly known OpenSSL CVEs in firmware releases. X-axis shows the firmware release date (YY-MM-DD format). We do not have CVE severity breakdowns for data prior to August 2014 (the red dashed line in (a)). For newer libraries, we find many High and Moderate CVEs present in the firmware. Certain Nest Protect firmware releases are skipped due to missing release dates.



4.3 CAPTURE Framework

To mitigate the security threats from outdated libraries in device firmware reported in Section 4.2, we present CAPTURE, a novel architecture for deploying IoT firmware to support centralized management of third-party libraries, alleviating the need for library updates by individual vendors.

4.3.1 Overview

Figure 4.4 provides an overview of CAPTURE. A Capture Hub in the local network centralizes library security updates. Every device has two components: a device firmware (F/W A*, B*), and a remote driver (Driver A*, B*) running on the Capture Hub. Developers can use default drivers (provided by CAPTURE) or implement custom ones to use the latest libraries on the hub. The device firmware and the driver use CAPTURE SDK libraries for network communication. Moreover, the driver uses API wrappers provided by CAPTURE to interact with common libraries on the hub. If vendors need libraries not provided by CAPTURE, they can include custom dependencies in their firmware while still benefiting from CAPTURE's isolation protection. The

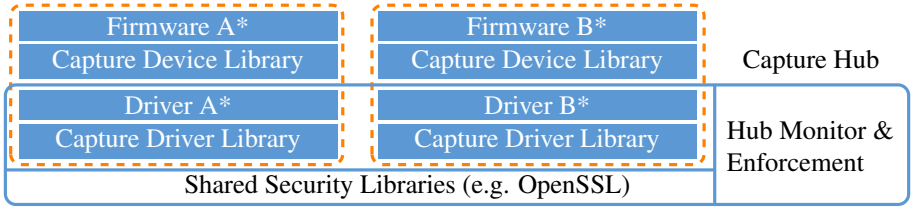
Table 4.2: Details of devices and firmware releases included in the measurement. For each device, we count the number of unique libraries and unique library-version combinations across all firmware releases.

Device	Vendor	Firmware	Release Date	Libraries	Library Versions
WeMo F7C027/F7C028	Belkin	1	2019/08/09	53	55
Wemo Light Switch v1 F7C030	Belkin	1	2019/08/09	53	55
WeMo SNS	Belkin	1	2015/10/14	53	54
WeMo Mini F7C063	Belkin	1	2019/09/05	54	54
WeMo Smart	Belkin	1	2015/06/30	54	55
WeMo Smart F7C046/47/49/50	Belkin	1	2019/09/05	53	54
WeMo WLS040	Belkin	1	2019/09/04	55	55
WeMo Dimmer	Belkin	1	2019/09/03	47	48
WeMo InsightCR	Belkin	1	2019/08/09	53	54
WeMo Jarden	Belkin	1	2019/09/03	53	54
WeMo Maker	Belkin	1	2019/09/03	53	54
WeMo Insight F7C029	Belkin	1	2019/08/09	53	54
SmartPlug - HS100	TP-Link	1	N/A	5	5
SmartPlug - HS110	TP-Link	1	N/A	5	5
SmartPlug - HS200	TP-Link	1	N/A	5	5
Generic Release	Ring	1	N/A	53	55
Nest Cam	Nest	2	N/A	177	186
Nest Connect	Nest	1	N/A	7	8
Nest Detect	Nest	1	N/A	12	13
Nest Guard	Nest	1	N/A	107	108
Nest Hello	Nest	1	N/A	20	20
Nest Learning Thermostat	Nest	57	2011/10/25 - 2017/10/16	140	194
Nest Protect	Nest	11	2013/11/19 - 2016/07/13	18	21
DSPW110	D-Link	9	2014/07/15 - 2016/07/14	75	86
DSPW215	D-Link	14	2014/06/05 - 2016/03/07	72	85
DCHS150	D-Link	9	2014/07/09 - 2016/04/30	51	54

Capture Hub Monitor and Enforce module manages all drivers and provides runtime and network isolation for all devices supported by it.

Threat Model. We assume that the Capture Hub is trusted, and all standard wireless protocols and Linux tools we use to provide isolation are up-to-date to address any known vulnerabilities. We consider an adversary who seeks to compromise IoT devices through known vulnerabilities in unpatched third-party libraries. Unlike prior efforts that restrict devices to explicitly whitelisted hosts (e.g., the vendors' cloud backend) [98, 115], we allow devices to communicate with arbitrary hosts to avoid limiting their functionality. Since local devices (or drivers) may be compromised, our goal is to prevent them from being able to affect other non-compromised devices and drivers in the same home deployment. Attack vectors from zero-day exploits (i.e. no patches available) and non-library vulnerabilities (e.g., weak passwords) are out of the scope of this work. In addition, we exclude side-channel attacks arising from the shared hub access from different drivers.

Figure 4.4: Capture system architecture. Every device consists of local device firmware and a driver on the hub. They form a logically unified entity, Virtual Device Entity (orange dashed box). The Capture Hub maintains a central version of common libraries and has extra monitoring and enforcement modules.



Security Goals. Intuitively, the main goal of CAPTURE is to **centralize library management** by providing a consistent, up-to-date set of third-party libraries for devices in the local network, configured and managed by the central hub. Since we do this by splitting the firmware across an IoT device and a hub, CAPTURE should not introduce *new* vulnerabilities or attack opportunities. For example, CAPTURE needs to preserve **device integrity** by protecting communication that would normally be on the device. Hence CAPTURE needs to prevent any entity from intercepting or impersonating a device with its driver on the hub and vice-versa.

In addition, CAPTURE needs to maintain proper **isolation** between devices and drivers. This implies that compromised *devices* should not be able to communicate with other hosts on the same local network, and that compromised *drivers* on the hub should not affect the operation of devices other than the one that they represent.

4.3.2 Library Update Management

CAPTURE alleviates the burden of patching security-critical shared libraries, enabling device vendors to use the up-to-date versions on the Capture Hub without managing patches themselves. Notably, vendors still implement their device firmware and the corresponding drivers. However, they may be concerned with losing control over devices' stability whenever CAPTURE automatically updates shared libraries. These library updates can potentially cause semantic changes (e.g., new APIs) or unexpected bugs to break the existing functionality of the drivers. Fortunately, prior work on patching vulnerable libraries for Android apps provides an optimistic outlook [52], reporting that 97.8% of apps using libraries with known vulnerabilities can be fixed with a drop-in patched version of the library. To determine whether this finding applies to IoT devices, we analyze the dataset from Section 4.2 for potential impacts of library updates on device functionality. We focus on the OpenSSL library usage in Nest devices, since their dataset has a comprehensive history of versions and upgrades.

OpenSSL Versioning. OpenSSL's versioning scheme uses letters to denote minor security patches and numbers for major changes [156]. For example, an application using version 1.0.2a can upgrade to 1.0.2b to fix bugs and security vulnerabilities, while an upgrade to 1.1.0 indicates new features and APIs. Each major version has an end-of-life date, after which users stop receiving security updates. OpenSSL's staggered release strategy supports multiple major versions at

the same time, providing a buffer to transition between versions. Our analysis of Nest's OpenSSL use finds that Nest always upgrades the major version before the old one reaches end-of-life.

Library Update Strategies. There are three strategies for CAPTURE to support multiple library versions concurrently.

Maintain Multiple Majors in Parallel. The most stable strategy to preserve device functionality is to support all active major versions in parallel. The hub applies security patches for each major version independently. According to the OpenSSL's release history [157, 219], CAPTURE has to support two or three majors concurrently and needs to apply security updates every few months. This strategy will not break any Nest device's functionality in our dataset, since they never use any outdated major versions.

Only Maintain the Latest Major Version. Managing multiple library versions in parallel may become complicated as the number of libraries increases. A simple strategy is only to keep one version per library on the hub, presumably the latest major release. Based on our dataset, Nest devices use a non-latest major version in 1238 out of 2184 days. This strategy will cause version mismatches almost half of the time. Mixing drivers intended for older versions with newer runtime can be problematic. Although OpenSSL meticulously preserves backward compatibility across major upgrades [156], we are pessimistic about third-party libraries ' stability in general. Therefore, we use the major mismatch as a *conservative estimator* of potential functionality breakages. Choosing how many major versions to support demonstrates the tradeoffs between manageability and functionality.

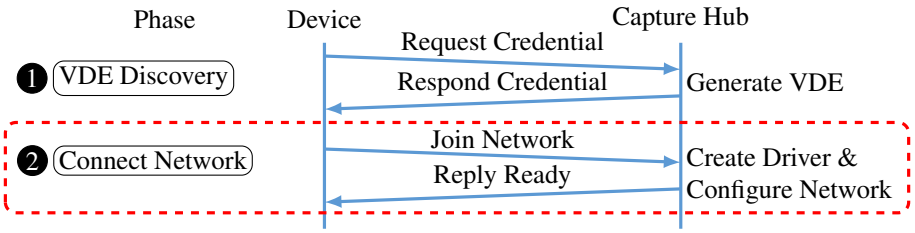
Forceful Major Upgrades after End-of-Life. Vendors could ignore library upgrades so long that it reaches the end-of-life dates. CAPTURE could forcefully upgrade major versions to maintain security at the expense of potential functionality breakages. Since Nest always upgrades OpenSSL to the next major version before the end-of-life dates, we do not have data to measure the impact of a forceful upgrade. However, this tradeoff is a very difficult yet open challenge. Prior works proposed various strategies from blocking devices with insecure libraries [115], quarantining insecure devices locally [65], to preserving functionality at the expense of security [125]. We plan to leave this as a configurable option for end-users to make informed decisions based on their concerns.

4.3.3 Virtual Device Entities (VDEs)

An IoT device supporting CAPTURE comprises of two components: a CAPTURE-enabled firmware on the device and an associated software driver running on a hub, collectively forming a *Virtual Device Entity* (VDE). Note that CAPTURE creates a unique VDE instance for every deployed device. Even if there are multiple identical devices, CAPTURE instantiates separate VDE instances for each of them. CAPTURE ensures confidentiality within the VDE and enforces isolation across different VDEs, as we will explain in the following sections.

Device Bootstrap. Figure 4.5 illustrates the process of bootstrapping new devices and obtaining VDE. A device first connects to a *setup network* with pre-shared credentials, just like traditional home WiFi. In Step ①, the Capture Hub creates a fresh VDE and prepares a VDE-specific

Figure 4.5: Device bootstrap procedure. In Step 1, the device connects to the Capture Hub using a shared setup network. Then it joins a VDE-specific VLAN network in Step 2 (dashed box). Section 4.3.4 discusses more details on network configurations. Section 4.4 addresses potential attacks during bootstrap.



VLAN on the second operation network. After receiving the VDE-specific credential, the device disconnects the setup network and joins the *operation network* (Step ②), where the hub binds the device to its VLAN. This transition won't affect other existing devices, since they are connected to their VDE-specific VLANs already. The hub creates a driver for the VDE, sets up network interfaces and isolation, and enforces resource isolation for the driver on the hub.

4.3.4 Communication Isolation

A CAPTURE-enabled device essentially functions as a "local" device since it can only communicate with its driver on Capture Hub and vice versa. Other communication, such as between local devices or different drivers, is automatically blocked. We achieve this in CAPTURE by creating unique logical networks for each VDE with its own subnet and virtual interface.

The Capture Hub simultaneously manages two separate WiFi Access Points (APs). The first one is a WPA2-Personal AP with pre-shared credentials for the first step of initialization (Figure 4.5), similar to current home WiFi. The second AP uses WPA2-Enterprise and enforces VDE-based isolation. Specifically, Capture Hub creates unique RADIUS user accounts and constructs different virtual Network Interface Cards (vNICs) for each VDE. Using enterprise features such as VLAN and RADIUS authentication, the second AP binds each VDE's device into its own subnet and vNIC. The hub binds the corresponding driver to the same vNIC interface using TOMOYO [206], a Linux security module for mandatory access control. If the driver needs Internet access, the hub creates a designated public-facing port and enables the driver's connection to the port via TOMOYO. We then configure the firewall program `iptables`'s rulesets to block communications across vNICs to achieve VDE-based isolation. CAPTURE's VDE-based isolation is inspired by DreamCatcher [65], which shows vNIC-based isolation is effective against link-layer spoofing. We extend DreamCatcher's network isolation with additional mandatory access control to accommodate Capture Hub's shared driver execution environment.

To bind multiple devices into different vNICs while using a single WiFi AP, we utilize the VLAN isolation feature from WPA2-Enterprise. While WPA2-Personal is common for home users, popular WiFi modules used by vendors to build their products already support WPA2-Enterprise [67]. Hence we believe modern devices can support CAPTURE and WPA2-Enterprise either out of the box or with updated firmware. For legacy devices without WPA2-Enterprise support, CAPTURE can create a new WPA2-Personal AP for each legacy device, however that

may run into software limitations of the number of SSIDs per antenna [65]. An alternative approach is to create unique WPA group keys for each device, isolating hosts under one shared WPA2-Personal network [209]. CAPTURE didn't take this approach as it requires modifying standard protocols.

4.3.5 Resource Isolation

Since Capture Hub executes multiple drivers, a key challenge is to ensure secure and fair resource sharing on the hub. CAPTURE needs to ensure slow or malicious drivers are contained and cannot affect other VDE's availability and private data. Linux containers [132] seem like a natural choice for process isolation. However, they are ill-suited for CAPTURE since each container has a copy of the libraries the driver needs. Whenever the library is updated, all container images would have to be updated and rebuilt, which conflicts with our goal of managing libraries centrally. Instead, CAPTURE provides resource isolation and access control using lightweight Linux system primitives. The Capture Hub creates a new Linux user account per VDE, under which context the associated driver runs, applying standard Linux filesystem and memory protections. We further limit the driver's capability by utilizing the TOMOYO Linux extension and its domain-based security management. We assign each VDE and all of its subprocesses to the same security domain and enforce security policies for networking and file systems. Finally, we used Linux `cgroups` [96], a key building block for implementing containers, to limit the resources used by each VDE. Linux cgroups are known to be an efficient and low overhead mechanism to account for resource usage[167, 225]. Currently we statically set the CPU and the memory resources for each driver to equally share the total system resources, but in the future, we can add support for drivers to specify their resource demands (such as via manifest files during installation, similar to mobile apps) and dynamically enforce them.

4.4 Security Analysis

External Threats. CAPTURE protects devices from external threats by securing the driver components, which are reachable from the Internet. This is done by the Capture Hub, which ensures that the latest library versions are installed automatically and used by the drivers, without the device vendors having to do this. Unlike drivers, the actual devices are isolated from other hosts in the local network. Manufacturers still implement custom firmware running on their devices, meaning that some outdated libraries and vulnerabilities may still exist. However, since the network isolation in CAPTURE only allows communication between driver and device, it limits other hosts from exploiting them. This security protection is contingent on vendor adoptions and properly implemented driver software.

Internal Threats. We consider internal threats which include compromised devices, drivers, and other devices within the WiFi range. CAPTURE prevents compromised local devices from attacking other Virtual Device Entities (VDEs) through network isolation since these devices are confined to their VDE and cannot reach any other hosts directly. Similarly, a compromised driver

is also isolated from other VDE drivers using our network and other resource isolation mechanisms (Mandatory Access Control, cgroups) mentioned above. In CAPTURE, drivers communicate with their associated device using our library runtime, which requires developers to specify the message format between the device firmware and driver. This design prohibits compromised drivers from sensing arbitrary packets to their associated devices and affecting them. Furthermore, drivers cannot communicate with other VDEs on the hub due to our resource isolation mechanisms.

Malicious devices (including CAPTURE-incompatible local devices) can not learn about other VDE's network credentials simply by eavesdropping on the setup network. Although the setup network is a WPA2-Personal AP with shared password credentials, each device actually has its own PTK (pairwise transient key) through WPA2 4-way handshake [125, 220]. However, link-layer encryption provided by WPA2 is insufficient for CAPTURE's network isolation because all drivers will run in the same application layer on the hub. Therefore, we generate a unique network interface and VLAN for each VDE during the bootstrap process (Figure 4.5).

An adversary could potentially impersonate the Capture Hub and perform man-in-the-middle attacks during new device bootstraps (Figure 4.5). This threat can be mitigated by using certificates and public key infrastructure for devices to verify the hub's identity, or other novel device pairing and initialization techniques [91, 190]. We did not implement these features in our prototype since our current threat model focuses on attacks from vulnerable third-party libraries (Section 4.3.1).

4.5 Integration Approaches

We propose three integration approaches for developers to adopt CAPTURE, motivated by current IoT development practices. Our goal is to provide paths of least resistance to help with the adoption while providing flexibility to developers.

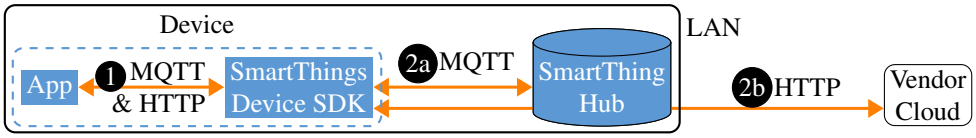
4.5.1 OS Library Replacement

The first approach is to provide a CAPTURE-enabled version of standard OS libraries. Take the OS networking library in ESP32 platform, `WiFi.h`, for example. Devices use APIs from this library to connect access points, maintain web servers, and communicate over sockets. We provided a fully-compatible CAPTURE-enabled library, named as `CaptureWiFi.h`. Developers just need to make minor changes to use CAPTURE, such as replacing the `#include <WiFi.h>` statement and initializing CAPTURE global runtime. We provide a default CAPTURE driver on the hub, which acts as a proxy to relay network traffic. If the original device works as a webserver, we open a public-facing server on the driver to forward traffic and restrict network traffic between driver and device.

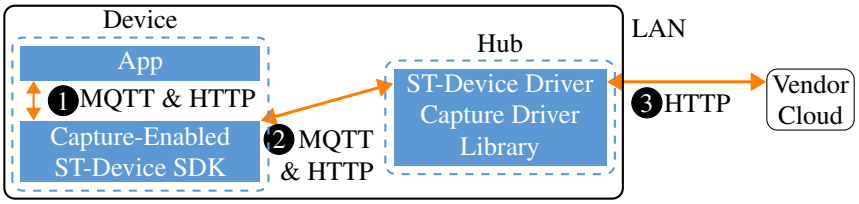
This approach is platform-dependent. We need custom implementations for specific OS APIs and libraries. However, this is a one-time effort that can then be used by device developers with minimal porting effort. For example, all of our prototype apps use the same ESP32 modified library runtime.

Figure 4.6: Integration using IoT framework SDK extension.

(a) Current deployment requires SmartThings Hub for networking.



(b) Capture-enabled SmartThings devices move all network communication onto the device drivers at the central hub.



4.5.2 IoT Framework SDK Extension

Similar to replacing OS APIs, our second approach is to extend the SDK of a popular IoT framework to support CAPTURE. IoT frameworks (e.g., Azure Sphere [140], Particle OS [165], and Samsung SmartThings Device SDK (ST-SDK) [184]) provide rich functionalities to differentiate from standalone embedded device OSes with limited networking APIs. For example, Azure Sphere [140] and Particle DeviceOS [165] provide APIs to communicate with their native cloud backends; Samsung SmartThings Device SDK [184] offers local devices the option of using the SmartThings Hub as an MQTT broker.

In this case, the developers of the IoT frameworks can incorporate CAPTURE by modifying their SDK implementation while preserving existing functionality. As a proof of concept, we added CAPTURE support into the ST-SDK, which enables third-party devices to use their SmartThings Hub. Figure 4.6(a) shows how an example device would integrate with the ST-SDK, similar to a custom OS library. A locally installed SmartThings Hub (ST-Hub) provides functions such as MQTT brokers, which device developers can directly invoke using ST-SDK APIs. A device-side library manages the underlying connections with the ST-Hub. We develop a CAPTURE-augmented ST-SDK library (Figure 4.6(b)), so that device developers only need to switch their ST-SDK library runtime without modifying their application. Since the SmartThings Hub is proprietary, we were only able to re-create their known functions such as MQTT brokers using corresponding open source versions. We provide a default SmartThings-compatible driver to mimic the ST-Hub operations in CAPTURE.

4.5.3 Native Driver Development

The two prior approaches provide default drivers on the Capture Hub to aid developer adoption. As a complementary approach, we developed a CAPTURE Native Driver SDK, for developers to implement their own custom drivers with much more flexibility. To motivate this, consider an IoT device with a web server. Using the previous approaches, the default driver on our hub will create another public accessible web server for new connections, and relay incoming client

connections to the device local-only web server. However, this may cause unnecessary latency to serve the web request since both inbound and outbound traffic has to go through the hub and processed by two webservers. To address this, we propose the CAPTURE Native Driver SDK for developers looking to build customized drivers. Developers can use our SDK APIs to access security and networking functions on the Capture Hub, and even offload some computation to the hub.

4.6 Implementation

4.6.1 Core Hub Functionality

We implement the Capture Hub using a Raspberry Pi 3B+ with Linux in 1874 lines of C++ [237]. We use the TOMOYO Linux security module [206] and `iptables` to implement the Virtual Device Entity based isolation mechanisms. Our hub uses hostapd [99] to manage WPA2 Personal and Enterprise WiFi APs. The main CAPTURE program listens for new connections on the setup network, and upon request, creates a new VDE for the incoming device, allocates a new VLAN subnet with fresh RADIUS credentials, launches the corresponding driver program based on the device type, and updates the TOMOYO and `iptables` rulesets accordingly. The main program stores all metadata for each VDE locally. While our current prototype does not address device removal, this functionality can be added in a straightforward manner.

Optimizations. Existing applications often use blocking network calls. During prototype development, we observed a pathological case wherein the application only communicated using one sequential byte at a time. Clearly, adapting such applications into CAPTURE introduces a significant performance penalty, as each read request will incur one round of communication with the driver residing on the hub. We found that without correction, this can lead to a 9.56x latency penalty for the simple Web Server app (listed in Section 4.6.2).

The first optimization we perform to address this issue is to introduce read and write buffers on the device. When an Internet host sends data to the driver, the payload is forwarded to the local device in batch. Subsequent read calls from the device will just retrieve the payload from the local buffer. Similarly, using write buffers enables network writes to be non-blocking I/Os, aggregating multiple payloads into chunks in one round of driver communication. We found that this reduces the latency penalty for the Web Server app from 9.56x to 1.62x, largely due to the reduced number of round trips to the hub.

Although the previous approach reduces average latency overhead to an acceptable 1.62x, it still incurs a median increase of 31 ms. We were able to attribute this to the poor wireless performance on the budget-oriented ESP32 microcontroller, where a single packet transmission can take up to 6 milliseconds. To reduce the total number of packets sent, we extended the protocol header fields and aggressively coalesce small packets throughout our protocol. One concrete example is proactively loading read buffers after accepting new clients, where previously the device needs to send two messages to check client status and fetch data to read, respectively.

Applying protocol optimizations and message coalescing bring down the median latency overhead to 1.2x (+10 ms), using the Web Server's baseline performance as a reference. Given

Table 4.3: Prototype applications and descriptions.

Abbreviation	App Name	Platform	Description
WEB	Web Server	ESP32	Standard web server to display and manage GPIO on/off status.
CAM	Camera	ESP32	Stream live video, take pictures.
SM	Servo Motor	ESP32	Adjust the speed of a servo motor.
CP	Color Picker	ESP32	Change the color of LED light bulb.
WS	Weather Station	ESP23	Monitor weather with a BME sensor.
TH	Temperature & Humidity	ESP32	Display temperature and humidity data from DHT sensor.
ST-L	SmartThings Lamp	SmartThings	Subscribe to MQTT broker to receive on/off message.
ST-S	SmartThings Switch	SmartThings	Publish to MQTT broker to issue on/off message.
MM	MagicMirror	Linux	Smart mirror display with online data such as news and weather.

that the ESP32 takes 5-6 ms to send a single packet, this approaches the limit of what can be done without better hardware. Detailed results are discussed in Section 4.7.1.

4.6.2 Benchmark Applications

To evaluate CAPTURE, and explore different approaches for integrating apps, we developed 9 prototype applications (Table 4.3), including smart devices, Linux applications, and IoT frameworks, and 3 IFTTT automation applets for benchmarking (Table 4.4). CAPTURE provides runtime libraries for device firmware and drivers to handle network setup and communication with the hub. The device-side library was implemented in 1335 lines of C++ code while the driver-side library varies.

Prototype Apps. We collected 6 open source applications from popular online forums and tutorials [66, 88, 211], and adapted them to use CAPTURE. We chose the Espressif ESP32 platform given its reported popularity [4, 5] and use in hundreds of millions of IoT devices [68]. We implemented a generic default driver to support the OS Replacement approach, which required 166lines of Python.

IoT Framework. We extended the Samsung SmartThings Device SDK (ST-SDK) [184] to showcase integrating CAPTURE with existing frameworks (Section 4.5.2). ST-SDK is open-source, whereas other proprietary alternatives (e.g., Azure Sphere and Particle OS) raise challenges for replication and comparison. CAPTURE-enabled devices cannot work directly with unmodified SmartThing Hub, so we analyzed ST-SDK's codebase and replicated its functionality with a driver that executes on the CAPTURE hub. We then adapted sample applications provided by ST-SDK [185] into CAPTURE.

Linux apps. Some IoT devices are powerful enough to run a Linux OS and applications (Section 4.2), so we adapted Linux smart devices into CAPTURE to demonstrate its capability. We selected MagicMirror, a project with over 12K Github stars [139], that uses Raspberry Pi with a display to function as a smart mirror, displaying custom content (e.g. news and weather). Internally, the app includes a webserver and a browser to display the webpage. We migrated MagicMirror into a CAPTURE prototype using the custom driver integration (Section 4.5.3) and separated the server component to the driver on the hub, keeping the display parts on the firmware.

Automation Applets. To better measure CAPTURE's macro-benchmark performance impact on real-world scenarios, we implemented several home automation applets developed for IFTTT [104]. Prior work [138] categorized IFTTT applets by trigger-action service types (Device ⇒ WebApp, WebApp ⇒ Device, Device ⇒ Device) and reported an average execution latency of several seconds. We implemented CAPTURE-enabled devices for all three trigger-action service types (Table 4.4), using the Web Server app (c.f. Table 4.3, WEB) on ESP32 in place of physical lights and switches, since it can control GPIO pins. Since ESP32 boards are lower performance and slow at performing SSL encryption, integrating these devices into CAPTURE often improves performance due to our hub hardware being more capable. To provide a fair comparison, we also implement "mock" lights and switches directly on the Raspberry Pi and measure the latency impact from CAPTURE integration as well.

4.7 Evaluation

Our evaluation aims to answer three primary questions.

- How much performance overhead do key device functionalities incur on CAPTURE versus their native platform, and is the amount tolerable for typical home use?
- Can the Capture Hub scale to home deployments with hundreds of devices in the near future, and how many devices can our prototype reliably support at once?
- Roughly how much effort is required to port existing IoT devices to CAPTURE, and do the integration approaches in Section 4.5 entail meaningful differences in the effort?

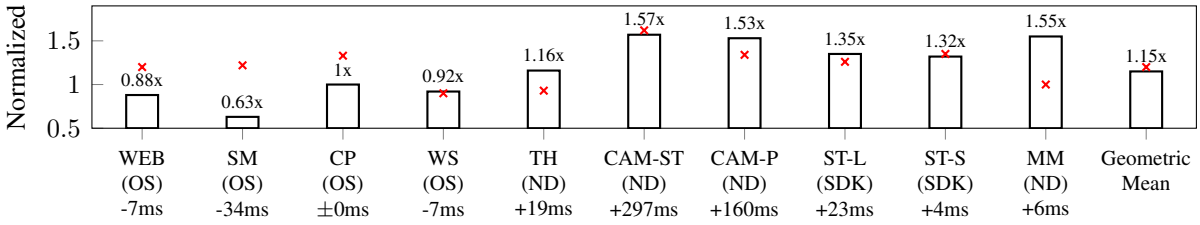
Our experiments were performed in a laboratory setting on 9 prototype devices (Table 4.3) and 3 IFTTT automation applets (Table 4.4). We use one Raspberry Pi 3 B+ as the Capture Hub and another Raspberry Pi and multiple ESP32 boards for prototype apps. Our evaluation results show that CAPTURE typically incurs low overhead (15% latency increase, 10% device resource utilization), insignificant impact on applets from real-world automation platforms, and can support hundreds of devices for a single Capture Hub.

4.7.1 Performance Overhead

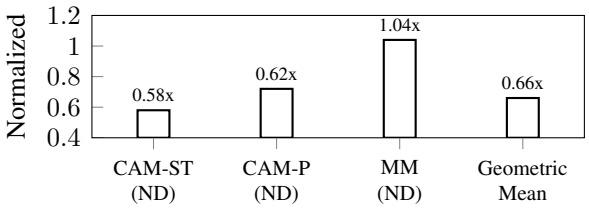
Setup. We compare the performance of apps running on CAPTURE to that achieved by their original implementations. Because many IoT devices and automation apps are event-driven, they usually transmit a small amount of traffic but are sensitive to delays in latency. We categorize

Figure 4.7: Performance overhead for all prototype apps. Data are normalized to results from the orignal apps. CAM has two modes: STreaming videos and taking Pictures. We denote integration approaches in parentheses: OS Replacement, Native Driver, and Framework SDK Replacement. Based on geometric means, Figure (a) shows a 15% latency increase and Figure (b) shows a 34% throughput reduction. Figure (c) shows the CAPTURE-enabled firmware incur around 10% more on-device resource utilization.

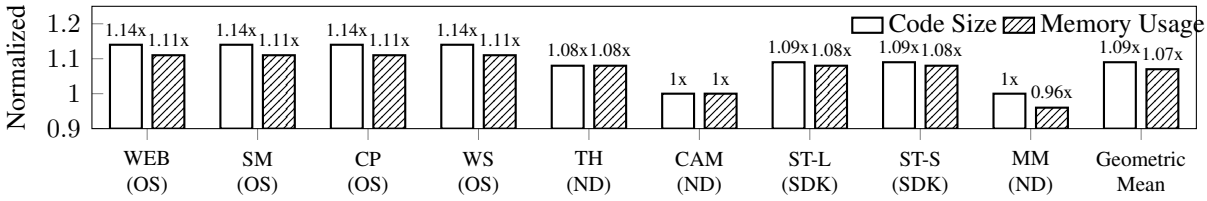
(a) Normalized average latency and numerical differences. Red crosses show *median* latency changes.



(b) Normalized average throughput.



(c) Application code size and memory usage on-device.



prototype apps (Table 4.3) into two categories: latency-sensitive and throughput-sensitive. We measure application-layer latency for all of them, but only measure the throughput reduction for the second group (such as a streaming camera). For most applications, we use Apache JMeter [14] to benchmark *average* and *median* latency for 500 HTTP requests. For the streaming camera (CAM), we measure the video latency by pointing the camera towards a millisecond clock and calculate average delays from 50 readings. For the SmartThings apps (ST-L and ST-S), we add instrumentation to send a notification packet to the hub so that we can calculate the time duration between the first MQTT message and the final notification from Wireshark's packet capture history. Finally, we measure the firmware code size and memory utilization on the device.

Simple Integration with All Apps. We aim to conservatively estimate CAPTURE's performance impact assuming minimal burden on the developer. Hence, we first try to integrate apps

with either OS or SDK replacements, since these require minimal modifications by the developer. If this attempt fails (for example, the app requires features not supported by our current prototype), we develop simple native drivers without spending too much engineering effort on app-specific optimizations.

Figure 4.7(a) shows the normalized latency for integrated apps. On average, apps experience a 15% latency increase due to the extra processing by the drivers on the hub. The baseline apps for the comparison process everything on the device and communicate directly with external hosts. After CAPTURE integration, external hosts need the drivers on the hub acting as a proxy. For example, the camera streaming app driver needs to retrieve the raw footage from the device and forward it to the viewers. These extra steps introduce overhead to the end application. However, as Figure 4.7(a) shows, most apps experience a modest latency change between -34 ms and $+23$ ms. Given most apps' event-driven nature, this minor increase in absolute latency should not impact the quality of services for end applications. CAM app experiences the most substantial latency increase, increasing from 523 ms to an average of 820 ms ($+297$ ms), and a 40% FPS throughput reduction. However, the relative increase (1.6x) is on par with other apps. Since the baseline latency is very high, we believe the original app is not designed to be real-time for ESP32, and thus we did not further optimize its driver.

Several of the apps integrated with OS-Replacement see improved *average* latency results. This is because CAPTURE-integrated apps perform more consistently, while the ESP32-only baselines occasionally experience latency spikes (thus having higher average results). *Median* results are more robust against outliers, and confirm CAPTURE often increases latency slightly. The overall results show that CAPTURE offers comparable performance to the baseline for most requests.

We measure the throughput overhead for several throughput-sensitive apps and report results in Figure 4.7(b). For throughput metrics, we choose FPS for streaming, packet transfer rates for taking pictures, and full web page load time for the complex MagicMirror dashboard. The Camera app has a modest throughput reduction of around 40%. We observe no throughout drop for the Linux-based MagicMirror benchmark. Figure 4.7(c) shows that the CAPTURE firmware is, on average, 10% larger and uses 7% more *on device* memory. We only measure the code increase for ESP-based devices given they have limited flash storage.

4.7.2 Overhead Perceived in the Real World

We implemented several IFTTT automation applets and measured CAPTURE's impact on latency (Table 4.4). We programmatically trigger applets 30 times, reporting the average end-to-end latency. These results show moderate variances, largely due to the fact that these applets interact with remote cloud services (IFTTT, Google Sheets, and email servers), which is consistent with results from prior work [138]. Applets A1 and A2 show insignificant latency changes from CAPTURE integration, indicating the communications to Internet services as the performance bottleneck. Applet A3's ESP32 integration demonstrates a benefit of CAPTURE for low-budget devices. A3-ESP32 baseline has high latency due to compute-intensive tasks such as TLS encryption, while A3-Raspberry Pi and CAPTURE-integrated ones have comparable latency results.

Table 4.4: Average latency for automation apps with standard deviations (30 runs). Overall, CAPTURE has insignificant impacts, with noteworthy improvements on A1 and A3 (ESP) due to offloading TLS operations on the hub. See Section 4.7.2 for further analysis.

ID	Service Type		IFTTT Applet Rule	ESP32		Raspberry Pi	
	Trigger	Action		Original	CAPTURE	Original	CAPTURE
A1	Device	Web App	Turn on switch. ⇒ Add line to Google Sheet.	2.65 ± 0.42	2.00 ± 0.35	2.04 ± 0.66	1.83 ± 0.75
A2	Web App	Device	New email arrives. ⇒ Turn on light bulb.	2.93 ± 0.82	2.93 ± 0.90	2.62 ± 0.62	2.83 ± 0.87
A3	Device	Device	Turn on switch. ⇒ Turn on light bulb.	2.21 ± 0.43	0.81 ± 0.16	0.94 ± 0.28	0.88 ± 0.35

4.7.3 Scalability

Since our Capture Hub executes all drivers on the hub, its resources limit the number of devices it can support. Among resources including memory, CPU, network interfaces, and private IP addresses, we identify the memory capacity as the key scaling bottleneck. The default driver for OS replacement uses the least amount of memory (3.7 MB) while the MagicMirror's driver uses the most memory (42 MB) as reported by `smem`'s Proportional Set Size [207]. Therefore, we emulated a deployment of 40 devices using the default drivers and 10 devices with MagicMirror drivers on a single Raspberry Pi 3B unit (1 GB RAM, quad-core). This setup uses 664 MB memory, but the CPU load average never exceeds 0.8 (max 4.0, due to four cores). Network virtual interfaces and subnets do not impose any practical limits with fine-grained assignments [175]. While the RAM on the hub is a limiting factor, several inexpensive platforms exist with more memory (e.g., Raspberry Pi 4 with 8 GB RAM for $75 [173]), which can potentially support hundreds of devices.

4.7.4 Integration Efforts and Tradeoffs

Integrating apps by replacing OS libraries or framework SDKs is straightforward, requiring modifying less than 10 lines of code after importing the CAPTURE device library. Developing native drivers is more involved since it requires declaring a custom message format for device-driver communications and implementing the driver while delegating the network management to CAPTURE's library runtime. The most sophisticated CAM driver we implemented was 817 lines of Python.

We demonstrate the tradeoff between ease of adoption and performance impact by analyzing different integration approaches for the Web Server app. Although we spent considerable effort optimizing the default OS-replacement driver, it yielded a modest 12% average latency reduction over the baseline ESP32 app. The integration only requires changing a few lines of the original code. In comparison, implementing a native driver for this app significantly reduces latency by 36% over the same baseline. However, to implement the driver, we modified 264 lines of source code to process device-driver communication and customize protocols.

4.8 Discussions and Limitations

Vendor Incentives and Adoption Challenges. Vendors may be incentivized to use CAPTURE because they can offload the security upkeep responsibility to a central trusted entity (the Capture Hub). They no longer need to keep applying security patches themselves, a task they often lag behind (Section 4.2). CAPTURE's isolation design also helps protect vendors from other compromised devices in the user's local home.

There might be several hurdles for vendor's adoption. We have already proposed various integration approaches Section 4.5 to reduce adoption costs for existing devices and hub's library management strategies Section 4.3.2 to alleviate vendor's loss of agency and to avoid breaking functions.

The need for firmware splitting may pose another major roadblock for vendors. They have to bear the extra onus of developing two separated pieces of the "device" and the additional overhead in signing and logistics involved in firmware updates. Implementing CAPTURE drivers and new firmware would require vendors to change significantly from the current status quo and would induce extra engineering efforts.

Single Point of Failure. CAPTURE's centralized design means that the Capture Hub is a potential single point of failure; this is part of our threat model (Section 4.3.1), where the hub is assumed to be trustworthy. If the hub is compromised by vulnerabilities or privilege escalation bugs like those on conventional systems [31, 48], the integrity and confidentiality of the installed devices will be likewise compromised. By centralizing the management of security-critical updates, and providing additional isolation between devices, we hope to contribute to improving the overall security posture of devices deployed within the network (i.e., relative to the status quo). However, this improvement is contingent on vendor adoption.

Centralization may lead to a less robust network even without adversarial compromise. If the hub goes down, devices would lose network connectivity and drivers become unresponsive. Because most device firmware controls local actions (e.g., managing the on/off states for smart plugs), most devices should still function (e.g., through physical buttons on the device). Capture Hub failures, in this case, largely resemble network outages and router failures in current smart homes.

Protocol Compatibility. Since CAPTURE isolates devices, link layer discovery and local network scanning no longer work. One such example is UPnP, an infamous protocol for posing security threats in IoT devices [117, 124] and recent exploits like CallStranger [232]. A future direction for our work is to provide a secure centralized discovery service on the Capture Hub itself with co-located drivers and shared libraries, substituting link layer discovery and mitigating fallout like CallStranger. With that said, many smart devices have companion smartphone apps that communicate with the device via a cloud service to support access to the device behind a home NAT. As communication through the cloud will not be impeded by our approach, we believe that the practical impact of CAPTURE's isolation on everyday use will be minimal.

There are other potential security improvements, which are out of the scope of the current security goals for CAPTURE and threat model. We do not support alternative wireless proto-

cols such as BLE, Zigbee, and Z-Wave since Internet-based attacks over WiFi, the focus of our work, impose significant threats already. As future work, we can look into incorporating related works in securing other wireless protocols [98, 242] into CAPTURE's centralized hub design. In addition, CAPTURE does not address potential attacks due to weak security practices, such as the use of default credentials. However, CAPTURE's Virtual Device Entity isolation blocks compromised devices from exploiting any other devices' vulnerabilities.

Augmenting Device Resources. Another opportunity that we have not explored is to use the hub's computation resources to augment the limited resources of local devices. Specifically, by introducing additional CAPTURE APIs, we can extend the storage and processing capability of low-power microprocessors on the device to the hub.

Firmware Splitting. CAPTURE proposes splitting monolithic firmware into remote and local components, an approach that could face practical challenges, such as data serialization, consistency, and fault tolerance. These issues are not uncommon to many distributed systems that make use of RPC-like components and have been studied extensively [24, 81, 86, 176, 199, 213, 214, 216, 221]. While our prototype implementation does not make use of all of these advances, CAPTURE can benefit from this work to enhance its robustness and reliability. We view this as important future work.

4.9 Summary

This section of this thesis illustrates the benefits of offloading library management responsibilities for IoT device vendors to a trusted third party. Based on historical data, we can identify the necessity of better library management and timely security patching for future IoT devices. To achieve our goal, we propose the design of Capture, a new system architecture for IoT device deployment. We demonstrate that Capture is a practical solution to mitigate the security challenges rising from the prevalent use of outdated libraries while maintaining the security and performance requirement of individual IoT device vendors.

Chapter 5

VERISPLIT: Efficient Computation Offloading for IoT Devices with Neural Network Applications

The previous two chapters of this thesis have addressed security and privacy concerns mainly from the device users' perspective. In this chapter, we will investigate another opportunity for functionality splitting that may directly incentivize IoT vendors. We aim to provide an efficient computation offloading architecture to augment the limited computing resources on IoT devices while alleviating vendors' cloud resources and infrastructure management overheads. Meanwhile, device users can benefit from lower operating costs due to more efficient utilization of local resources.

With the growing adoption of IoT devices, smart homes are packed with more powerful computers and machines with special machine-learning accelerators (e.g., GPUs and multi-core processors). On the other hand, many resource-constrained IoT devices must rely on first-party vendor cloud services for transitory computing demands, such as machine learning inferences, under the current monolithic device design. There is a missed opportunity to utilize available computation resources in IoT home deployments to fulfill these demands.

In this thesis chapter, we present the design of a novel offloading architecture, VERISPLIT, that enables local IoT devices to share computation resources securely and privately. By focusing on machine learning inferences, VERISPLIT provides a practical solution to convince IoT vendors to adopt while addressing challenges in protecting privacy for user data, confidentiality for proprietary machine learning models, and integrity of the offloaded computation.

The rest of this chapter explains VERISPLIT in detail.

5.1 Motivation

Several Internet-of-Things (IoT) devices now support diverse functionality enabling the vision of smart and connected homes (e.g., smart cameras [223] and video doorbells [7]). Often these devices are inexpensive to purchase, and given their limited compute capabilities, device vendors set up dedicated cloud services to support advanced services like serving inferences using ma-

chine learning models as additional subscriptions [189, 224]. Since cloud offloading can incur high operational costs, researchers have proposed techniques to minimize works to be offloaded, such as executing part of the machine model on the local devices [111, 246].

Meanwhile, users of smart homes might have other computationally capable devices in their homes, such as gaming consoles, computers and mobile devices, and high-end IoT devices with expensive hardware (e.g., vacuum robots with GPUs [170]). These devices are wall-powered, have significant computation resources, and are often idle since they are used sporadically. It seems natural to try to broker a deal between the manufacturers of the resource-limited IoT devices to offload certain computation tasks to the more powerful (local) machines (when they are less busy). The worker devices can even potentially charge a lower rate for leasing out idle local resources and pass along any savings, compared to the status quo of offloading to the cloud, to the device maker or the homeowner. In this project, we focus on offloading ML inferences rather than just general computation offload [36, 47] since these workloads are becoming increasingly important for IoT settings [29, 120, 121, 136]. Moreover, we leverage the unique characteristics of machine learning workloads to develop secure and practical solutions, while it is challenging to address similar concerns for general-purpose offloading.

Specifically, there are three major security and practicality concerns we must address to convince IoT device vendors to adopt our vision of local offloading with third-party devices. First, it is necessary to provide input data with third-party workers for computation. Offloading devices must preserve the **data privacy** if users do not want to share their data with third parties. Second, offloading devices must also provide the model weights to workers, which may be proprietary and raise concerns of **model confidentiality**. Finally, ensuring the **integrity** of the offloaded inferences is critical [160]. Otherwise, cheating workers can perform inferences with lightweight models to save computation costs and remain undetected.

Many recent research works have been proposed for secure offloading of machine learning inferences, such as designing cryptographic protocols for data privacy [79, 102, 110, 142], applying homomorphic encryption for model confidentiality [106], and leveraging verifiable computation to provide inference integrity [78, 134, 208, 218]. These approaches, while indeed promising in providing strong security guarantees, often involve heavyweight computation and high communication overhead, diminishing the values of computation offloading and the practicality of such solutions (e.g., secure inference on ResNet-32 raises the latency from hundreds of milliseconds to 3.8 seconds [142], and some assume high-speed communication channel like PCI-e bus [208]). Moreover, cryptographic approaches usually require finite field arithmetic, affecting the model's quality (such as reducing accuracy when models are initially trained with floating point arithmetic operations) [208].

5.2 VERISPLIT Overview

VERISPLIT is a comprehensive framework designed to enable local IoT devices to offload machine learning inferences securely and efficiently. In this section, we start with a high-level overview of VERISPLIT, followed by our detailed design goals (Section 5.2.1) and our threat model (Section 5.2.2). The next few sections address solutions to each design goal in detail (Sections 5.3 to 5.5).

Figure 5.1: VERISPLIT deployment example in a smart home. The IoT device (a smart camera) can offload ML inferences to one or more local devices by sharing input data and model parameters (①) and receiving results (②). Later, the camera can verify the integrity of the inference by repeating selected computations and comparing the results (③), doing it asynchronously to avoid additional latency during inference.

Figure 5.1 illustrates the overall workflow of VERISPLIT offloading across local devices using a resource-constrained IoT device (a smart camera in this example). As the offloading device, it first goes through setup processes with local devices (not shown in the figure). This setup process includes various steps depending on the requirements (data privacy, confidentiality, and integrity), and the offloading device shares model weights with workers.

During inference, the camera shares the input data with one or more workers (Step ①). VERISPLIT needs multiple workers to support model confidentiality. To support only data privacy or integrity, a single worker is sufficient. Then, workers respond with the inference results and additional proofs for integrity, if applicable (Step ②). Next, to check the integrity of the offloaded computation, the camera can decide when and how many results to verify thanks to VERISPLIT's asynchronous and partial verification design.

We integrate two types of offloading solutions to accommodate various model architectures and desired security and privacy properties in VERISPLIT:

- **Holistic Offload.** The most straightforward approach is to offload the model in its entirety. The device shares all model parameters with workers and sends input data during inference. After executing the full model, workers reply with the final predictions. Compared to local execution, holistic offloading greatly improves performance but can only ensure inference integrity. However, this method cannot support data privacy and model confidentiality.
- **Layer-by-Layer Offload.** To protect data privacy and model confidentiality, we propose a layer-by-layer offloading alternative due to non-linearity across layers. The device can choose which layers to offload and offload them individually. This approach introduces much higher communication overhead; as a result, it may not be suitable for certain use cases or network conditions.

5.2.1 Design Goals

We want to propose a practical solution to motivate IoT device vendors to enlist help from other co-located devices in the same user's home. Therefore, we must address many common concerns related to data privacy, model confidentiality, computation integrity, and performance overhead

from the **offloading device vendor's** perspective.

Protecting User Data from Third-Party Workers. It is necessary for the offloading device to share input data with workers. However, this process inevitably leaks smart home users' private data to third parties. If the user does not want to expose their information to other local devices, VERISPLIT must support data privacy while preserving the functionality and accuracy of the offloaded computation.

Preserving Model Secrets. Device vendors may hesitate, or even refuse, to share their models with third-party workers due to intellectual property concerns. To protect vendors' models, VERISPLIT must provide practical solutions that prevent third-party workers' unauthorized use of secret models.

Ensuring Correct Computation. A fundamental requirement for offloading is to ensure the correctness of the computation. Otherwise, lazy workers may avoid work and return arbitrary results. Therefore, VERISPLIT should provide efficient verification solutions so that device vendors can easily check the results reported by workers are computed correctly.

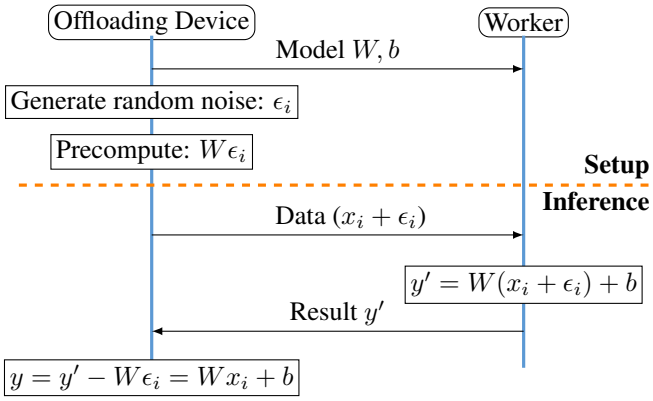
Practical and Low Overhead. While addressing the aforementioned secure and private offloading features, VERISPLIT needs to have low overhead to be practical for different IoT device use cases. We provide comprehensive analyses regarding VERISPLIT's performance trade-offs.

5.2.2 Threat Model

We assume workers performing the computation are untrusted. They may attempt to steal any data the offloading device shares, such as the inputs for inference and the ML model parameters (e.g., layer weights and bias), raising concerns for data privacy and model confidentiality. In addition, workers may have incentives to "cheat" by executing inferences using lightweight models or reporting incorrect results to save computation costs or reduce application accuracy from competing IoT vendors. Therefore, it calls for verification solutions to ensure inference integrity. We assume multiple workers used in the *same* round of inference offloading are non-colluding (e.g., belonging to different vendors) as a prerequisite for model confidentiality. This assumption of non-colluding workers has been commonly used in various prior works [144, 177].

Such a strong attacker model may be too restrictive in certain cases and can be relaxed to reduce the performance overhead, depending on the use cases and deployment environment. Specifically, if the smart home user can trust the worker devices with their personal data (e.g., if they use their own PC as workers), they may not require data privacy protection. However, model confidentiality and integrity are still important for *offloading device* vendors. On the other hand, if the offloading device only cares about integrity (e.g., when using public models for feature extraction and keeping proprietary layers on-device, as used in transfer learning applications), they can further relax the requirement of model confidentiality, further reducing VERISPLIT's overhead.

Figure 5.2: VERISPLIT's workflow for data privacy. During setup, the device shares model parameters with the worker, generates multiple one-time noises ϵ_i, and precomputes their values $W\epsilon_i$. For each inference, the device applies a mask ϵ_i to the input x_i and subtracts the precomputed values from the offloading results.



5.3 Data Privacy

This section discusses how VERISPLIT utilizes linear arithmetic and one-time masks to preserve users' private data during offloading. We do this to prevent third-party workers from accessing user data while providing inferences.

Prior research has proposed using cryptography for data privacy in inference offloading (e.g., garbled circuits, homomorphic encryption, etc.). These approaches are still heavyweight, despite recent advances significantly improving performance [110, 142]. For example, Delphi [142] takes 3.8 seconds for a single secure inference of ResNet-32, while similar models only need a few hundred milliseconds[1].

We introduce a linear masking solution to protect data privacy in VERISPLIT without relying on computationally expensive cryptographic primitives. It must be used with layer-by-layer offloading due to non-linearity across layers (e.g., activation functions). Our masking design is partly inspired by prior works [142, 208], commonly used in various cryptographic inference protocols, while we provide additional security analysis for floating-point masks (Section 5.6) instead of finite field.

Figure 5.2 presents the workflow of VERISPLIT's data privacy protection. During setup, the offloading device shares model parameters (e.g., weights W and bias b) with the worker and locally generates one-time masks (ϵ_i). In addition, the device computes the value $W\epsilon_i$ for each mask and stores it locally. It can generate many masks (and precompute $W\epsilon_i$'s) during idle times since each inference will consume a fresh one. This pre-processing step is commonly used by prior works [142, 208].

For a particular inference i, the device sends masked inputs ($x' = x + \epsilon_i$) instead of the real data. The worker executes inference and replies with the results ($y' = W(x + \epsilon_i) + b$). The worker never has access to users' private data. Once the device receives y', it can subtract the precomputed mask noise ($W\epsilon_i$) and obtain the true layer output $y = y' - W\epsilon_i = Wx_i + b$. This

[1]Baselines empirically measured by us since not presented in Delphi [142].

Figure 5.3: A simple but ineffective approach to adding masks for model confidentiality. This does not save any work for the device during inference, since it still needs to compute δx_i.

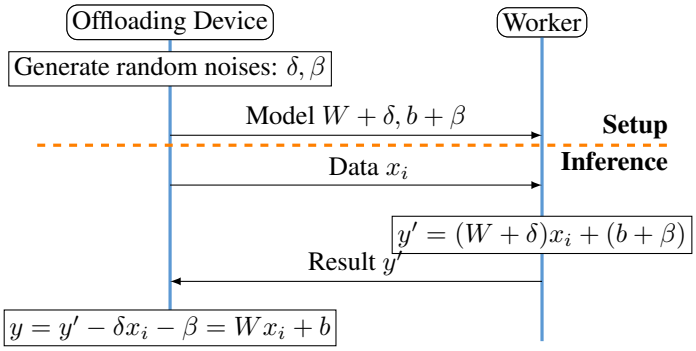
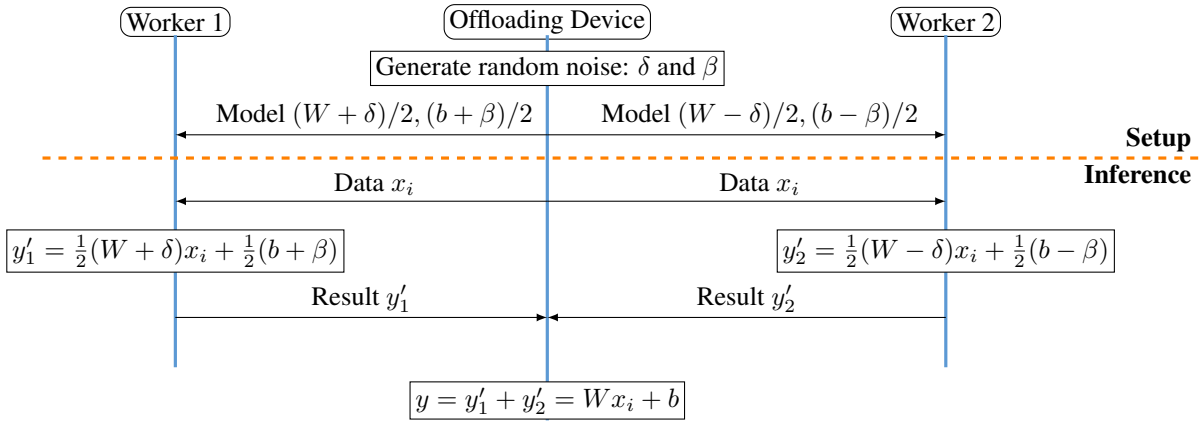


Figure 5.4: VERISPLIT's workflow for model confidentiality. During setup, the offloading device generates masking noises (δ, β) and shares modified model parameters with two non-colluding workers. For inferences, the device sends data and receives layer results (y'_1, y'_2). It can reconstruct the actual results by combining the two results.



solution offloads the expensive matrix multiplication computation to workers *during inferences*. As a compromise, the device must perform pre-computation ($W\epsilon_i$) asynchronously during setup or obtain pre-computed mask values from a trusted source (e.g., vendor backend can generate many masks and distribute them to edge devices).

5.4 Model Confidentiality

For model confidentiality, we propose a similar masking approach to that for data privacy. However, simply adding noise to these parameters does not work. Figure 5.3 illustrates this alternative and why it does not work for offloading since the device must still perform large matrix multiplication (δx_i in the last step) for each inference. Since this value is input-dependent, the device can not pre-compute it as in the case of data privacy. Offloading is actually worse in this case.

In VERISPLIT, we propose a solution leveraging multiple workers to provide model confidentiality. By sharing inference tasks with multiple workers, the device can avoid computing

matrix multiplications and leverage the performance benefits of more capable workers. As mentioned in our threat model (Section 5.2.2), we assume these workers are non-colluding, so they will not collectively compromise model confidentiality. This is a common threat model in many prior works [144, 177]. To achieve this, the offloading device can choose two workers from different vendors (or one personal device like a PC owned by the smart home user).

Figure 5.4 illustrates VERISPLIT's workflow for model confidentiality. The offloading device generates masks for model parameters (e.g., δ for weights and β for biases) and selects two local workers to share masked values. This is a one-time setup effort; subsequent inferences can use the same weights and biases. During the offload of an inference i, the device shares the input data with both workers and receives results y'_1 and y'_2. By combining these two results, the mask terms cancel out and provide the actual result of $Wx_i + b$. On the other hand, both workers only have access to masked weight parameters. Neither one can reconstruct original values without collusion. We elaborate on the security guarantee this method provides in Section 5.6.

The online inference process requires two workers to perform an equal workload with different model parameters. Both workers need to have capable hardware. In comparison, the offloading device now only needs to sum up intermediate results (y'_1 and y'_2), which is much faster than performing matrix multiplication operations locally.

5.5 Inference Integrity

This section explains how VERISPLIT ensures the integrity of the offloaded computation. We propose asynchronous verification to minimize the additional latency by removing it from the critical path of inferences. We also propose a partial verification mechanism to reduce the communication overhead.

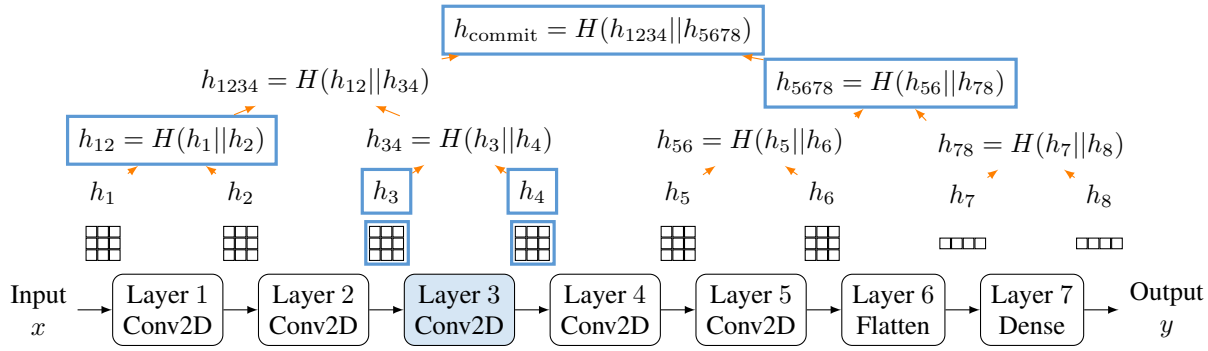
Layer-by-Layer Offloading

If the IoT device requires data privacy or model confidentiality, it has to offload the neural network's computations layer-by-layer because VERISPLIT's solutions only apply to linear operations (e.g., unsuitable for activation functions between layers). In the layer-by-layer offload scenario, verifying the integrity of inferences is fairly straightforward. During the layer-by-layer offloading, the device must transmit all intermediate results to workers. Therefore, the offloading device already has the complete inputs and outputs for each layer. To verify integrity, it can repeat the computation locally and check if the results match. As long as the device saves these results (at the expense of extra storage), it can choose when and how much to verify, retaining flexibility for asynchronous and partial verification. Alternatively, they can use VERISPLIT's Merkle-tree-based hash commits (explained in the next section) to further reduce local storage overhead, although our prototype currently does not implement this function for layer-by-layer offloading due to sufficient storage space.

Holistic Offloading

If the offloading device only requires integrity guarantees without data confidentiality or data privacy, it can perform holistic offloading — sending the entire model to the worker and directly

Figure 5.5: VERISPLIT's inference commitment design based on Merkle trees. This example network includes several 2D convolution layers, one flatten and one dense layer. During inference, the worker computes hash values of all layers' intermediate results (h_1–h_8) and reduces them into the final commit value h_{commit}. Assuming the verifier randomly decides to check results from layer 3 (blue shadow), the worker sends all values in blue boxes as its integrity proof. This commitment mechanism can be expanded to partially verify layer results (Section 5.5.2).



sharing the inputs to the model. The worker executes all model layers in one pass and returns the inference outputs. To verify the integrity of any layers, the device needs to access the inputs and outputs of the corresponding layer. Sending all intermediate results during inference can incur high communication overhead. To minimize inference latency, VERISPLIT separates verification from the inference process by proposing a commitment-based, asynchronous solution. The device can request intermediate data from previous inferences whenever it has free cycles. In addition, the device can request partial results from arbitrary layers (instead of full-layer results) if it only wants to check part of the results (to reduce computation and communication overhead).

5.5.1 Asynchronous Verification

The first key insight of VERISPLIT's integrity solutions is to move the verification process out of the critical path of the prediction itself while only requiring minimal storage overheads on the worker and verifier (i.e., the offloading device) sides. While prior works can support asynchronous verification (such as incorporating interactive proof techniques into neural network models [78]), they require modification to the model and affect its behavior and accuracy. On the other hand, works like Slalom [208] require transmitting intermediate results as part of the inference process, incurring high overhead. In addition, the verifier may need to store all intermediate results locally if they want to check them later. In contrast, VERISPLIT verifier only needs to store a single hash commit.

During inferences, the worker saves the input data while discarding all intermediate results, saving on storage. During verification, the verifier asks the worker for the original intermediate values (certain layers' inputs and outputs) and repeats the computation locally to check if the results match. To generate the selected layer results, the worker can repeat the inference a second time (since it stores the input data) and send back the requested values.

For integrity, VERISPLIT has to check whether the worker provides the same values during verification as those used in the original inferences. For example, a malicious or lazy worker

may only execute the correct model if an inference is selected for verification. To address this challenge, we design a Merkle tree-based inference commitment mechanism to capture all intermediate values generated during inference. The verifier receives this short commit value along with the inference results and stores it locally. During verification, the verifier can request arbitrary intermediate values from the worker, and the worker must generate proofs showing these values are included in the original commit.

In VERISPLIT, we designed a Merkle tree construction algorithm to enable flexible verification of arbitrary intermediate values across internal layers of a neural network model. Merkle tree [137] data structures have been widely used in many applications for generating short commit messages to provide verifiable integrity guarantees on system states [10, 63, 179, 226]. Recent works have also incorporated them into various machine learning applications [205, 243].

Figure 5.5 illustrates constructing the Merkle tree commit for an inference. For simplicity, we consider a 7-layer network with five 2D convolution layers, one flatten layer, and one dense layer for final prediction. The sizes of the matrices shown are for illustrative purposes only. During inference, the worker collects all layer inputs and outputs and computes their hash values (h_1–h_8). Then it generates the inference commit (h_{commit}) by recursively concatenating and hashing values in the Merkle tree. The verifier receives this commit along with the final prediction results at the end of the inference offloading process.

5.5.2 Partial Verification

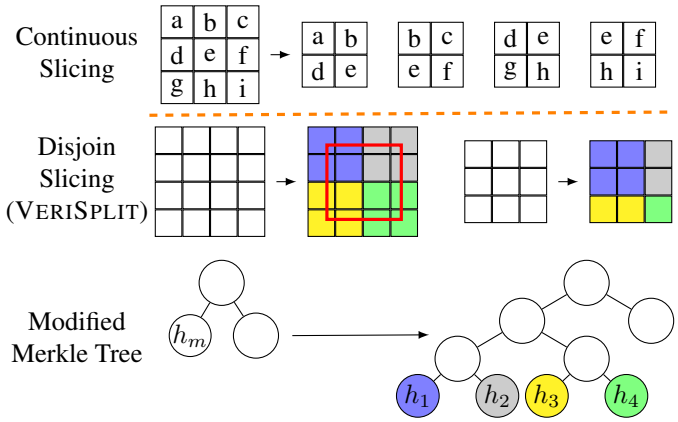
In this section, we explained how to verify the results of a single layer. However, the simple Merkle tree only supports checking each layer's results in their entirety. The inputs and outputs of one network layer can still be quite large, and verifying them requires significant computation. Instead, *partial* verification intuitively can be beneficial to reduce the overhead since the verifier would need to check a subset of values.

Directly reducing a simple Merkle tree for partial verification is inefficient. The worker needs to generate a proof with entire layer inputs and outputs (as the basis of the Merkle tree leaf nodes), even if the verifier only checks a small subset. If the verifier wants to verify a tiny portion of *every* layer, the worker needs to send *all* values! To solve this issue, we extend the original Merkle tree with fine-grained leaf nodes of smaller content sizes, reducing proof sizes and communication overhead. Figure 5.6 visualizes an example of VERISPLIT's slicing algorithm, which we will explain next.

Slicing Verify Units

It turns out that slicing layers' outputs into smaller units involves many trade-offs and deliberations. At one extreme, we can single out every element in the matrix into one unit. This provides fine granularity for partial verification since verifiers can choose arbitrary regions. However, it is impractical because of the large dimensions of layer results (e.g., certain VGG16 layers have outputs of 224x224x64). The more units we have, the larger the Merkle tree will be, and the worker needs to compute significantly more hashes to create the commit during inference time. Instead, we choose a fixed unit size, balancing the complexity of the Merkle tree and the flexibility of partial verification (Section 5.5.3 explains how to choose this value).

Figure 5.6: An example comparison between VERISPLIT's disjoint slicing mechanism and the continuous approach. We select the unit size to be 2x2. We color-code disjoint units and mark continuous units by their content. For continuous slicing, we only show slicing a 3x3 matrix due to limited spaces. The red box indicates one possible region verifiers may choose to check. After slicing, the Merkle tree adds extra leaf nodes, as shown in the transition from one hash h_m of the original matrix into 4 more leaves (h_1–h_4).



In VERISPLIT, we employ a disjoint slicing method to separate large matrices into smaller verify units. Figure 5.6 visualizes this slicing method compared with continuous alternatives. Disjoint slicing generates less number of units (we can not enumerate continuous units in a 4x4 matrix due to limited spaces), leading to less computation in the Merkle tree construction during inference. The compromise with disjoint slicing is that, during verification, the worker might need to include more neighboring units in the proof. For example, if the verifier selects the region in the red box, the worker needs to include all 4 slices since the exact region is not included in any individual Merkle tree leaf node.

Certain layer types cannot benefit from unit slicing to reduce the computation overhead. For example, dense layers with softmax activation are often used in the final prediction layer. To verify softmax, the verifier needs to gather all layer inputs and outputs, eliminating the benefit of partial verification. VERISPLIT thus reverts to full layer verification for those layers. Fortunately, many computationally demanding layers can benefit from partial verification (such as convolution layers, who take up over 90% of all computations in certain CNN models [35, 128, 131]).

5.5.3 Tunable Verification

VERISPLIT utilizes a configurable knob (verify ratio) to decide the size of individual verify units. If the verifier wants to enable partial verification as fine-grain as 1% (by setting this as the verify ratio), the worker slices each layer's results into at least 100 units. We can then calculate the unit size accordingly. The verifier can select a new value for every offloading request. This helps the verifier to adjust partial verification ratios based on the predicted arrival rates of new inferences. For scenarios where new inferences are rare, the verifier can set a higher ratio for individual offloads or maybe even perform full verification. On the other hand, if there are a large number of inferences consistently arriving, the verifier can set a smaller verify ratio to preserve resources

while retaining a high probability of catching malicious workers.

5.6 Security Analysis

In this section, we present a formal security analysis of VERISPLIT's solutions for VERISPLIT's three main goals (data privacy, model confidentiality, and inference integrity).

Data Privacy and Model Confidentiality. As a building block, we define a secure masking scheme between a client $\mathcal{C}$ and an adversary $\mathcal{A}$ that produces indistinguishable results for $\mathcal{A}$ when $\mathcal{C}$ randomly picks a value from choices (x_1, x_2) and applies a mask from range $[-\epsilon, \epsilon]$. We assume both choices for x and mask ranges are known to $\mathcal{A}$ and $\mathcal{C}$:

Definition 5.6.1 (ϵ-Masking Schemes)**.** An ϵ-masking scheme consists of the following steps:

- $x_i \leftarrow (x_1, x_2)$, $\epsilon_i \leftarrow [-\epsilon, \epsilon]$: $\mathcal{C}$ uniformly randomly selects a value x_i and a mask ϵ_i. $\mathcal{C}$ shares the masked value $(x_i + \epsilon_i)$ with $\mathcal{A}$.
- $\mathcal{A}$ guesses whether the selected value is x_1 or x_2 based on the observation of $(x_i + \epsilon_i)$.

Definition 5.6.1 defines an ϵ-bound masking scheme. Ideally, with perfect security, $\mathcal{A}$ should not gain any advantage in guessing x_i after observing the masked value of $(x_i + \epsilon_i)$. Therefore, we present the following definition of security:

Definition 5.6.2 (Secure Masking Schemes)**.** An ϵ-masking scheme is secure if it produces *indistinguishable* results such that $\mathcal{A}$ can only succeed at deciding the original value of x_i with up to 50% probability.

Furthermore, we define failures of such schemes as follows:

Definition 5.6.3 (Masking Scheme Failures)**.** A masking scheme fails when $\mathcal{A}$ can distinguish the original value of x_i based on observed $(x_i + \epsilon_i)$ with $> 50\%$ success rates.

Next, following these definitions, we can show that an ϵ-masking scheme can achieve perfect security by selecting a mask range of all possible numbers:

Theorem 5.6.1 (Perfectly Secure Masking Schemes)**.** If the masking scheme encodes all values in a finite field and selects the mask ϵ_i uniformly randomly from all field elements, then such a scheme is perfectly secure with 0% failure rates. The observed value $(x_i + \epsilon_i)$ will uniformly randomly distribute across the field elements regardless of which x_i is selected.

Theorem 5.6.1 describes the perfect masking scheme in which the adversary $\mathcal{A}$ can only observe uniformly random values. However, this scheme requires finite field operations, which are computationally expensive. Instead, we extend this masking mechanism to floating point numbers with bounded failure rates as follows:

Theorem 5.6.2 (Failure Rates for ϵ-Masking)**.** Assuming $x_\delta = |x_1 - x_2|$ and $\epsilon = k\,x_\delta$ $(k \geq 1)$. If $\mathcal{C}$ selects mask $\epsilon_i \in [-\epsilon, \epsilon]$ instead of using finite field elements, then $\mathcal{A}$, after observing a masked value of $(x_i + \epsilon_i)$, will be able to distinguish the original value between x_1 and x_2 with a probability of $\frac{2}{2k+1}$.

Theorem 5.6.2 describes the probability of an adversary defeating the indistinguishability property of a masking scheme (Definition 5.6.3). We can apply this failure rate calculation to our solutions for data privacy and model confidentiality with the following conservative estimates:

Corollary 5.6.2.1 (Estimated Failure Rates for Data Privacy and Model Confidentiality)**.** Assuming input data $x = (x_1, x_2,$
$\cdots, x_n)$, $x_\delta = \max(x) - \min(x)$, and $\epsilon = k\,x_\delta$ $(k \geq 1)$. Then $\forall x_i \in x$, we estimate the failure rate of applying ϵ-masking is $\frac{2}{2k+1}$. We expect the masking scheme to leak up to $\frac{2n}{2k+1}$ data points from x overall. Similar estimates also apply to model weights w and biases b.

Inference Integrity. First, we define the offloading scheme with partial verification as follows:

Definition 5.6.4 (α-Verified Offload)**.** Assuming client $\mathcal{C}$ wants to offload the computation of function $\mathcal{F} : \mathbb{R}^m \to \mathbb{R}^n$ to server $\mathcal{S}$ using the following steps:

- $\mathcal{C}$ shares input data $x = (x_1, x_2, \cdots, x_m)$ with $\mathcal{S}$.
- $\mathcal{S}$ computes $\hat{y} = (\hat{y}_1, \hat{y}_2, \cdots, \hat{y}_n)$ and shares $\hat{y}$ with $\mathcal{C}$.
- $\mathcal{C}$ selects a verification set $V \subseteq \hat{y}$ such that $|V| = \lceil \alpha\, n \rceil, \alpha \in (0, 1]$. $\mathcal{C}$ verifies that $\forall \hat{y}_i \in V, \hat{y}_i = y_i$, where $y = \mathcal{F}(x)$.

With this definition, we can calculate the failure rates of VERISPLIT's partial verification mechanism under multiple offloading iterations:

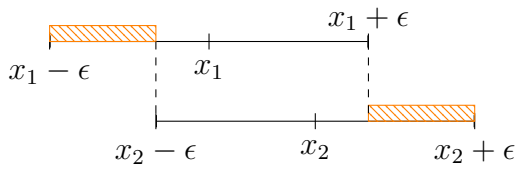
Theorem 5.6.3 (Failure Rates for α-Verification over k Iterations)**.** Assuming $\mathcal{C}$ and $\mathcal{S}$ conducts k rounds of α-verified offload. For each offload, $\mathcal{S}$ reports partially incorrect results, with ratio β. The probability of α-verification fails to detect incorrect results is $\left(\frac{\binom{n-b}{a}}{\binom{n}{a}}\right)^k$, where $a = \lceil \alpha\, n \rceil, b = \lceil \beta\, n \rceil$, $\binom{\cdot}{\cdot}$ is binomial coefficient (combination).

5.6.1 Proofs

Proof of Theorem 5.6.1. In this case, all numbers (e.g., x_1, x_2) are embedded in a finite field $\mathbb{Z}_p$. Assuming client $\mathcal{C}$ picks masking noise ϵ_i uniformly randomly from finite field $\mathbb{Z}_p$. After applying the mask, $\mathcal{C}$ shares the value of $(x_i + \epsilon_i) \in \mathbb{Z}_p$ with the adversary $\mathcal{A}$. $\mathcal{A}$ has no advantage in determining the original values of x_i since both $(x_1 + \epsilon_i)$ and $(x_2 + \epsilon_i)$ would appear uniformly random across all field elements in $\mathbb{Z}_p$. □

Proof of Theorem 5.6.2. Recall $x_\delta = |x_1 - x_2|$ and $\epsilon = k\,x_\delta$ $(k \geq 1)$. Assuming, without loss of generality, that $x_1 \leq x_2$. Figure 5.7 provides an example visualization. If the masked value $(x_i + \epsilon_i)$ falls between the regions $[x_1 - \epsilon, x_2 - \epsilon]$ and $[x_1 + \epsilon, x_2 + \epsilon]$ (orange regions in the figure), then $\mathcal{A}$ can distinguish whether the original value is x_1 or x_2. Otherwise, $\mathcal{A}$ has no advantage

Figure 5.7: Visualization of masking failure probabilities. If the observed value ($x_i + \epsilon_i$) falls within the orange dashed region, then adversary $\mathcal{A}$ can distinguish the original value.



over randomly guessing with a 50% success rate. Therefore, the failure rate of this ϵ-masking scheme is:

$$\text{failure rates} = \frac{(x_2 - \epsilon) - (x_1 - \epsilon) + (x_2 + \epsilon) - (x_1 + \epsilon)}{(x_2 + \epsilon) - (x_1 - \epsilon)} \tag{5.1}$$

$$= \frac{x_2 - x_1 + x_2 - x_1}{x_2 - x_1 + 2\epsilon} \tag{5.2}$$

$$= \frac{2x_\delta}{x_\delta + 2\epsilon} \tag{5.3}$$

$$= \frac{2}{2k+1} \tag{5.4}$$

□

Proof of Corollary 5.6.2.1. We assume there are n elements in data $x = (x_1, x_2, \cdots, x_n)$ (similar proof holds for weights w and biases b). For any individual data point $x_i \in x$, the *worst-case* estimated failure rates are bounded by the minimal and maximal values in x (hence, x_δ). Therefore, the upper bound of failure rates for *individual* data points is $\frac{2}{2k+1}$ (Theorem 5.6.2). Because ϵ-masking scheme is applied to each data point *independently*, the total expected number of failed points in x is $\frac{2n}{2k+1}$ in the worst case. □

Proof of Theorem 5.6.3. First, consider the failure probability of α-verification in a single round of offload. Let $a = \lceil \alpha\, n \rceil = |V|$ and $b = \lceil \beta\, n \rceil$. With α-verification, $\mathcal{C}$ fails to detect misbehaving $\mathcal{S}$ if and only if no points in V selected by $\mathcal{C}$ are incorrect. Therefore, by definition of combination, we calculate the failure probability of a single round offload with verification as:

$$\frac{\binom{n-b}{a}}{\binom{n}{a}}$$

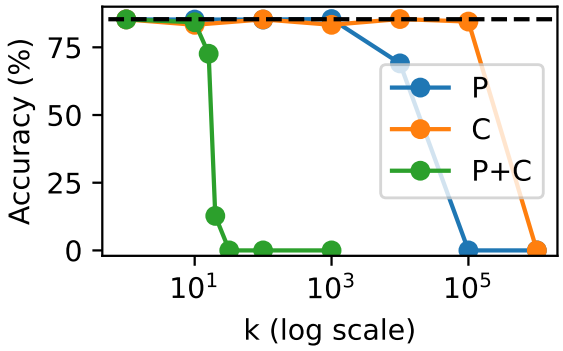
that is, the total number of ways of choosing a points from correct values in $\hat{y}$ divided by the total number of ways of choosing a points from all values in $\hat{y}$.

Similarly, for k iterations, $\mathcal{S}$ must consistently avoid detection by $\mathcal{C}$. Therefore, the overall failure rate for k iterations is the product of individual failure rates:

$$\left(\frac{\binom{n-b}{a}}{\binom{n}{a}} \right)^k$$

□

Figure 5.8: Inference accuracy of ImageNet data after applying different magnitudes of masks to data (for **P**rivacy) and model weights (for **C**onfidentiality). Neither option has noticeable impacts unless $k \geq 10^4$. However, combining both quickly degrades inference accuracy when $k > 10^1$.



5.7 Floating Point Errors

As explained in (Section 5.6), VERISPLIT utilizes floating point numbers to provide security guarantees without converting models into finite fields or using quantization. Unfortunately, floating point operations introduce new challenges due to precision issues and platform dependencies.

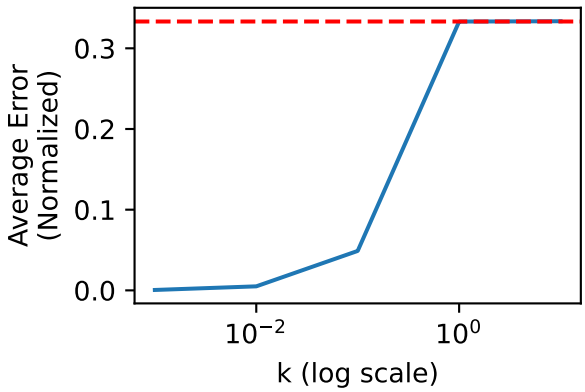
5.7.1 Mask Precision

With floating points masks, the key security parameter is k, the multiplier for the scale of masks relative to the range of input values (i.e., $\epsilon = kx_\delta$, Theorem 5.6.2). As k gets larger, the failure rate ($\frac{2}{2k+1}$) gets smaller and smaller. Ideally, we would make k as large as possible. However, we quickly run into a limitation of floating point operations: precision issues. If the mask value is too large, it is impossible to recover original results from the masked values (e.g., $y = y' - W\epsilon_i$ in Figure 5.2) due to a loss in precision ($y' \approx W\epsilon_i$). Therefore, we must consider practical limits on how large k can be.

Figure 5.8 measures the inference accuracy of the Vision Transformer model (ViT-L16) on the ImageNet validation dataset using different masking parameters. Applying masks for either data privacy or model confidentiality alone does not affect accuracy significantly for smaller k values ($< 10^4$). Eventually, masks become too large and imprecise for VERISPLIT to produce accurate inference results. Moreover, combining masking for both privacy and confidentiality significantly limits the maximum values of k before losing precision. We observe a drop of accuracy for k between 10^1 and 10^2. This is because of the multiplicative nature of masking (i.e., applying masks for both W and x and computing Wx).

Setting $k = 10$ may seem like a relatively weak security guarantee. However, our security analysis is based on the indistinguishability of two values (Theorem 5.6.2); hence, the failure rates are a conservative estimate for an attacker to recover original values based on masked observations. To understand the practical implications of these parameters, we design an empirical

Figure 5.9: Average recovery errors for attackers to predict original value x based on observed masked value x'. The red dashed line indicates randomly guessing values from $[x_{min}, x_{max}]$. The blue line represents guessing values from $[x' - \epsilon, x' + \epsilon]$ where ϵ depends on the masking scale multiplier k.



attack simulation to measure the recovery error rates under various k's. Specifically, we define a game where client $\mathcal{C}$ chooses values and adversary $\mathcal{A}$ tries to recover the original values with minimal errors (in L_1 distance):

Definition 5.7.1 (Attack on Floating Point Masks)**.** The attack on masked values is conducted as follows:

- $\mathcal{C}$ uniformly randomly selects a value $x_i \in [x_{min}, x_{max}]$ and a mask $\epsilon_i \in [-\epsilon, \epsilon]$, where $\epsilon = k(x_{max} - x_{min})$.
- $\mathcal{A}$ receives the value $x_m = (x_i + \epsilon_i)$ and the range $[x_{min}, x_{max}]$. $\mathcal{A}$ tries to guess x' with two strategies. The first one is to guess $x'_i \in [x_{min}, x_{max}]$ randomly as a baseline. The second one is to guess $x'_i \in [x_m - \epsilon, x_m + \epsilon]$ (clipped by x_{min}, x_{max}).
- We measure the average errors as $|x_i - x'_i|$.

Figure 5.9 presents the average errors empirically measured following Definition 5.7.1. We normalize errors by the range of the input $(x_{max} - x_{min})$. The red dashed line represents the baseline attack of uniformly randomly guessed values from $[x_{min}, x_{max}]$. The blue line represents the more sophisticated strategy of ϵ-bounded guesses. When k is small ($\leq 10^{-1}$), $\mathcal{A}$ has a strong advantage in recovering the original values with low errors. However, as k gets larger (> 1), the adversary has no optimal strategy compared to random guesses. These results demonstrate that our security analysis (Theorem 5.6.2) is very conservative, and k doesn't have to be too large to effectively protect masked data.

5.7.2 Cross-Platform Numerical Errors

Another floating-point challenge comes from the hardware platform differences in heterogeneous IoT devices. Specifically, for workers with GPUs, floating point operations can often cause small numerical errors due to imprecision and non-determinism in runtime libraries [153, 200, 200,

201]. These errors can not be reproduced on the offloading devices (CPUs or ARM microprocessors), leading to inconsistent results. Therefore, VERISPLIT can not guarantee the integrity and distinguish results from unfaithful inference executions. Therefore, we develop a solution to address the cross-platform result discrepancy with tolerances.

To select appropriate tolerances for cross-platform numerical errors, we introduce a *profiling* phase before the beginning of VERISPLIT offload. During profiling, we assume workers faithfully execute all inferences so we can correctly select tolerances. This process is commonly used in anomaly detection to select threshold and cut-off values [51, 141]. In future deployment, VERISPLIT users can collectively share an open-source database to look up appropriate tolerances for various hardware platforms without going through a worker-specific profiling process (and hence avoiding the trust-on-first-use issue for new workers).

During profiling, the device should select a small but representative dataset to offload to the worker. After receiving the worker's results, the device computes tolerances by measuring the maximum differences τ between offloaded and locally-computed results. Moreover, we multiply the tolerance by a factor of 4 (empirically chosen) as a relaxation to avoid mistakenly rejecting any offloading results unseen in the profile dataset. Finally, because each layer has different error ranges, we profile them individually and choose different tolerances.

Verifying integrity with tolerances introduces another challenge since VERISPLIT may fail to detect unfaithful workers. To better analyze this trade-off, we experiment with attacks on VERISPLIT's tolerance verification. We consider a malicious worker who performs Fast Gradient Method (FGM) attacks [80] aiming to covertly change prediction results by perturbing intermediate values while remaining below the acceptable tolerance. In short, an FGM attack maximizes the classification loss w.r.t. to the input by gradient ascend and ensures the attacked input is τ-away from the original input w.r.t to L_2 norms. The attacker can either change values from one *single-layer* or perform *multi-layer* attacks to modify layer i and all subsequent layers to cascade perturbations. We measure the number of predictions the attacker can change in the validation set of ImageNet [182] and use 10% of the dataset for profiling. We choose VGG16 models for experiments because VERISPLIT must offload them holistically, hence more vulnerable to attacks (i.e., multi-layer attacks are not feasible for layer-by-layer offloading). Our experiment results show that, with tolerances, an attacker can change the prediction results for 0.38% of the data (single-layer attacks) or 0.79% (multi-layer attacks) without being detected by the VERISPLIT verification algorithm.

Prior works [201] and our own experience finds that different batch sizes can affect the range of numerical errors. Switching from a batch size of 1 to 16 for profiling and inference significantly reduces the number of vulnerable points in the dataset (0.02% for single-layer attacks, 0.04% for multi-layer ones). Therefore, to further improve the integrity guarantee and to avoid false negatives, the device can pad inferences to use larger batch sizes in deployment.

5.8 Implementation

We implemented a prototype of VERISPLIT in 8728 lines of Python code. We implemented VERISPLIT's machine learning module based on TensorFlow [202] framework and communication serialization using Google Protocol Buffers [85]. We implemented the Merkle tree con-

struction and verification operation using `pymerkle` Python library [73]. In VERISPLIT we assume all workers are connected to wall power and hence do not consider battery constraints. We also explored alternative embedded devices with TPU accelerators (e.g., Pixels with Tensor chips [84] and Coral USB Edge TPU [43]). Unfortunately, TPUs (and their TensorFlow Lite SDK [203]) lack high precision floating point support and require model quantization [44] and quantization-aware training to provide competitive behavior [148]. This violates our practical goals of maintaining high flexibility with floating point operations (Section 5.2.1), so we did not pursue this route.

We implemented several performance optimizations, including model weight in-memory caching, multi-threading, and data compression. These optimizations bring mixed returns. For holistic offloading, they greatly reduced inference and verification latency. However, layer-by-layer offloading and memory-constrained devices experience negative results as these options exacerbate resource contention. For example, sending uncompressed data from memory-constrained Raspberry Pi has lower latency due to lower CPU utilization.

5.9 Evaluations

We evaluate the performance overhead of VERISPLIT with various security options compared to local execution. We present results from two types of machine learning models — Vision Transformers [58] and a convolution-based VGG16 [197]. Their architectural differences lead to different practical choices (layer-by-layer vs. holistic).

5.9.1 Setup

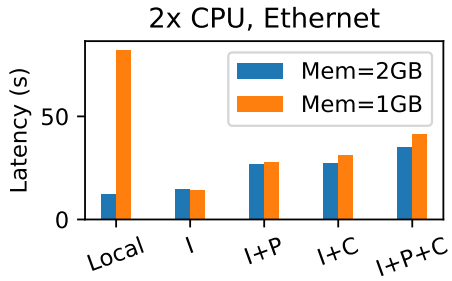
We use a Raspberry Pi 4 (RPi4) with 4 CPU cores (64 bit ARM Cortex-A72 cores, running at 1.5Ghz) and 4GB RAM as an example IoT device. To keep a representative setup, we manually restrict RPi's available resources (CPU, memory, network bandwidth) for the VERISPLIT application, considering 1) many IoT devices are less powerful (e.g., using micro-controllers [4, 5]) and 2) similar-equipped devices must share resources across many applications [30, 170]. We enforce hard limits on memory capacity and processors by modifying the Linux boot configuration. We allocate 4GB for memory swap space. The RPi connects to a local LAN either over Gigabit Ethernet or the 5GHz WiFi interfaces, depending on the evaluation.

For the local worker, we choose a gaming desktop (16 cores, 48GB RAM) with a GPU (RTX 3090, 24GB VRAM). The computer is connected to the LAN via Gigabit Ethernet. During our evaluation, the worker does not execute any resource-intensive workload and remains idle. To emulate multiple workers for model confidentiality, we create multiple VERISPLIT runtime on the same worker machine.

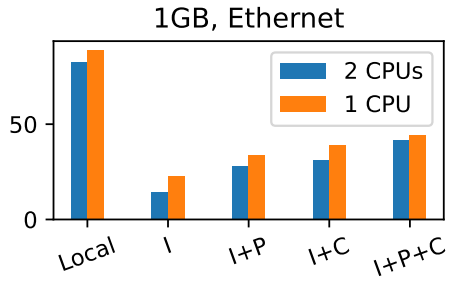
We selected the ImageNet [182] dataset for offloading workload and measured the average inference latency over 50 inferences. The offloading device conducts each inference in single batch to better capture real-time inference experience. We incorporate two representative machine learning models — a classic, convolution-based one (VGG16 [197]) and a more recent, attention-based family of Vision Transformers [58] (different sizes and configurations).

Figure 5.10: Average inference latency of Vision Transformer models (ViT-L16 unless noted otherwise) with guarantees: Integrity (I), Data Privacy (P), and Model Confidentiality (C), under different configurations for the offloading device. (a) Latency comparison of memory sizes. (b) Latency comparison of numbers of CPU cores. (c) Latency comparison of different network conditions. (d) Latency comparison of a larger model (ViT-H16) with different memory sizes.

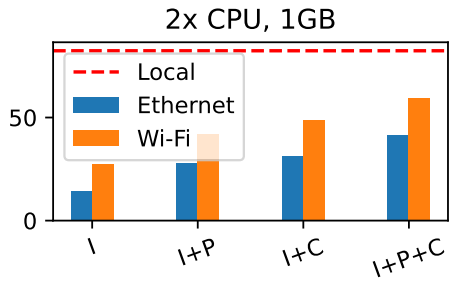
(a) Impact of Memory Size



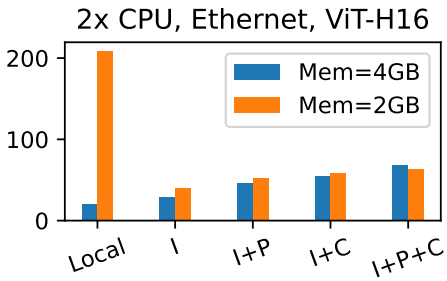
(b) Impact of CPU Cores



(c) Impact of the Network



(d) Memory Impact on Big Model



5.9.2 Vision Transformers

In this section, we present the evaluation results for offloading Vision Transformer (ViT) models with VERISPLIT. We employ a *layer-by-layer* offloading approach to demonstrate VERISPLIT's full capability of protecting data privacy, model confidentiality, and inference integrity.

For ViT models, we offload computationally-expensive dense layers in MLP to workers while keeping other layers (e.g., non-linear layers, attentions) on the device. This reduces computation demands on the device without incurring too much communication overhead.

Inference Latency. Figure 5.10 presents the average latency of model inferences with the ViT Large model (ViT-L16, with a batch size of 16) under various settings. As a baseline, we select local execution on the IoT device rather than comparison with first-party cloud offloading, which would require vendors to set up an additional cloud backend.

First, offloading with VERISPLIT outperforms local execution for devices with limited computation. Figure 5.10(a) shows that, for a device with 1GB memory, running ViT-L32 locally takes 82 seconds, while VERISPLIT offloading reduces latency by 49%–83%, depending on the set of options enabled. One might argue that the IoT device could increase memory to 2GB, and

CPU (cores)	Pre-Process (seconds/mask)	Inference (seconds)	Verification (seconds)
1	14.48	25.92	17
2	13.20	28.04	12.36

Table 5.1: Pre-process and verification time for VERISPLIT offloading with data privacy and integrity options, measured on a device with 1GB memory. Both overheads can be asynchronous: generate masks before inference starts and verify anytime after inference finishes.

the local execution would improve significantly. However, as machine learning models get more complex, larger models would quickly exceed device hardware capability. For example, even 2GB memory is insufficient for a larger vision transformer (ViT-H16, Figure 5.10(d)), which provides better accuracy. Looking forward, we believe VERISPLIT provides a practical solution to enable the execution of complex large models on resource-limited IoT devices with strong security and privacy guarantees.

In addition to memory size, we observe a slight impact from the number of CPU cores. Figure 5.10(b) presents the average inference latency on a device with 1GB memory and connected over Ethernet. Increasing CPU cores from 1x to 2x reduces latency of VERISPLIT offloading by up to 37% using the same set of options. In comparison, local execution only experiences 10% latency reduction due to it being memory constrained (i.e., not bounded by CPU).

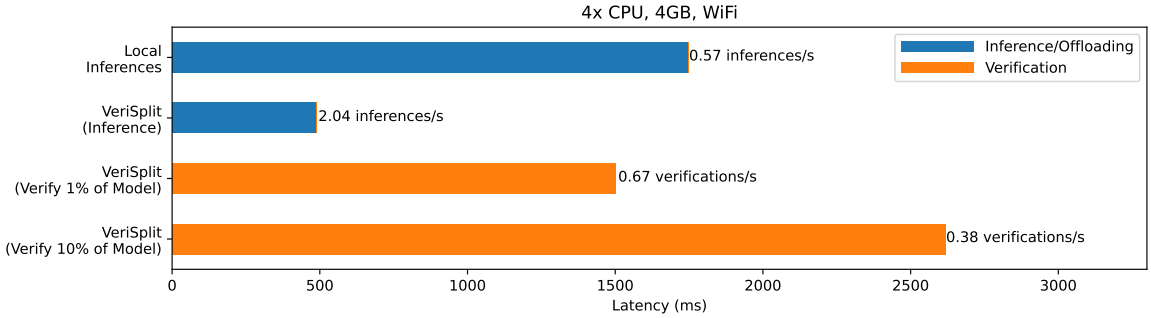
As expected, different network conditions significantly impact the performance of VERISPLIT offloading. Figure 5.10(c) compares the average inference latency between WiFi 5GHz and Ethernet connections for devices with 2 CPU cores and 1GB memory. The horizontal red dashed line indicates the local execution baseline. Switching the connection from Ethernet to WiFi raises the inference latency by up to 1.89x. However, it is important to note that offloading over WiFi with all options enabled still outperforms local inference baselines.

Additional Overhead. In addition to inference delays, different options of VERISPLIT incur additional overhead, such as pre-processing for data privacy and verification to ensure integrity. Table 5.1 presents these overheads compared to inference times. Here the device verifies *all* results without the need for partial verification. As expected, more CPU cores help reduce latency, although they both take considerable time. Unlike time-sensitive inferences, the device can pre-process multiple masks during idle times for future use and aggregate verification for many prior inferences altogether. Therefore, these delays should not impose too much performance penalty in a real-world deployment. Even if we combine both overheads together, VERISPLIT with data privacy and integrity verification still outperforms local inference baselines (Figure 5.10(b)).

5.9.3 VGG16

In addition to Vision Transformers, we evaluate VERISPLIT with a more traditional, convolution-based VGG16 model. Unlike transformers, VGG16 utilizes convolution layers extensively. Convolutions generate large amounts of intermediate values (layer inputs and outputs), as measured by prior research [111, 208]. Therefore, offloading VGG16 layer-by-layer incurs high commu-

Figure 5.11: Latency for offloading and verifying VGG16 model with VERISPLIT. The local execution baseline provides 0.57 inference per second, while integrity-enabled VERISPLIT can provide 2.04 inference/s. Notably, for VERISPLIT, verification can be performed asynchronously when the device is idle. Verification overhead includes the workers assembling the proofs, transmitting them over the network (WiFi in this setting), and the device recomputing the results and validating the proofs.



nication overhead, limiting the practicality of VERISPLIT and the performance benefits. Instead, we employ a *holistic* offloading approach to ensure inference integrity with VERISPLIT with minimal overhead. This implies VERISPLIT can provide inference integrity guarantees in this case.

Figure 5.11 presents the average latency for inferences with VGG16 models with integrity verification. We evaluate these tasks on a device with 4x CPU cores and 4GB memory. As a baseline, it takes 1746 ms to perform inferences locally. The device can use VERISPLIT to offload inferences to a worker with a GPU. This helps reduce inference latency to 491 ms. The most significant benefit of VERISPLIT offloading, in this case, is reduced CPU loads on the device (from 2.61 for local inferences to 0.63 for VERISPLIT with verification, not shown in the figure). If the device selects 1% of the intermediate values to verify, the verification overhead averages around 1503 ms, but this step can happen asynchronously. In addition, if the device wants to verify more values, it must spend more time collecting the additional intermediate results and validating proofs.

5.10 Limitations and Discussion

Applying VERISPLIT to Cloud Offloading. We designed VERISPLIT as a solution for secure and private offloading across local IoT devices. Therefore, we made a few design decisions based on network conditions and trade-offs between communication and computation overhead. We believe that the techniques we propose in VERISPLIT should also apply to certain cloud offloading applications depending on the network condition. For example, data-center networking may provide higher bandwidth and lower latency, making it ideal for VERISPLIT solutions. However, offloading over a WAN might introduce significantly higher latency and potentially metered bandwidth, and is thus less applicable for VERISPLIT.

Extending VERISPLIT to Additional Models. Incorporating new models into VERISPLIT requires a few steps. First, analyzing whether layer-by-layer or holistic offload is more suitable is important. If the new model involves layers with large intermediate values (e.g., convolution layers), offloading those layers might not be worthwhile due to the communication overhead. Therefore, one task is determining where each layer should be executed (locally or remotely) and how many layers should be offloaded altogether. Second, for new layer types, it is necessary to implement solutions for verifying partial results (instead of running the entire layer) to save on computation costs.

Resource Requirements for Efficient Verification. Although VERISPLIT enables devices to offload the bulk of computation to more capable workers, these devices still need sufficient resources (memory, processors) to perform masking and verification steps. If the device has insufficient resources, the performance will degrade: smaller memory sizes will trigger more memory swaps (if supported), and slower processor speeds will incur higher latency. If the device's resources are too limited, VERISPLIT offloading may not be suitable.

5.11 Summary

This chapter of the thesis presents the design and implementation of an efficient offloading solution, VERISPLIT, for local IoT devices to securely and privately utilize idle computing resources without managing infrastructure themselves. VERISPLIT addresses several practical challenges in protecting data privacy, model confidentiality, and inference integrity for resource-constrained IoT devices. We introduce masking mechanisms to protect private data and models by adding noises and an inference commitment mechanism based on Merkle trees to separate verification steps from real-time inferences. In addition, we addressed challenges in floating-point computation errors and precision issues.

Chapter 6

Conclusions

In this thesis, we have demonstrated that by applying various formal security analysis techniques and optimizing diligently, we can successfully split many functionalities from the list of IoT vendors' responsibilities to achieve a better landscape for IoT deployment's security, privacy, and cost-efficiency. More importantly, we have demonstrated the benefits of various formal security analysis techniques through their applications in the system design process.

With TEO, we showed that splitting the responsibility of managing application-level access controls can provide better privacy control for smart device users, especially in shared environments, which would be increasingly popular in future IoT deployments. For security analysis, we adopted an automated protocol verification tool ProVerif to examine design vulnerabilities and potential attacks violating security goals exhaustively. During the iterative process of verifying and revising, we addressed all security problems, streamlined protocol workflow, and reduced complexity.

With CAPTURE, we proposed that separating the role for managing security updates for third-party software libraries helps improve the overall IoT deployments' security by reducing the potential attack surfaces. Individual IoT vendors can delegate the responsibility of applying timely updates to centralized library management services. As for ensuring security properties, we employed a secure-by-construction approach, carefully considered potential attack points in the system, and analyzed the effectiveness of our isolation approaches in potentially compromised scenarios. Moreover, we identified network communication bottlenecks and incorporated proactive packet transmissions to minimize latency impacts.

With VERISPLIT, we presented that secure offloading machine learning inferences could alleviate IoT vendors' burdens and costs of managing first-party cloud services and infrastructures. We addressed the main challenges hindering vendors' adoptions, including data privacy, model confidentiality, and inference integrity, with novel designs of the offloading algorithm. To ensure security guarantees, we constructed formal proofs for quantitative analysis of security risks under different parameters. To improve the solution's practicality, we avoided heavyweight cryptographic operations but delivered a secure system design with lightweight alternatives.

6.1 Lessons Learned

Through conducting research for this thesis, we have obtained many insights and accumulated experiences in designing and implementing secure and practical IoT devices and systems by splitting functionalities. Upon reflection, we summarize and share several important lessons we learned in our journey.

Formal analyses provide strong security guarantees for new system designs. We employed various formal analysis techniques throughout the work presented in different thesis chapters. These approaches help us deliver strong security guarantees with our new system designs. In TEO, we leverage a symbolic protocol verification tool to ensure the security of our new protocol suite. In CAPTURE, we took a secure-by-construction approach and combined multiple security primitives into the new system design. We followed up with an in-depth analysis of how CAPTURE would prevent internal and external attackers. Finally, in VERISPLIT, we provided proofs of our new offloading algorithms' security arguments and statistic guarantees.

It is often challenging to justify the security guarantees when designing new systems. An accomplished faculty once told me that *"you kind of have to take a kitchen sink approach"* while asked about how to ensure any new system we develop is secure. His answer essentially means we should utilize all available tools and analyze as many security properties as possible. As a more principled approach, many formal analysis techniques we employed in this thesis could be easily adapted into other application domains and provide guidance on how to design new systems with strong security guarantees.

Splitting holistic designs demands performance-critical optimizations. This thesis presents many systems that separate certain functionalities from the traditional, holistic IoT device design. Without sufficient optimization, splitting those functionalities to third parties often incurs exorbitant performance overhead regarding network communication and computation. Sometimes, the optimizations can be as straightforward as buffering and batch processing — as shown in the CAPTURE's design of data buffers and message coalescing. In other cases, we may run into fundamental limitations of the splitting design and have to revise the systems and targeted use cases. For example, in VERISPLIT, the amounts of intermediate values in some neural networks are so large that it is impractical to offload them on a layer-by-layer basis. Therefore, we must relax the security guarantees for certain machine learning model types (e.g., VGG-16 vs. Vision Transformers).

Functionality splitting must address vendors' loss of agency and reduce adoption barriers. By delegating functionalities to third parties, IoT vendors no longer control the end-to-end software stack on the device. Therefore, a common theme of our works presented in this thesis is to alleviate vendors' adoption barriers and tackle their loss of agency. For example, centralizing library management on the CAPTURE hub takes away vendors' control over what library runtime their applications may be executed with. To avoid interrupting the normal operations of IoT applications, we conducted extensive analyses of library updating practices and the effectiveness of semantic versioning in preserving library stability in the real world. Based on

our insights, we proposed three library update strategies to balance between manageability and backward compatibility with existing applications. In addition, we also considered many other aspects to reduce vendors' adoption barriers. For example, throughout many of our projects, we proposed a number of integration approaches and often provided cross-platform library runtime to help existing applications integrate our new systems with low modifications.

6.2 Future Directions

More Capable and Secure Smart Home Hub Designs. Looking forward, many important functionalities still need improvements. As demonstrated in this thesis, splitting them from the monolithic design into dedicated services could provide many practical benefits and improve the landscape of future IoT deployment. A few prominent examples include providing stronger encryption for resource-limited devices [3] and more privacy-preserving user data processing with local hubs [107, 108].

As we split more and more functionalities from the holistic designs, the prevalent smart home hubs seem to be ideal candidates for taking over these new responsibilities. his future direction builds upon many important lessons learned and the limitations of systems presented in this thesis. For example, TEO-enabled trusted local hubs could help resource-constrained IoT devices better protect users' private data. We can extend the CAPTURE framework with more library supports to lower the adoption barrier for existing devices. In VERISPLIT, we can integrate with more machine learning models and inference workloads to make it widely available to common, existing workloads. One of the main contributions of this thesis is that we presented several approaches to designing practical hubs with strong security guarantees through a combination of formal analyses and optimizations. We believe these projects would provide valuable insights into designing the next-generation smart home hubs with minimal trust required from individual devices.

Beyond Smart Homes. Besides smart homes, many new applications can benefit from a more secure and practical offloading design, such as the emerging usage of vehicular networks [59], wearable computing [21], and virtual reality devices [195]. Many approaches and solutions proposed in this thesis are not limited to smart home applications. For example, VERISPLIT introduces a secure and verifiable offloading algorithm for neural network models, and it can be easily adapted into applications beyond smart home devices. One of the main challenges is that new use cases may not have the same network characteristics as smart homes — high bandwidth over unmetered networks — and hence, many targeted neural network models would require further investigation on which layers to offload and when to do it.

We can extend works in this thesis to more and more use cases. Although they may require small modifications, many components, such as the security analysis and the common optimization techniques, can be reused. Therefore, we believe it would be a promising direction to apply ideas from this thesis to other emerging new use cases to provide secure and practical offloading solutions.

Appendix A

Formal Modeling Code for TEO Protocol Verification

```
free io:channel.

(*******************************************************)
(* HMAC *)
(*******************************************************)

type mac_key.
fun mk2b(mac_key):bitstring [data,typeConverter].
fun b2mk(bitstring):mac_key [data,typeConverter].


(*******************************************************)
(* Public Key Signatures *)
(*******************************************************)

type privkey.
type pubkey.
fun pk(privkey): pubkey.
const NoPubKey:pubkey.

fun pubkey2b(pubkey):bitstring [data, typeConverter].
fun b2pubkey(bitstring):pubkey [data, typeConverter].


(*******************************************************)
(* Function mapping between Libsodium and ProVerif APIs *)
(*******************************************************)

type nonce.
fun nonce2b(nonce):bitstring [data, typeConverter].
fun b2nonce(bitstring):nonce [data, typeConverter].

(*
  Secret key encryption
```

```
*)

(* Debug helper function to test key secrecy, not real crypto APIs *)
fun debug_enc(bitstring, bitstring):bitstring.
fun debug_dec(bitstring, bitstring):bitstring
reduc forall k,msg:bitstring;
  debug_dec(k, debug_enc(k, msg)) = msg.

(* Message authentication HMAC *)
fun crypto_auth(mac_key, bitstring):bitstring.

fun crypto_auth_verify(mac_key, bitstring, bitstring):bool
reduc forall k:mac_key, msg:bitstring;
  crypto_auth_verify(k, msg, crypto_auth(k, msg)) = true
otherwise forall k:mac_key, x,y:bitstring;
  crypto_auth_verify(k, x, y) = false.

(* Encrypting file streams *)
type secretstream_key.
type secretstream_header.

fun crypto_secretstream_encrypt(secretstream_key, secretstream_header,
   bitstring):bitstring.

fun crypto_secretstream_decrypt(secretstream_key, secretstream_header,
   bitstring):bitstring
reduc forall k:secretstream_key, h:secretstream_header, msg:bitstring;
  crypto_secretstream_decrypt(k, h, crypto_secretstream_encrypt(k, h, msg))
     = msg.

(*
  Public key cryptography
*)

(* AEAD operations *)
(* encryption(msg, nonce, receiverPubkey, senderPrivkey) *)
fun crypto_box_easy(bitstring, nonce, pubkey, privkey):bitstring.

(* decryption(ciphertext, nonce, senderPubkey, receiverPrivkey) *)
fun crypto_box_open_easy(bitstring, nonce, pubkey, privkey):bitstring
reduc forall msg:bitstring, n:nonce, senderPrivkey:privkey, receiverPrivkey:
   privkey;
  crypto_box_open_easy(crypto_box_easy(msg, n, pk(receiverPrivkey),
     senderPrivkey),
                       n, pk(senderPrivkey), receiverPrivkey) = msg.


(* Sealed boxes for encrypting with receiver's public key *)
fun crypto_box_seal(bitstring, pubkey):bitstring.

fun crypto_box_seal_open(bitstring, privkey):bitstring
reduc forall msg:bitstring, k:privkey;
```

```
  crypto_box_seal_open(crypto_box_seal(msg, pk(k)), k) = msg.


(* EdDSA signature algorithm *)
fun crypto_sign(privkey, bitstring): bitstring.

(* verify(signerPubkey, message, hashValue) *)
fun crypto_sign_verify(pubkey, bitstring, bitstring):bool
reduc forall k:privkey, x:bitstring;
  crypto_sign_verify(pk(k), x, crypto_sign(k, x)) = true
otherwise forall k:pubkey, x,y:bitstring;
  crypto_sign_verify(k, x, y) = false.


(*
  Sieve cryptography functions
*)

type sieve_key.
type rekey_token.

fun sieve_key2b(sieve_key):bitstring [data, typeConverter].
fun b2sieve_key(bitstring):sieve_key [data, typeConverter].

fun sieve_enc(sieve_key, nonce, bitstring):bitstring.

fun sieve_dec(sieve_key, nonce, bitstring, bitstring):bitstring
reduc forall k:sieve_key, n:nonce, hint,msg:bitstring;
  sieve_dec(k, n, hint, sieve_enc(k, n, msg)) = msg.

fun sieve_rekey_token(sieve_key, sieve_key):rekey_token.

fun sieve_rekey(bitstring, rekey_token):bitstring
reduc forall k:sieve_key, kNew:sieve_key, n:nonce, msg:bitstring;
  sieve_rekey(sieve_enc(k, n, msg), sieve_rekey_token(k, kNew)) = sieve_enc(
     kNew, n, msg).

(*
  Shamir secret sharing

  Split secrets into variable number of parts.
*)

$template ${UID}
fun split_key_shares_${UID}(secretstream_key):bitstring.
$endtemplate

fun assemble_key_shares($expand{bitstring}$with{,}$by{${UID}}):
   secretstream_key
reduc forall d:secretstream_key;
  assemble_key_shares($expand{split_key_shares_${UID}(d)}$with{,}) = d.
```

```
(*******************************************************)
(* Security property prooofs *)
(*******************************************************)

(*********** Debug ***********)
(*
  Helper variable to test out secret keys.

  Expected value: query returns true so attacker doesn't know the secret.
*)
free debugSecret:bitstring [private].
query attacker(debugSecret).

(*********** Initialization ***********)
event InitializationAdminAcquireDevice(pubkey, pubkey).
event InitializationDeviceAcceptAdmin(pubkey, pubkey).

(*
  Reachability queries:
    If the query returns "RESULT not event() is true", this means the event
       is
    not reachable. Thus the corresponding code belongs to an unreachable
       branch.

    We can use conjunction to ensure multiple events are reachable.

  Expected value: Query not (event() && ... && event()) is false.
*)
query admin, device:pubkey;
  event(InitializationAdminAcquireDevice(admin, device))
  && event(InitializationDeviceAcceptAdmin(admin, device))
  .

(*
  Property: when device accepts a new admin manager, the admin must have
  initiated a request first.
 *)
query admin, device:pubkey;
  inj-event(InitializationDeviceAcceptAdmin(admin, device))
  ==> inj-event(InitializationAdminAcquireDevice(admin, device))
  .


(*********** AcquirePreAuthToken ***********)

event AcquirePreAuthTokenAdminGrant(pubkey, pubkey, bitstring).
event AcquirePreAuthTokenUserReceive(pubkey, pubkey, bitstring).

query admin, user:pubkey, token:bitstring;
  event(AcquirePreAuthTokenAdminGrant(admin, user, token))
  && event(AcquirePreAuthTokenUserReceive(admin, user, token))
  .
```

```
(*
  Property: When user receives a token, this is a valid token issued by
  the official system admin.
*)
query admin, user:pubkey, token:bitstring;
  inj-event(AcquirePreAuthTokenUserReceive(admin, user, token)) ==> inj-
      event(AcquirePreAuthTokenAdminGrant(admin, user, token)).


(*********** ClaimDevice ***********)

event ClaimDeviceDeviceAcceptUser(pubkey, pubkey, pubkey, bitstring).
event ClaimDeviceUserFinishDevice(pubkey, pubkey, pubkey).

query user, device, admin:pubkey, preAuthToken:bitstring;
  event(ClaimDeviceDeviceAcceptUser(user, device, admin, preAuthToken))
  && event(ClaimDeviceUserFinishDevice(user, device, admin))
  .

(*
  Property: if a device accepts a new user, the user must have initiated it
  and the user must possess a valid pre-auth token issued by admin.
*)
query user, device, admin:pubkey, preAuthToken:bitstring;
  inj-event(ClaimDeviceUserFinishDevice(user, device, admin))
  ==> (
    inj-event(ClaimDeviceDeviceAcceptUser(user, device, admin, preAuthToken)
        )
    ==> (
      inj-event(AcquirePreAuthTokenAdminGrant(admin, user, preAuthToken))
    )
  )
  .


(*********** DataStore ***********)

free privateUserData:bitstring [private].
query attacker(privateUserData).

$template ${UID}
(* user, device, sessionID *)
event DataStoreUserReceiveNotification${UID}(pubkey, pubkey, bitstring).
$endtemplate
(* users, device, sessionID *)
event DataStoreDeviceFinish($expand{pubkey}$with{,}$by{${UID}}, pubkey,
   bitstring).
(* users, UUID, content *)
event DataStoreAssignOwnership($expand{pubkey}$with{,}$by{${UID}}, bitstring
   , bitstring).
```

```
(* Basic reachability queries *)
query $expand{user${UID}}$with{,}, device:pubkey, sessionID:bitstring;
  $expand{event(DataStoreUserReceiveNotification${UID}(user${UID}, device,
      sessionID))}$with{ && }
  && event(DataStoreDeviceFinish($expand{user${UID}}$with{,}, device,
      sessionID))
  .

query $expand{user${UID}}$with{,}, device:pubkey, sessionID:bitstring;
  $expand{inj-event(DataStoreUserReceiveNotification${UID}(user${UID},
      device, sessionID))}$with{ && }
  ==> inj-event(DataStoreDeviceFinish($expand{user${UID}}$with{,}, device,
      sessionID))
  .




(*********** DataAccess ***********)
(* accessor, UUID *)
event DataAccessAccessorDecrypt(pubkey, bitstring).
$template ${UID}
(* user, accessor, UUID *)
event DataAccessUserGrantPermission${UID}(pubkey, pubkey, bitstring).
$endtemplate


query $expand{user${UID}}$with{,}, accessor:pubkey, metaDataUUID:bitstring;
  event(DataAccessAccessorDecrypt(accessor, privateUserData))
  &&
  $expand{event(DataAccessUserGrantPermission${UID}(user${UID}, accessor,
      metaDataUUID))}$with{ && }
  .


query $expand{user${UID}}$with{,}, accessor:pubkey, metaDataUUID:bitstring;
  inj-event(DataAccessAccessorDecrypt(accessor, privateUserData))
  ==> (
    $expand{inj-event(DataAccessUserGrantPermission${UID}(user${UID},
        accessor, metaDataUUID))}$with{ && }
  )
  .

(*********** DataReencryption ***********)
$template ${UID}
event DataReencryptionUserRequestKeySwitch${UID}(bitstring,nonce,nonce).
event DataReencryptionUserSwitchKey${UID}(bitstring,nonce,nonce).
event DataReencryptionStorageSwitchKey${UID}(bitstring,nonce,nonce).
event DataReencryptionAccessorTryDecryptOldKey${UID}(bitstring,bitstring).
event DataReencryptionAccessorSucceedDecryptOldKey${UID}(bitstring,bitstring
   ).
```

```
query metaDataUUID, sieveDataUUID:bitstring, userNonce,storeNonce:nonce;
  event(DataReencryptionUserRequestKeySwitch${UID}(sieveDataUUID, userNonce,
      storeNonce))
  && event(DataReencryptionUserSwitchKey${UID}(sieveDataUUID, userNonce,
      storeNonce))
  && event(DataReencryptionStorageSwitchKey${UID}(sieveDataUUID, userNonce,
      storeNonce))
  && event(DataReencryptionAccessorTryDecryptOldKey${UID}(metaDataUUID,
      sieveDataUUID))
  .

(*
  This event is unreachable if re-encrypt succeeds;
  otherwise, accessor can decrypt the data with old keys, make it reachable.
*)
query metaDataUUID, sieveDataUUID:bitstring;
  event(DataReencryptionAccessorSucceedDecryptOldKey${UID}(metaDataUUID,
      sieveDataUUID))
  .


(*
  If storage provider switches key, user must have requested it already.
*)
query metaDataUUID, sieveDataUUID:bitstring, userNonce,storeNonce:nonce;
  inj-event(DataReencryptionUserSwitchKey${UID}(sieveDataUUID, userNonce,
      storeNonce))
  ==> (
    inj-event(DataReencryptionStorageSwitchKey${UID}(sieveDataUUID,
        userNonce,storeNonce))
    ==> (
      inj-event(DataReencryptionUserRequestKeySwitch${UID}(sieveDataUUID,
          userNonce,storeNonce))
    )
  )
.

$endtemplate


(*****************************************************)
(* Protocol Message Header *)
(*****************************************************)
(* Placeholder for switching group mode in Device Claim request *)
$template ${UID}
const GROUP_MODE_${UID} : bitstring.
$endtemplate

const INITIALIZATION_REQUEST
  , INITIALIZATION_DEVICE_INFO
  , INITIALIZATION_ADMIN_REPLY
```

```
  , ACQUIRE_PRE_AUTH_TOKEN_REQUEST
  , ACQUIRE_PRE_AUTH_TOKEN_RESPONSE

  , CLAIM_DEVICE_DISCOVERY
  , CLAIM_DEVICE_DISCOVERY_RESPONSE
  , CLAIM_DEVICE_REQUEST
  , CLAIM_DEVICE_RESPONSE

  , DATA_STORE_SIEVE_CRED_REQUEST
  , DATA_STORE_SIEVE_CRED_RESPONSE
  , DATA_STORE_UPLOAD
  , DATA_STORE_UPLOAD_NOTIFICATION_1
  , DATA_STORE_UPLOAD_NOTIFICATION_2
  , DATA_STORE_DOWNLOAD_REQUEST
  , DATA_STORE_DOWNLOAD_RESPONSE_1
  , DATA_STORE_DOWNLOAD_RESPONSE_1_ACK

  , DATA_ACCESS_FETCH
  , DATA_ACCESS_RESPONSE

  , DATA_REENCRYPTION_PRE_REQUEST
  , DATA_REENCRYPTION_PRE_RESPONSE
  , DATA_REENCRYPTION_REQUEST
  , DATA_REENCRYPTION_RESPONSE
  :bitstring.


const ADMIN_SIGN_TYPE_DEVICE_PROOF
  , ADMIN_SIGN_TYPE_PRE_AUTH_TOKEN
  , USER_SIGN_TYPE_REKEY_TOKEN
  :bitstring.

const CIPHER_INITIALIZATION_REQUEST
  , CIPHER_INITIALIZATION_DEVICE_INFO
  , CIPHER_INITIALIZATION_ADMIN_REPLY

  , CIPHER_ACQUIRE_PRE_AUTH_TOKEN_RESPONSE

  , CIPHER_CLAIM_DEVICE_REQUEST
  , CIPHER_CLAIM_DEVICE_RESPONSE

  , CIPHER_DATA_STORE_SIEVE_CRED_REQUEST
  , CIPHER_DATA_STORE_SIEVE_CRED_RESPONSE
  , CIPHER_DATA_STORE_UPLOAD_NOTIFICATION_1
  , CIPHER_DATA_STORE_UPLOAD_NOTIFICATION_2

  , CIPHER_DATA_ACCESS_RESPONSE

  , CIPHER_DATA_REENCRYPTION_PRE_REQUEST
  , CIPHER_DATA_REENCRYPTION_PRE_RESPONSE
  , CIPHER_DATA_REENCRYPTION_REQUEST
  :bitstring.
```

```
(*****************************************************)
(* Protocol Modules *)
(*****************************************************)

(*********** Initialization ***********)

(* device pubkey <--> device manager/accepted admin pubkey *)
table deviceAdminKeyTable(pubkey, pubkey).
table deviceValidProofTable(pubkey, bitstring).

let AdminInitialization(setupKey:secretstream_key, adminPrivkey:privkey) =
  new setupHeader:secretstream_header;
  new adminChallenge:bitstring;
  let adminPubkey = pk(adminPrivkey) in

  out(io, (INITIALIZATION_REQUEST,
           setupHeader,
           crypto_secretstream_encrypt(
             setupKey,
             setupHeader,
             (CIPHER_INITIALIZATION_REQUEST, adminPubkey, adminChallenge)
           )));

  in(io, (=INITIALIZATION_DEVICE_INFO, ciphertext:bitstring));

  let (=CIPHER_INITIALIZATION_DEVICE_INFO, =adminChallenge, deviceChallenge:
     bitstring, devicePubkey:pubkey) = crypto_box_seal_open(ciphertext,
     adminPrivkey) in

  event InitializationAdminAcquireDevice(adminPubkey, devicePubkey);

  let validDeviceProof = crypto_sign(adminPrivkey, (
     ADMIN_SIGN_TYPE_DEVICE_PROOF, devicePubkey)) in

  new msgNonce:nonce;
  out(io, (INITIALIZATION_ADMIN_REPLY,
           msgNonce,
           crypto_box_easy(
             (CIPHER_INITIALIZATION_ADMIN_REPLY, deviceChallenge,
                validDeviceProof),
             msgNonce,
             devicePubkey,
             adminPrivkey
           )));

  out(io, debug_enc((setupKey, setupHeader), debugSecret)).


let DeviceInitialization(setupKey:secretstream_key, devicePrivkey:privkey) =

  let devicePubkey = pk(devicePrivkey) in
```

```
  in(io, (=INITIALIZATION_REQUEST, setupHeader:secretstream_header,
      ciphertext:bitstring));
  new deviceChallenge:bitstring;

  let (=CIPHER_INITIALIZATION_REQUEST, adminPubkey:pubkey, adminChallenge:
      bitstring) = crypto_secretstream_decrypt(setupKey, setupHeader,
      ciphertext) in

  out(io, (INITIALIZATION_DEVICE_INFO,
          crypto_box_seal((CIPHER_INITIALIZATION_DEVICE_INFO, adminChallenge
              , deviceChallenge, devicePubkey),
            adminPubkey
          )));

  in(io, (=INITIALIZATION_ADMIN_REPLY, msgNonce:nonce, msgEncrypted:
      bitstring));

  let (=CIPHER_INITIALIZATION_ADMIN_REPLY, =deviceChallenge,
      validDeviceProof:bitstring) = crypto_box_open_easy(msgEncrypted,
      msgNonce, adminPubkey, devicePrivkey) in

  insert deviceAdminKeyTable(devicePubkey, adminPubkey);
  insert deviceValidProofTable(devicePubkey, validDeviceProof);
  event InitializationDeviceAcceptAdmin(adminPubkey, devicePubkey);
  out(io, debug_enc((setupKey, setupHeader), debugSecret)).


(*********** AcquirePreAuthToken ***********)

table userPreAuthTokenTable(pubkey, bitstring).

let AdminAcquirePreAuthToken(adminPrivkey:privkey) =
  in(io, (=ACQUIRE_PRE_AUTH_TOKEN_REQUEST, userPubkey:pubkey));

  new boxNonce:nonce;
  let token = crypto_sign(adminPrivkey, (ADMIN_SIGN_TYPE_PRE_AUTH_TOKEN,
      userPubkey)) in
  event AcquirePreAuthTokenAdminGrant(pk(adminPrivkey), userPubkey, token);
  out(io, (ACQUIRE_PRE_AUTH_TOKEN_RESPONSE, boxNonce, crypto_box_easy((
      CIPHER_ACQUIRE_PRE_AUTH_TOKEN_RESPONSE, token), boxNonce, userPubkey,
      adminPrivkey))).

let UserAcquirePreAuthToken(userPrivkey:privkey, adminPubkey:pubkey) =
  let userPubkey = pk(userPrivkey) in
  out(io, (ACQUIRE_PRE_AUTH_TOKEN_REQUEST, userPubkey));

  in(io, (=ACQUIRE_PRE_AUTH_TOKEN_RESPONSE, boxNonce:nonce, ciphertext:
      bitstring));
  let (=CIPHER_ACQUIRE_PRE_AUTH_TOKEN_RESPONSE, token:bitstring) =
      crypto_box_open_easy(ciphertext, boxNonce, adminPubkey, userPrivkey) in
  insert userPreAuthTokenTable(userPubkey, token);
```

```
  event AcquirePreAuthTokenUserReceive(adminPubkey, userPubkey, token).


(*********** ClaimDevice ***********)


$template ${UID}
table userControlDeviceTable${UID}(pubkey, pubkey).

let UserClaimDevice${UID}(userPrivkey:privkey, adminPubkey:pubkey) =
  let userPubkey = pk(userPrivkey) in
  get userPreAuthTokenTable(=userPubkey, token) in

  out(io, (CLAIM_DEVICE_DISCOVERY, userPubkey));
  in(io, (=CLAIM_DEVICE_DISCOVERY_RESPONSE, devicePubkey:pubkey,
      validDeviceProof:bitstring));

  if crypto_sign_verify(adminPubkey, (ADMIN_SIGN_TYPE_DEVICE_PROOF,
      devicePubkey), validDeviceProof) then
    new sessionNonce:nonce;
    new userChallenge:bitstring;

    out(io, (CLAIM_DEVICE_REQUEST,
            sessionNonce,
            crypto_box_easy((CIPHER_CLAIM_DEVICE_REQUEST, GROUP_MODE_${UID},
                token, userChallenge), sessionNonce, devicePubkey,
               userPrivkey)
            ));
    in(io, (=CLAIM_DEVICE_RESPONSE, status:bool, responseNonce:nonce,
       challengeResponseEncrypted:bitstring));
    if status then
      let (=CIPHER_CLAIM_DEVICE_RESPONSE, challengeDecrypted:bitstring) =
         crypto_box_open_easy(challengeResponseEncrypted, responseNonce,
         devicePubkey, userPrivkey) in
      if challengeDecrypted = userChallenge then

        insert userControlDeviceTable${UID}(userPubkey, devicePubkey);

        event ClaimDeviceUserFinishDevice(userPubkey, devicePubkey,
           adminPubkey).
$endtemplate


$template ${UID}
table deviceOwnerKeyTable${UID}(pubkey, pubkey).

let DeviceClaimDevice${UID}(devicePrivkey:privkey) =
  let devicePubkey = pk(devicePrivkey) in
  get deviceValidProofTable(=devicePubkey, deviceValidProof) in
  get deviceAdminKeyTable(=devicePubkey, adminPubkey) in

  in(io, (=CLAIM_DEVICE_DISCOVERY, userPubkey:pubkey));
```

```
  out(io, (CLAIM_DEVICE_DISCOVERY_RESPONSE, devicePubkey, deviceValidProof))
      ;

  in(io, (=CLAIM_DEVICE_REQUEST,
          sessionNonce:nonce,
          msgEncrypted:bitstring
         ));
  let (=CIPHER_CLAIM_DEVICE_REQUEST, =GROUP_MODE_${UID}, token:bitstring,
      userChallenge:bitstring) = crypto_box_open_easy(msgEncrypted,
      sessionNonce, userPubkey, devicePrivkey) in
  if crypto_sign_verify(adminPubkey, (ADMIN_SIGN_TYPE_PRE_AUTH_TOKEN,
      userPubkey), token) then
    event ClaimDeviceDeviceAcceptUser(userPubkey, devicePubkey, adminPubkey,
        token);

    insert deviceOwnerKeyTable${UID}(devicePubkey, userPubkey);

    new responseNonce:nonce;
    out(io, (CLAIM_DEVICE_RESPONSE, true, responseNonce, crypto_box_easy((
        CIPHER_CLAIM_DEVICE_RESPONSE, userChallenge), responseNonce,
        userPubkey, devicePrivkey)));
    0.

$endtemplate

(*********** DataStore ***********)

(* storageTableV0(UUID, ownerPubkey, content) *)
const NULLPK: pubkey.
table storageTableV0(bitstring, $expand{pubkey}$with{,}$by{${UID}},
    bitstring).
$template ${UID}
(* Work around for rekey since tables are append-only *)
table storageTableV${UID}
  (bitstring, $expand{pubkey}$with{,}$by{${UID}}, bitstring).
$endtemplate

let DeviceDataStore(devicePrivkey:privkey) =
  let devicePubkey = pk(devicePrivkey) in
$template ${UID}
  get deviceOwnerKeyTable${UID}(=devicePubkey, ownerPubkey${UID}) in
$endtemplate

  (* Session creds are used for communication b/w device and owner *)
  new sessionID:bitstring;
$template ${UID}
  new sessionNonce${UID}:nonce;
$endtemplate

  (* Data creds are used for later accessor decryption *)
  new dataKey:secretstream_key;
```

```
  new dataHeader:secretstream_header;

  (* Generate list of UUIDs *)
  new dataUUID:bitstring;
$template ${UID}
  new sieveDataUUID${UID}:bitstring;
$endtemplate
  new metaDataUUID:bitstring;

  (* Encrypte data locally *)
  let dataEncrypted = crypto_secretstream_encrypt(dataKey, dataHeader,
     privateUserData) in

  out(io, (DATA_STORE_UPLOAD, dataUUID, $expand{ownerPubkey${UID}}$with{,},
     dataEncrypted));

  (* Fetch Sieve creds to retrieve *)
$template ${UID}
  out(io, (DATA_STORE_SIEVE_CRED_REQUEST,
           devicePubkey,
           sessionNonce${UID},
           ownerPubkey${UID},
           crypto_box_easy((CIPHER_DATA_STORE_SIEVE_CRED_REQUEST, sessionID),
                sessionNonce${UID}, ownerPubkey${UID}, devicePrivkey)));

  in(io, (=DATA_STORE_SIEVE_CRED_RESPONSE, =ownerPubkey${UID},
     responseNonce${UID}:nonce, ciphertext${UID}:bitstring));

  let (=CIPHER_DATA_STORE_SIEVE_CRED_RESPONSE, sieveKey${UID}:sieve_key,
     sieveNonce${UID}:nonce) = crypto_box_open_easy(ciphertext${UID},
     responseNonce${UID}, ownerPubkey${UID}, devicePrivkey) in

  (* Split data key into shares *)
  let dataKeyShare${UID} = split_key_shares_${UID}(dataKey) in

  (* Create Sieve data block per user *)
  let sieveContentBlock${UID} = (dataKeyShare${UID}) in
  new sieveDataHint${UID}:bitstring;
  let sieveDataBlock${UID} = sieve_enc(sieveKey${UID}, sieveNonce${UID},
     sieveContentBlock${UID}) in
  out(io, (DATA_STORE_UPLOAD, sieveDataUUID${UID}, $foreach{i}{UID}{$if{${i}
       == ${UID}}{ownerPubkey${UID}}{NULLPK}}$with{,},
   sieveDataBlock${UID}));

  new notificationNonce1${UID}:nonce;
  out(io, (DATA_STORE_UPLOAD_NOTIFICATION_1,
           ownerPubkey${UID},
           notificationNonce1${UID},
           crypto_box_easy((CIPHER_DATA_STORE_UPLOAD_NOTIFICATION_1,
              sessionID, sieveDataUUID${UID}, dataUUID), notificationNonce1${
              UID}, ownerPubkey${UID}, devicePrivkey)));
$endtemplate
```

```
  (* Assemble meta data block *)
  let ownerInfo = ($expand{sieveNonce${UID}, ownerPubkey${UID},
     sieveDataUUID${UID}, sieveDataHint${UID}}$with{,}) in
  let metaData = (ownerInfo, dataUUID, dataHeader) in
  out(io, (DATA_STORE_UPLOAD, metaDataUUID, $expand{ownerPubkey${UID}}$with{
     ,}, metaData));

  event DataStoreDeviceFinish($expand{ownerPubkey${UID}}$with{,},
     devicePubkey, sessionID);
  event DataStoreAssignOwnership($expand{ownerPubkey${UID}}$with{,},
     metaDataUUID, privateUserData);

  (* Send upload notification to data owner *)
$template ${UID}
  new notificationNonce2${UID}:nonce;
  out(io, (DATA_STORE_UPLOAD_NOTIFICATION_2,
           ownerPubkey${UID},
           notificationNonce2${UID},
           crypto_box_easy((CIPHER_DATA_STORE_UPLOAD_NOTIFICATION_2,
              sessionID, metaDataUUID), notificationNonce2${UID},
              ownerPubkey${UID}, devicePrivkey)));
$endtemplate

  0.


$template ${UID}
(* userRecordingLookupTable(userPubkey, sessionID, metaDataUUID,
   sieveDataUUID, sieveKey) *)
table userRecordingLookupTable${UID}(pubkey, bitstring, bitstring, bitstring
   , sieve_key).


let UserDataStore${UID}(userPrivkey:privkey) =
  let userPubkey = pk(userPrivkey) in

  in(io, (=DATA_STORE_SIEVE_CRED_REQUEST, devicePubkey:pubkey, sessionNonce:
     nonce, =userPubkey, ciphertext:bitstring));

  let (=CIPHER_DATA_STORE_SIEVE_CRED_REQUEST, sessionID:bitstring) =
     crypto_box_open_easy(ciphertext, sessionNonce, devicePubkey,
     userPrivkey) in

  get userControlDeviceTable${UID}(=userPubkey, =devicePubkey) in

  new sieveKey:sieve_key;
  new sieveNonce:nonce;
  new responseNonce:nonce;

  out(io, (DATA_STORE_SIEVE_CRED_RESPONSE,
           userPubkey,
```

```
          responseNonce,
          crypto_box_easy((CIPHER_DATA_STORE_SIEVE_CRED_RESPONSE, sieveKey,
              sieveNonce), responseNonce, devicePubkey, userPrivkey)
          ));

  in(io, (=DATA_STORE_UPLOAD_NOTIFICATION_1, =userPubkey, notificationNonce1
      :nonce, notificationPayload1:bitstring));

  let (=CIPHER_DATA_STORE_UPLOAD_NOTIFICATION_1, =sessionID, sieveDataUUID:
      bitstring, dataUUID:bitstring) = crypto_box_open_easy(
      notificationPayload1, notificationNonce1, devicePubkey, userPrivkey) in

   in(io, (=DATA_STORE_UPLOAD_NOTIFICATION_2, =userPubkey,
       notificationNonce2:nonce, notificationPayload2:bitstring));

  let (=CIPHER_DATA_STORE_UPLOAD_NOTIFICATION_2, =sessionID, metaDataUUID:
      bitstring) = crypto_box_open_easy(notificationPayload2,
      notificationNonce2, devicePubkey, userPrivkey) in

  insert userRecordingLookupTable${UID}(userPubkey, sessionID, metaDataUUID,
       sieveDataUUID, sieveKey);

  event DataStoreUserReceiveNotification${UID}(userPubkey, devicePubkey,
      sessionID);

  out(io, metaDataUUID);
  0.
$endtemplate

(********** DataAccess **********)
table approvedAccessor(pubkey).

(* Malicious accessor stores stale sieve key *)
table accessorDataCache(bitstring, sieve_key).

table malicious(bitstring).

let AccessorDataAccess(accessorPrivkey:privkey) =
  let accessorPubkey = pk(accessorPrivkey) in

  in(io, metaDataUUID:bitstring);
  insert malicious(metaDataUUID);

  (* Download Sieve data block from storage *)
  out(io, (DATA_STORE_DOWNLOAD_REQUEST, metaDataUUID));
  in(io, (=DATA_STORE_DOWNLOAD_RESPONSE_1, $expand{ownerPubkey${UID}:pubkey}
      $with{,}, metaData:bitstring));

  (* Unmarshalling Sieve data block *)
  let (($expand{sieveNonce${UID}:nonce, =ownerPubkey${UID}, sieveDataUUID${
      UID}:bitstring, sieveDataHint${UID}:bitstring}$with{,}), dataUUID:
      bitstring, dataHeader:secretstream_header) = metaData in
```

```
$template ${UID}
  new randomNoise${UID}:bitstring;
  new payloadNonce${UID}:nonce;
  let payload${UID} = crypto_box_easy((metaDataUUID, randomNoise${UID}),
      payloadNonce${UID}, ownerPubkey${UID}, accessorPrivkey) in
  out(io, (DATA_ACCESS_FETCH, accessorPubkey, payloadNonce${UID}, payload${
      UID}));

  in(io, (=DATA_ACCESS_RESPONSE, msgNonce${UID}:nonce, ciphertext${UID}:
      bitstring));

  (* Retrieve user specific Sieve key *)
  let (=CIPHER_DATA_ACCESS_RESPONSE, sieveKey${UID}:sieve_key, =randomNoise$
      {UID}) = crypto_box_open_easy(ciphertext${UID}, msgNonce${UID},
      ownerPubkey${UID}, accessorPrivkey) in

  (* Download metadata block and decrypt *)
  out(io, (DATA_STORE_DOWNLOAD_REQUEST, sieveDataUUID${UID}));
  in(io, (=DATA_STORE_DOWNLOAD_RESPONSE_1, $foreach{i}{UID}{$if{${i} == ${
      UID}}{=ownerPubkey${UID}}{dummy_${UID}_${i}:pubkey}}$with{,},
      sieveDataBlock${UID}:bitstring));
  let sieveContentBlock${UID} = sieve_dec(sieveKey${UID}, sieveNonce${UID},
      sieveDataHint${UID}, sieveDataBlock${UID}) in
  let (dataKeyShare${UID}:bitstring) = sieveContentBlock${UID} in
$endtemplate

  let dataKey:secretstream_key = assemble_key_shares($expand{dataKeyShare${
      UID}}$with{,}) in

  (* Retrieve data *)
  out(io, (DATA_STORE_DOWNLOAD_REQUEST, dataUUID));
  in(io, (=DATA_STORE_DOWNLOAD_RESPONSE_1, $expand{=ownerPubkey${UID}}$with{
      ,}, dataEncrypted:bitstring));

(*
  get storageTableVersion2(=dataUUID, a:pubkey, b:bitstring) in 0 else
  insert storageTableVersion2(dataUUID, ownerPubkey, dataEncrypted);
*)

  let dataDecrypted = crypto_secretstream_decrypt(dataKey, dataHeader,
      dataEncrypted) in

  if (dataDecrypted = privateUserData) then
    event DataAccessAccessorDecrypt(accessorPubkey, privateUserData);
$template ${UID}
    insert accessorDataCache(sieveDataUUID${UID}, sieveKey${UID});
$endtemplate
    0.


$template ${UID}
```

```
let UserDataAccess${UID}(userPrivkey:privkey) =
  let userPubkey = pk(userPrivkey) in
  in(io, (=DATA_ACCESS_FETCH, accessorPubkey:pubkey, payloadNonce:nonce,
      payload:bitstring));

  get approvedAccessor(=accessorPubkey) in

  let (metaDataUUID:bitstring, randomNoise:bitstring) = crypto_box_open_easy
      (payload, payloadNonce, accessorPubkey, userPrivkey) in

  get userRecordingLookupTable${UID}(=userPubkey, sessionID:bitstring, =
      metaDataUUID, sieveDataUUID:bitstring, sieveKey:sieve_key) in

  event DataAccessUserGrantPermission${UID}(userPubkey, accessorPubkey,
      metaDataUUID);

  new msgNonce:nonce;
  let ciphertext = crypto_box_easy((CIPHER_DATA_ACCESS_RESPONSE, sieveKey,
      randomNoise), msgNonce, accessorPubkey, userPrivkey) in
  out(io, (DATA_ACCESS_RESPONSE, msgNonce, ciphertext));

  0.
$endtemplate


(*********** Storage daemon ***********)
let StorageDownloadDaemon() =
  in(io, (=DATA_STORE_DOWNLOAD_REQUEST, blockUUID:bitstring));
  get storageTableV0(=blockUUID, $expand{ownerPubkey${UID}:pubkey}$with{,},
      content:bitstring) in
  out(io, (DATA_STORE_DOWNLOAD_RESPONSE_1, $expand{ownerPubkey${UID}}$with{,
      }, content));
  0.

let StorageUploadDaemon() =
  in(io, (=DATA_STORE_UPLOAD, blockUUID:bitstring, $expand{ownerPubkey${UID}
      :pubkey}$with{,}, content:bitstring));
  insert storageTableV0(blockUUID, $expand{ownerPubkey${UID}}$with{,},
      content);
  0.


(*********** DataReencryption ***********)
$template ${UID}
let UserDataReencryption${UID}(userPrivkey:privkey, storagePubkey:pubkey) =
phase $compute{(${UID}-1)*${REENCRYPT_PHASE_INTERVAL}+1};

  let userPubkey = pk(userPrivkey) in
  get userRecordingLookupTable${UID}(=userPubkey, sessionID:bitstring,
      metaDataUUID:bitstring, sieveDataUUID:bitstring, sieveKey:sieve_key) in

  (* Model key revocation: make a fresh new sieve key *)
```

```
  new sieveNewKey:sieve_key;
  (* Alternatively: make the new key the same as old key to check the
     reachability of DataReencryptionAccessorSucceedDecryptOldKey()  *)
  (* let sieveNewKey = sieveKey in *)

  let rekeyToken = sieve_rekey_token(sieveKey, sieveNewKey) in

  (* Add session-specific negotiation to prevent replay attack *)
  new userNonce: nonce;
  let preReqCiphertext = crypto_box_seal(
    (
      CIPHER_DATA_REENCRYPTION_PRE_REQUEST,
      metaDataUUID,
      sieveDataUUID,
      userNonce
    ),
    storagePubkey) in
  out(io, (DATA_REENCRYPTION_PRE_REQUEST, preReqCiphertext));
  in(io, (=DATA_REENCRYPTION_PRE_RESPONSE, preResCiphertext:bitstring));
  let (=CIPHER_DATA_REENCRYPTION_PRE_RESPONSE, =userNonce, storeNonce:nonce)
   = crypto_box_seal_open(preResCiphertext, userPrivkey) in

  event DataReencryptionUserRequestKeySwitch${UID}(sieveDataUUID, userNonce,
     storeNonce);

  new notificationToken:bitstring;
  new msgNonce:nonce;

  out(io, (DATA_REENCRYPTION_REQUEST, metaDataUUID, msgNonce,
     crypto_box_easy((CIPHER_DATA_REENCRYPTION_REQUEST, rekeyToken,
     notificationToken, userNonce, storeNonce), msgNonce, storagePubkey,
     userPrivkey)));

  in(io, (=DATA_REENCRYPTION_RESPONSE, =notificationToken));

(* Can't update this table again here. Cause infinite loop. It's okay if we
    just want to validate rekey token revoke old sieve keys. As the result,
    the old data is never accessible after rekey in the current model... *)
(*    insert userRecordingLookupTable(userPubkey, sessionID, metaDataUUID,
    dataUUID, sieveNewKey);
*)
  event DataReencryptionUserSwitchKey${UID}(sieveDataUUID, userNonce,
     storeNonce);
  0.
$endtemplate

$template ${UID}
let StorageDataReencryptionPhased${UID}(storagePrivkey:privkey) =
  phase $compute{(${UID}-1)*${REENCRYPT_PHASE_INTERVAL}+1};


  in(io, (=DATA_REENCRYPTION_PRE_REQUEST, preReqCiphertext:bitstring));
```

```
  let (=CIPHER_DATA_REENCRYPTION_PRE_REQUEST, metaDataUUID:bitstring,
      sieveDataUUIDReq:bitstring, userNonce:nonce) = crypto_box_seal_open(
      preReqCiphertext, storagePrivkey) in

  get storageTableV$compute{${UID}-1}
    (=metaDataUUID, $expand{ownerPubkey${UID}:pubkey}$with{,}, metaData:
       bitstring) in

  let (($expand{sieveNonce${UID}:nonce, =ownerPubkey${UID}, sieveDataUUID${
      UID}:bitstring, sieveDataHint${UID}:bitstring}$with{,}), dataUUID:
      bitstring, dataHeader:secretstream_header) = metaData in

  if sieveDataUUID${UID} = sieveDataUUIDReq then
  new storeNonce:nonce;
  let preCiphertext = crypto_box_seal((CIPHER_DATA_REENCRYPTION_PRE_RESPONSE
      , userNonce, storeNonce), ownerPubkey${UID}) in
  out(io, (DATA_REENCRYPTION_PRE_RESPONSE, preCiphertext));

  in(io, (=DATA_REENCRYPTION_REQUEST, metaDataUUID:bitstring, msgNonce:nonce
      , ciphertext:bitstring));


  insert storageTableV${UID}
    (metaDataUUID, $expand{ownerPubkey${UID}}$with{,}, metaData);

  let (=CIPHER_DATA_REENCRYPTION_REQUEST, rekeyToken:rekey_token,
      notificationToken:bitstring, =userNonce, =storeNonce) =
      crypto_box_open_easy(ciphertext, msgNonce, ownerPubkey${UID},
      storagePrivkey) in


  get storageTableV0(=sieveDataUUID${UID}, $foreach{i}{UID}{$if{${i} == ${
      UID}}{=ownerPubkey${UID}}{=NULLPK}}$with{,}, sieveDataBlock:bitstring)
      in

  let sieveDataBlockNewKey = sieve_rekey(sieveDataBlock, rekeyToken) in

  insert storageTableV${UID}
    (sieveDataUUID${UID}, $foreach{i}{UID}{$if{${i} == ${UID}}{ownerPubkey${
       UID}}{NULLPK}}$with{,}, sieveDataBlockNewKey);

  event DataReencryptionStorageSwitchKey${UID}(sieveDataUUID${UID},
      userNonce,storeNonce);

  out(io, (DATA_REENCRYPTION_RESPONSE, notificationToken));
  0.
$endtemplate

let StorageDataReencryption(storagePrivkey:privkey) =
  $expand{StorageDataReencryptionPhased${UID}(storagePrivkey)}$with{ | }
  | 0.
```

```
$template ${UID}
let AccessorBadDataReencryptionPhased${UID}() =
phase $compute{(${UID}-1)*${REENCRYPT_PHASE_INTERVAL}+2};
  get malicious(metaDataUUID:bitstring) in

  get storageTableV${UID}
    (=metaDataUUID, $expand{ownerPubkey${UID}:pubkey}$with{,}, metaData) in

  let (($expand{sieveNonce${UID}:nonce, =ownerPubkey${UID}, sieveDataUUID${
      UID}:bitstring, sieveDataHint${UID}:bitstring}$with{,}), dataUUID:
      bitstring, dataHeader:secretstream_header) = metaData in

  $expand{get accessorDataCache(sieveDataUUID${UID}:bitstring, sieveKey${UID
      }:sieve_key) in }$with{ }

  event DataReencryptionAccessorTryDecryptOldKey${UID}(metaDataUUID,
      sieveDataUUID${UID});


  $foreach{i}{UID}{get storageTableV$if{${i} == ${UID}}{${i}}{0} (=
      sieveDataUUID${i}, $foreach{j}{UID}{$if{${j} == ${UID}}{=ownerPubkey${
      UID}}{=NULLPK}}$with{,} , sieveDataBlock${i}) in }$with{ }


  $expand{let sieveContentBlock${UID} = sieve_dec(sieveKey${UID},
      sieveNonce${UID}, sieveDataHint${UID}, sieveDataBlock${UID}) in let (
      dataKeyShare${UID}:bitstring) = sieveContentBlock${UID} in }$with{ }

  let dataKey:secretstream_key = assemble_key_shares($expand{dataKeyShare${
      UID}}$with{,}) in

  out(io, (DATA_STORE_DOWNLOAD_REQUEST, dataUUID));
  in(io, (=DATA_STORE_DOWNLOAD_RESPONSE_1, a:pubkey, dataEncrypted:bitstring
      ));

  let dataDecrypted = crypto_secretstream_decrypt(dataKey, dataHeader,
      dataEncrypted) in

    if (dataDecrypted = privateUserData) then
    event DataReencryptionAccessorSucceedDecryptOldKey${UID}(metaDataUUID,
        sieveDataUUID${UID});

  0.
$endtemplate


let AccessorBadDataReencryption() =
  $expand{AccessorBadDataReencryptionPhased${UID}()}$with{ | }
  | 0.
```

```
(*********************************************************)
(* Main Process *)
(*********************************************************)

const adminName, storageName:bitstring.

(* Lookup table for canonical name <--> pubkey *)
table KMS(bitstring, pubkey).

let AdminProcess(setupKey:secretstream_key) =
  new adminPrivkey: privkey;
  let adminPubkey = pk(adminPrivkey) in
  insert KMS(adminName, adminPubkey);
  AdminInitialization(setupKey, adminPrivkey)
  | !AdminAcquirePreAuthToken(adminPrivkey)
  | 0.

let DeviceProcess(setupKey:secretstream_key) =
  new devicePrivkey: privkey;
  let devicePubkey = pk(devicePrivkey) in
  DeviceInitialization(setupKey, devicePrivkey)
$template ${UID}
  | DeviceClaimDevice${UID}(devicePrivkey)
$endtemplate
  | DeviceDataStore(devicePrivkey)
  | 0.

$template ${UID}
let UserProcess${UID}() =
  new userPrivkey:privkey;
  get KMS(=adminName, adminPubkey) in
  get KMS(=storageName, storagePubkey) in
  UserAcquirePreAuthToken(userPrivkey, adminPubkey)
  | UserClaimDevice${UID}(userPrivkey, adminPubkey)
  | UserDataStore${UID}(userPrivkey)
  | UserDataAccess${UID}(userPrivkey)
  | UserDataReencryption${UID}(userPrivkey, storagePubkey)
  | 0.
$endtemplate

let StorageProcess() =
  new storagePrivkey:privkey;
  let storagePubkey = pk(storagePrivkey) in
  insert KMS(storageName, storagePubkey);
  !StorageDownloadDaemon()
  | !StorageUploadDaemon()
  | !StorageDataReencryption(storagePrivkey)
  | 0.

let AccessorProcess() =
  new accessorPrivkey:privkey;
```

```
  let accessorPubkey = pk(accessorPrivkey) in
  insert approvedAccessor(accessorPubkey);
  AccessorDataAccess(accessorPrivkey)
  | AccessorBadDataReencryption()
  | 0.

process
  new setupKey:secretstream_key;
  AdminProcess(setupKey)
  | !DeviceProcess(setupKey)
$template ${UID}
  | !UserProcess${UID}()
$endtemplate
  | StorageProcess()
  | !AccessorProcess()
```

Bibliography

[1] Paarijaat Aditya, Rijurekha Sen, Peter Druschel, Seong Joon Oh, Rodrigo Benenson, Mario Fritz, Bernt Schiele, Bobby Bhattacharjee, and Tong Tong Wu. I-pic: A platform for privacy-compliant image capture. In *Proceedings of the 14th annual international conference on mobile systems, applications, and services*, pages 235–248, 2016. 2.1.2

[2] Ayaz Akram, Venkatesh Akella, Sean Peisert, and Jason Lowe-Power. Sok: Limitations of confidential computing via tees for high-performance compute systems. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED)*, pages 121–132. IEEE, 2022. 2.3.1

[3] Fatemah Alharbi, Arwa Alrawais, Abdulrahman Bin Rabiah, Silas Richelson, and Nael Abu-Ghazaleh. Csprop: Ciphertext and signature propagation low-overhead public-key cryptosystem for iot environments. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, pages 609–626, 2021. 3.7, 6.2

[4] Alasdair Allan. The problem with throwing away a smart device. `https://www.hackster.io/news/the-problem-with-throwing-away-a-smart-device-75c8b35ee3c7`, 2020. 1.1, 4.2.2, 4.6.2, 5.9.1

[5] Alasdair Allan. Teardown of a smart plug (or two). `https://www.hackster.io/news/teardown-of-a-smart-plug-or-two-6462bd2f275b`, 2020. 1.1, 4.2.2, 4.6.2, 5.9.1

[6] Omar Alrawi, Chaz Lever, Manos Antonakakis, and Fabian Monrose. SoK: Security evaluation of home-based IoT deployments. In *2019 IEEE Symposium on Security and Privacy*, 2019. 1, 4.1, 4.2

[7] Amazon. Blink video doorbell. `https://www.amazon.com/Blink-Video-Doorbell/dp/B08SG2MS3V`, 2022. 5.1

[8] Amazon Web Services. AWS IoT Greengrass. `https://aws.amazon.com/greengrass/`, 2020. 1, 2.2.3, 4.1

[9] Mahmoud Ammar, Bruno Crispo, and Gene Tsudik. Simple: A remote attestation approach for resource-constrained iot devices. In *2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS)*, pages 247–258. IEEE, 2020. 3.7

[10] Michael P Andersen, Sam Kumar, Moustafa AbdelBaky, Gabe Fierro, John Kolb, Hyung-Sin Kim, David E Culler, and Raluca Ada Popa. {WAVE}: A decentralized authorization framework with transitive delegation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1375–1392, 2019. 2.1.2, 3.1, 5.5.1

[11] Android. Batterymanager. https://developer.android.com/reference/android/os/BatteryManager, 2021. 3.6.1

[12] Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J. Alex Halderman, Luca Invernizzi, Michalis Kallitsis, Deepak Kumar, Chaz Lever, Zane Ma, Joshua Mason, Damian Menscher, Chad Seaman, Nick Sullivan, Kurt Thomas, and Yi Zhou. Understanding the mirai botnet. In *26th USENIX Security Symposium*, 2017. 1, 4.1

[13] Yoshinori Aono, Takuya Hayashi, Lihua Wang, Shiho Moriai, et al. Privacy-preserving deep learning via additively homomorphic encryption. *IEEE Transactions on Information Forensics and Security*, 13(5):1333–1345, 2017. 2.3.3

[14] apache-jmeter. Apache JMeter. https://jmeter.apache.org/, 2020. 4.7.1

[15] Andrew W Appel and Edward W Felten. Proof-carrying authentication. In *Proceedings of the 6th ACM Conference on Computer and Communications Security*, pages 52–62, 1999. 3.2.1

[16] Apple. Bonjour. https://developer.apple.com/bonjour/, 2021. 3.3.3

[17] Aref Asvadishirehjini, Murat Kantarcioglu, and Bradley Malin. Goat: GPU outsourcing of deep learning training with asynchronous probabilistic integrity verification inside trusted execution environment. *arXiv preprint arXiv:2010.08855*, 2020. 2.3.2

[18] Gbadebo Ayoade, Vishal Karande, Latifur Khan, and Kevin Hamlen. Decentralized IoT data management using blockchain and trusted execution environment. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 15–22. IEEE, 2018. 2.1.2, 3, 3.1, 3.7

[19] Manuel Barbosa, Gilles Barthe, Karthik Bhargavan, Bruno Blanchet, Cas Cremers, Kevin Liao, and Bryan Parno. SoK: Computer-aided cryptography. In *IEEE Symposium on Security and Privacy*, May 2021. 3.4.2

[20] Lujo Bauer, Scott Garriss, Jonathan M McCune, Michael K Reiter, Jason Rouse, and Peter Rutenbar. Device-enabled authorization in the grey system. In *International Conference on Information Security*, pages 431–445. Springer, 2005. 2.1.2, 3, 3.1, 3.2.1

[21] Abdelkareem Bedri, Yuchen Liang, Sudershan Boovaraghavan, Geoff Kaufman, and Mayank Goel. Fitnibble: A field study to evaluate the utility and usability of automatic diet monitoring in food journaling using an eyeglasses-based wearable. *Proceedings of the Annual Conference on Intelligent User Interfaces*, 2022. 6.2

[22] Julia Bernd, Ruba Abu-Salma, and Alisa Frik. Bystanders' privacy: The perspectives of nannies on smart home surveillance. In *10th USENIX Workshop on Free and Open Communications on the Internet (FOCI 20)*, 2020. 1, 2.1.1, 3, 3.1

[23] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with contextual caveats for decentralized authorization in the cloud. In *Network and Distributed System Security Symposium*, 2014. 3.2.1

[24] Andrew D Birrell and Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 1984. 4.8

[25] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *International Conference on Principles of Security and Trust*, pages 3–29. Springer, 2012. 3.4.2

[26] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and proverif. *Found. Trends Priv. Secur.*, 2016. 1.2, 3.4, 3.4.2

[27] Blynk.io. Blynk library. `https://github.com/blynkkk/blynk-library`, 2022. 3.6.2

[28] Dan Boneh, Kevin Lewi, Hart Montgomery, and Ananth Raghunathan. Key homomorphic prfs and their applications. In *Annual Cryptology Conference*, pages 410–428. Springer, 2013. 1.2, 3.3, 3.3.4

[29] Sudershan Boovaraghavan, Anurag Maravi, Prahaladha Mallela, and Yuvraj Agarwal. MLIoT: An end-to-end machine learning system for the Internet-of-Things. In *6th ACM/IEEE Conference on Internet of Things Design and Implementation*, IoTDI, 2021. 5.1

[30] Sudershan Boovaraghavan, Chen Chen, Anurag Maravi, Mike Czapik, Yang Zhang, Chris Harrison, and Yuvraj Agarwal. Mites: Design and deployment of a general-purpose sensing infrastructure for buildings. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 7(1), mar 2023. doi: 10.1145/3580865. URL `https://doi.org/10.1145/3580865`. 1, 3.1, 5.9.1

[31] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on Android. In *NDSS*, 2012. 4.8

[32] Daming D. Chen, Maverick Woo, David Brumley, and Manuel Egele. Towards automated dynamic analysis for Linux-based embedded firmware. In *NDSS*, 2016. 4.2

[33] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 142–157. IEEE, 2019. 2.3.1

[34] Huili Chen, Cheng Fu, Bita Darvish Rouhani, Jishen Zhao, and Farinaz Koushanfar. Deepattest: an end-to-end attestation framework for deep neural networks. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pages 487–498. IEEE, 2019. 2.3.1

[35] Yu-Hsin Chen, Tushar Krishna, Joel S Emer, and Vivienne Sze. Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks. *IEEE journal of solid-state circuits*, 52(1):127–138, 2016. 5.5.2

[36] Byung-Gon Chun, Sunghwan Ihm, Petros Maniatis, Mayur Naik, and Ashwin Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314, 2011. 5.1

[37] CNBC. Amazon Alexa records you every time you ask it something — here's how to delete those recordings. `https://www.cnbc.com/2021/02/18/how-to-del`

ete-amazon-alexa-recordings-for-privacy.html, 2021. 3.6.2

[38] CNBC. Meet the $10,000 Nvidia chip powering the race for A.I. https://www.cnbc.com/2023/02/23/nvidias-a100-is-the-10000-chip-powering-the-race-for-ai-.html, 2023. 2.3.1

[39] CNET. Amazon's Astro may be cute, but security experts warn of privacy concerns. https://www.cnet.com/tech/amazons-astro-may-be-cute-but-security-experts-warn-of-privacy-concerns/, 2021. 3.6.2

[40] Camille Cobb, Sruti Bhagavatula, Kalil Anderson Garrett, Alison Hoffman, Varun Rao, and Lujo Bauer. "i would have to evaluate their objections": Privacy tensions between smart home device owners and incidental users. *Proceedings on Privacy Enhancing Technologies*, 4:54–75, 2021. 2.1.1, 3.1

[41] Jessica Colnago, Yuanyuan Feng, Tharangini Palanivel, Sarah Pearman, Megan Ung, Alessandro Acquisti, Lorrie Faith Cranor, and Norman Sadeh. Informing the design of a personalized privacy assistant for the internet of things. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, 2020. 2.1.1

[42] Computer Weekly. Third-party code bug left instagram users at risk of account takeovers. https://www.computerweekly.com/news/252489542/Third-party-code-bug-left-Instagram-users-at-risk-of-account-takeover, 2020. 4.1

[43] Coral. Usb accelerator. https://coral.ai/products/accelerator/, 2022. 5.8

[44] Coral. Edge tpu inferencing overview. https://coral.ai/docs/edgetpu/inference/, 2022. 5.8

[45] Craig Costello, Cédric Fournet, Jon Howell, Markulf Kohlweiss, Benjamin Kreuter, Michael Naehrig, Bryan Parno, and Samee Zahur. Geppetto: Versatile verifiable computation. In *2015 IEEE Symposium on Security and Privacy*, pages 253–270. IEEE, 2015. 2.3.2

[46] Crypto++. Crypto++. https://www.cryptopp.com/, 2021. 3.5

[47] Eduardo Cuervo, Aruna Balasubramanian, Dae-ki Cho, Alec Wolman, Stefan Saroiu, Ranveer Chandra, and Paramvir Bahl. Maui: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62, 2010. 5.1

[48] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on Android. In *International Conference on Information Security*, 2010. 4.8

[49] Alexandre Decan, Tom Mens, and Eleni Constantinou. On the impact of security vulnerabilities in the npm package dependency network. In *15th International Conference on Mining Software Repositories*, MSR, 2018. 4.1

[50] Soteris Demetriou, Nan Zhang, Yeonjoon Lee, XiaoFeng Wang, Carl A. Gunter, Xiaoyong Zhou, and Michael Grace. HanGuard: SDN-driven protection of smart home WiFi devices

from malicious mobile apps. In *10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec, 2017. 2.2.1

[51] Ailin Deng and Bryan Hooi. Graph neural network-based anomaly detection in multivariate time series. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 4027–4035, 2021. 5.7.2

[52] Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017. 4.3.2

[53] Rajib Dey, Sayma Sultana, Afsaneh Razi, and Pamela J Wisniewski. Exploring smart home device use by Airbnb hosts. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems*, pages 1–8, 2020. 3.1

[54] Brian Dipert. Teardown: A WiFi smart plug for home automation. `https://www.edn.com/teardown-a-wi-fi-smart-plug-for-home-automation/`, 2020. 1.1, 4.2.2

[55] Brian Dipert. Teardown: WeMo switch is highly integrated. `https://www.edn.com/teardown-wemo-switch-is-highly-integrated/`, 2020. 1.1, 4.2.2

[56] Colin Dixon, Ratul Mahajan, Sharad Agarwal, A.J. Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *9th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2012. 2.2.3

[57] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983. 3.2.3

[58] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021. URL `https://openreview.net/forum?id=YicbFdNTTy`. 5.9, 5.9.1

[59] Jianbo Du, F Richard Yu, Xiaoli Chu, Jie Feng, and Guangyue Lu. Computation offloading and resource allocation in vehicular networks based on dual-side cost minimization. *IEEE Transactions on Vehicular Technology*, 68(2):1079–1092, 2018. 6.2

[60] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *2017 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2017. 4.2.1

[61] Chethana Dukkipati, Yunpeng Zhang, and Liang Chieh Cheng. Decentralized, blockchain based access control framework for the heterogeneous internet of things. In *Proceedings of the Third ACM Workshop on Attribute-Based Access Control*, pages 61–69, 2018. 2.1.2, 3, 3.1

[62] Zakir Durumeric, David Adrian, Ariana Mirian, Michael Bailey, and J. Alex Halderman. A search engine backed by Internet-wide scanning. In *2015 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2015. 4.1

[63] Stefan Dziembowski, Lisa Eckey, and Sebastian Faust. Fairswap: How to fairly exchange

digital goods. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 967–984, 2018. 5.5.1

[64] Pardis Emami-Naeini, Janarth Dheenadhayalan, Yuvraj Agarwal, and Lorrie Faith Cranor. Which privacy and security attributes most impact consumers' risk perception and willingness to purchase IoT devices? In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1937–1954, 2021. 1, 3, 3.1

[65] Jeremy Erickson, Qi Alfred Chen, Xiaochen Yu, Erinjen Lin, Robert Levy, and Z Morley Mao. No one in the middle: Enabling network access control via transparent attribution. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, pages 651–658, 2018. 2.2.1, 3.7, 4.1, 4.3.2, 4.3.4

[66] ESP32.NET. The internet of things with ESP32. `http://esp32.net/`, 2020. 4.6.2

[67] Espressif. [SDK release] esp8266-nonos-sdk-v1.5.0-15-11-27. `https://bbs.espressif.com/viewtopic.php?f=46&t=1442`, 2015. 4.3.4

[68] Espressif. We just hit a new major milestone. `https://www.espressif.com/en/media_overview/news/espressif-achieves-100-million-target-iot-chip-shipments`, 2018. 4.6.2

[69] Espressif. WolfSSL for ESP-IDF. `https://github.com/espressif/esp-wolfssl`, 2020. 4.2.2

[70] Espressif. Esp8266 datasheet. `https://www.espressif.com/sites/default/files/documentation/0a-esp8266ex_datasheet_en.pdf`, 2020. 3.7

[71] 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, Austin, TX, August 2016. USENIX Association. ISBN 978-1-931971-32-4. URL `https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/fernandes`. 2.1.2, 3, 3.1

[72] Dario Fiore, Cédric Fournet, Esha Ghosh, Markulf Kohlweiss, Olga Ohrimenko, and Bryan Parno. Hash first, argue later: Adaptive verifiable computations on outsourced data. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1304–1316, 2016. 2.3.2

[73] fmerg. pymerkle. `https://github.com/fmerg/pymerkle`, 2022. 5.8

[74] Francesco Fraternali, Bharathan Balaji, Yuvraj Agarwal, and Rajesh K Gupta. ACES: Automatic configuration of energy harvesting sensors with reinforcement learning. *ACM Transactions on Sensor Networks (TOSN)*, 16(4):1–31, 2020. 1, 3.1

[75] Diana Freed, Jackeline Palmer, Diana Elizabeth Minchala, Karen Levy, Thomas Ristenpart, and Nicola Dell. Digital technologies and intimate partner violence: A qualitative analysis with multiple stakeholders. *Proceedings of the ACM on Human-Computer Interaction*, 1(CSCW):1–22, 2017. 2.1.1

[76] Diana Freed, Sam Havron, Emily Tseng, Andrea Gallardo, Rahul Chatterjee, Thomas Ristenpart, and Nicola Dell. " is my phone hacked?" analyzing clinical computer security

interventions with survivors of intimate partner violence. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–24, 2019. 2.1.1

[77] Rosario Gennaro, Craig Gentry, and Bryan Parno. Non-interactive verifiable computing: Outsourcing computation to untrusted workers. In *Annual Cryptology Conference*, pages 465–482. Springer, 2010. 2.3.2

[78] Zahra Ghodsi, Tianyu Gu, and Siddharth Garg. Safetynets: Verifiable execution of deep neural networks on an untrusted cloud. *Advances in Neural Information Processing Systems*, 30, 2017. 2.3.2, 5.1, 5.5.1

[79] Ran Gilad-Bachrach, Nathan Dowlin, Kim Laine, Kristin Lauter, Michael Naehrig, and John Wernsing. Cryptonets: Applying neural networks to encrypted data with high throughput and accuracy. In *International conference on machine learning*, pages 201–210. PMLR, 2016. 2.3.3, 5.1

[80] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014. 5.7.2

[81] Google. Protocol buffers. `https://developers.google.com/protocol-buffers`, 2020. 4.8

[82] Google. Flatbuffers. `https://google.github.io/flatbuffers/`, 2021. 3.5

[83] Google. Leveldb. `https://github.com/google/leveldb`, 2021. 3.5

[84] Google. Google tensor is a milestone for machine learning. `https://blog.google/products/pixel/introducing-google-tensor/`, 2021. 5.8

[85] Google. Protocol buffers. `https://developers.google.com/protocol-buffers`, 2022. 5.8

[86] grpc. gRPC — a high-performance, open source universal rpc framework. `https://grpc.io/`, 2020. 4.8

[87] Hang Guo and John Heidemann. Detecting IoT devices in the internet (extended). Technical Report ISI-TR-726B, USC/Information Sciences Institute, 2018. 4.1

[88] Hackaday.io. 382 projects tagged with "ESP32". `https://hackaday.io/projects?tag=ESP32`, 2020. 4.6.2

[89] Hacker Shack. Smartphone connected home door lock. `https://www.hackster.io/hackershack/smartphone-connected-home-door-lock-69944f`, 2017. 3.4, 3.6.2

[90] Mike Hamburg. Ed448-goldilocks. `https://sourceforge.net/p/ed448goldilocks/wiki/Home/`, 2021. 3.5

[91] Jun Han, Albert Jin Chung, Manal Kumar Sinha, Madhumitha Harishankar, Shijia Pan, Hae Young Noh, Pei Zhang, and Patrick Tague. Do you feel what I hear? enabling autonomous IoT device pairing using different sensor types. In *2018 IEEE Symposium on Security and Privacy*, 2018. 4.4

[92] Sam Havron, Diana Freed, Rahul Chatterjee, Damon McCoy, Nicola Dell, and Thomas Ristenpart. Clinical computer security for victims of intimate partner violence. In *28th*

USENIX Security Symposium (USENIX Security 19), 2019. 2.1.1

[93] Weijia He, Maximilian Golla, Roshni Padhi, Jordan Ofek, Markus Dürmuth, Earlence Fernandes, and Blase Ur. Rethinking access control and authentication for the home internet of things (iot). In *27th USENIX Security Symposium (USENIX Security 18)*, 2018. 1, 2.1.1, 3, 3.1

[94] Weijia He, Valerie Zhao, Olivia Morkved, Sabeeka Siddiqui, Earlence Fernandes, Josiah D. Hester, and Blase Ur. SoK: Context sensing for access control in the adversarial home iot. In *Proceedings of the 6th IEEE European Symposium on Security and Privacy*, 2021. 2.1.2

[95] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. Finding software license violations through binary code clone detection. In *8th Working Conference on Mining Software Repositories*, MSR, 2011. 4.2.1

[96] Tejun Heo. Control group v2. `https://www.kernel.org/doc/Documentation/cgroup-v2.txt`, 2015. 4.3.5

[97] David Hodson. Nest learning thermostat 2nd generation teardown. `https://www.ifixit.com/Teardown/Nest+Learning+Thermostat+2nd+Generation+Teardown/13818`, 2020. 1.1, 4.2.2

[98] James Hong, Amit Levy, Laurynas Riliskis, and Philip Levis. Don't talk unless i say so! securing the Internet of things with default-off networking. In *3rd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018. 2.2.1, 3.7, 4.1, 4.3.1, 4.8

[99] hostapd. hostapd. `https://w1.fi/hostapd/`, 2020. 4.6.1

[100] Weizhe Hua, Muhammad Umar, Zhiru Zhang, and G Edward Suh. Guardnn: secure accelerator architecture for privacy-preserving deep learning. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 349–354, 2022. 2.3.1

[101] Qinlong Huang, Yixian Yang, and Licheng Wang. Secure data access control with ciphertext update and computation outsourcing in fog computing for internet of things. *IEEE Access*, 5:12941–12950, 2017. doi: 10.1109/ACCESS.2017.2727054. 2.1.2, 3, 3.1

[102] Zhicong Huang, Wen-jie Lu, Cheng Hong, and Jiansheng Ding. Cheetah: Lean and fast secure {Two-Party} deep neural network inference. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 809–826, 2022. 5.1

[103] Ahmad Ibrahim, Ahmad-Reza Sadeghi, and Gene Tsudik. Us-aid: Unattended scalable attestation of iot devices. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*, pages 21–30. IEEE, 2018. 3.7

[104] IFTTT. IFTTT. `https://www.ifttt.com`, 2020. 4.6.2

[105] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *21st Network and Distributed Security Symposium*, Feb 2017. 2.1.2, 3, 3.1

[106] Xiaoqian Jiang, Miran Kim, Kristin Lauter, and Yongsoo Song. Secure outsourced matrix

computation and application to neural networks. In *Proceedings of the 2018 ACM SIGSAC conference on computer and communications security*, pages 1209–1222, 2018. 2.3.3, 5.1

[107] Haojian Jin. *Modular Privacy Flows: A Design Pattern for Data Minimization*. PhD thesis, Carnegie Mellon University, 2022. 6.2

[108] Haojian Jin, Gram Liu, David Hwang, Swarun Kumar, Yuvraj Agarwal, and Jason I Hong. Peekaboo: A hub-based approach to enable transparency in data processing within smart homes. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 303–320. IEEE, 2022. 3.7, 6.2

[109] Michael Jones, John Bradley, and Nat Sakimura. Rfc 7519: Json web token (jwt), 2015. 3.3.3

[110] Chiraag Juvekar, Vinod Vaikuntanathan, and Anantha Chandrakasan. {GAZELLE}: A low latency framework for secure neural network inference. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1651–1669, 2018. 2.3.3, 5.1, 5.3

[111] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, 2017. 5.1, 5.9.3

[112] Keybase. Keybase. `https://keybase.io/`, 2021. 3.7

[113] Dohyun Kim, Prasoon Patidar, Han Zhang, Abhijith Anilkumar, and Yuvraj Agarwal. Self-serviced iot: Practical and private iot computation offloading with full user control. *arXiv preprint arXiv:2205.04405*, 2022. 3.7

[114] Wonjung Kim, Seungchul Lee, Youngjae Chang, Taegyeong Lee, Inseok Hwang, and Junehwa Song. Hivemind: social control-and-use of IoT towards democratization of public spaces. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 467–482, 2021. 2.1.2

[115] Ronny Ko and James Mickens. DeadBolt: Securing IoT deployments. In *Applied Networking Research Workshop*, ANRW, 2018. 2.2.1, 4.3.1, 4.3.2

[116] Boyu Kuang, Anmin Fu, Shui Yu, Guomin Yang, Mang Su, and Yuqing Zhang. Esdra: An efficient and secure distributed remote attestation scheme for iot swarms. *IEEE Internet of Things Journal*, 6(5):8372–8383, 2019. 3.7

[117] Deepak Kumar, Kelly Shen, Benton Case, Deepali Garg, Galina Alperovich, Dmitry Kuznetsov, Rajarshi Gupta, and Zakir Durumeric. All things considered: An analysis of IoT devices on home networks. In *28th USENIX Security Symposium*, 2019. 4.1, 4.8

[118] Sam Kumar, Yuncong Hu, Michael P Andersen, Raluca Ada Popa, and David E. Culler. JEDI: Many-to-many end-to-end encryption and key delegation for iot. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1519–1536, Santa Clara, CA, August 2019. USENIX Association. ISBN 978-1-939133-06-9. URL `https://www.usenix.org/conference/usenixsecurity19/presentation/kumar-sam`. 2.1.2, 3, 3.1

[119] Butler Lampson, Martín Abadi, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*

(TOCS), 10(4):265–310, 1992. 3.2.1

[120] Gierad Laput, Yang Zhang, and Chris Harrison. Synthetic sensors: Towards general-purpose sensing. In *Proc. of the 2017 CHI Conference on Human Factors in Computing Systems*, CHI '17, page 3986–3999, New York, NY, USA, 2017. ACM. ISBN 9781450346559. doi: 10.1145/3025453.3025773. URL https://doi.org/10.1145/3025453.3025773. 5.1

[121] Gierad Laput, Karan Ahuja, Mayank Goel, and Chris Harrison. Ubicoustics: Plug-and-play acoustic activity recognition. In *Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology*, pages 213–224, 2018. 5.1

[122] Tobias Lauinger, Chaabane Abdelberi, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript libraries on the web. In *NDSS*, 2017. 4.1

[123] Tam Le and Matt W Mutka. Access control with delegation for smart home applications. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 142–147, 2019. 2.1.2, 3, 3.1

[124] The Security Ledger. Devices' UPnP service emerges as key threat to home IoT networks. https://securityledger.com/2019/03/devices-upnp-service-emerges-as-key-threat-to-home-iot-networks/, 2020. 4.8

[125] Xinyu Lei, Guan-Hua Tu, Chi-Yu Li, Tian Xie, and Mi Zhang. SecWIR: Securing smart home IoT communications via wi-fi routers with embedded intelligence. In *18th International Conference on Mobile Systems, Applications, and Services*, MobiSys, 2020. 2.2.1, 4.1, 4.3.2, 4.4

[126] Ding Li, Shuai Hao, William GJ Halfond, and Ramesh Govindan. Calculating source line level energy information for android applications. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 78–89, 2013. 3.6.1

[127] Ding Li, Shuai Hao, Jiaping Gui, and William GJ Halfond. An empirical study of the energy consumption of android applications. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 121–130. IEEE, 2014. 3.6.1

[128] Sheng Li, Jongsoo Park, and Ping Tak Peter Tang. Enabling sparse winograd convolution by native pruning. *arXiv preprint arXiv:1702.08597*, 2017. 5.5.2

[129] Chieh-Jan Mike Liang, Börje F. Karlsson, Nicholas D. Lane, Feng Zhao, Junbei Zhang, Zheyi Pan, Zhao Li, and Yong Yu. SIFT: Building an internet of safe things. In *14th International Conference on Information Processing in Sensor Networks*, IPSN, 2015. 2.2.2

[130] libsodium. A modern, portable, easy to use crypto library. https://libsodium.org/, 2021. 3.5

[131] Yingyan Lin, Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. Predictivenet: An energy-efficient convolutional neural network via zero prediction. In *2017 IEEE international symposium on circuits and systems (ISCAS)*, pages 1–4. IEEE, 2017. 5.5.2

[132] linux-containers. Linux containers. https://linuxcontainers.org/, 2020.

4.3.5

[133] Jian Liu, Mika Juuti, Yao Lu, and Nadarajah Asokan. Oblivious neural network predictions via minionn transformations. In *Proceedings of the 2017 ACM SIGSAC conference on computer and communications security*, pages 619–631, 2017. 2.3.3

[134] Tianyi Liu, Xiang Xie, and Yupeng Zhang. Zkcnn: Zero knowledge proofs for convolutional neural network predictions and accuracy. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 2968–2985, 2021. 5.1

[135] Shrirang Mare, Franziska Roesner, and Tadayoshi Kohno. Smart devices in Airbnbs: Considering privacy and security for both guests and hosts. *Proc. Priv. Enhancing Technol.*, 2020(2):436–458, 2020. 3.1

[136] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. Edge machine learning for ai-enabled iot devices: A review. *Sensors*, 20(9):2533, 2020. 5.1

[137] Ralph C Merkle. A digital signature based on a conventional encryption function. In *Conference on the theory and application of cryptographic techniques*, pages 369–378. Springer, 1987. 5.5.1

[138] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *2017 Internet Measurement Conference*, IMC, 2017. 4.6.2, 4.7.2

[139] MichMich. MagicMirror. `https://github.com/MichMich/MagicMirror`, 2020. 4.6.2

[140] Microsoft Azure. Azure Sphere. `https://azure.microsoft.com/en-us/services/azure-sphere/`, 2020. 1, 2.2.3, 4.1, 4.5.2

[141] Yisroel Mirsky, Tomer Doitshman, Yuval Elovici, and Asaf Shabtai. Kitsune: An ensemble of autoencoders for online network intrusion detection. In *Network and Distributed Systems Security (NDSS) Symposium*, 2018. 5.7.2

[142] Pratyush Mishra, Ryan Lehmkuhl, Akshayaram Srinivasan, Wenting Zheng, and Raluca Ada Popa. Delphi: A cryptographic inference service for neural networks. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2505–2522, 2020. 2.3.3, 5.1, 5.3, 1

[143] Fan Mo, Hamed Haddadi, Kleomenis Katevas, Eduard Marin, Diego Perino, and Nicolas Kourtellis. Ppfl: privacy-preserving federated learning with trusted execution environments. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, pages 94–108, 2021. 2.3.1, 3.7

[144] Payman Mohassel and Yupeng Zhang. Secureml: A system for scalable privacy-preserving machine learning. In *2017 IEEE symposium on security and privacy (SP)*, pages 19–38. IEEE, 2017. 2.3.3, 5.2.2, 5.4

[145] Murat Moran and Dan S Wallach. Verification of star-vote and evaluation of fdr and proverif. In *International Conference on Integrated Formal Methods*, pages 422–436. Springer, 2017. 3.4.4

[146] Motion. Motion. `https://motion-project.github.io/`, 2021. 3.4, 3.6.2

[147] Mycroft. Mycroft – the open source privacy-focused voice assistant. https://mycroft.ai/, 2021. 3.4, 3.6.2

[148] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021. 5.8

[149] Asuka Nakajima, Takuya Watanabe, Eitaro Shioji, Mitsuaki Akiyama, and Maverick Woo. A pilot study on consumer IoT device vulnerability disclosure and patch release in japan and the united states. In *2019 ACM Asia Conference on Computer and Communications Security*, AsiaCCS, 2019. 4.1, 4.2

[150] Radius Network. Android beacon library. https://altbeacon.github.io/android-beacon-library/index.html, 2021. 3.5

[151] Hung Nguyen, Radoslav Ivanov, Linh T.X. Phan, Oleg Sokolsky, James Weimer, and Insup Lee. LogSafe: Secure and scalable data logger for IoT devices. In *3rd ACM/IEEE International Conference on Internet-of-Things Design and Implementation*, IoTDI, 2018. 2.2.2

[152] Ivan De Oliveira Nunes, Karim Eldefrawy, Norrathep Rattanavipanon, Michael Steiner, and Gene Tsudik. {VRASED}: A verified {Hardware/Software} {Co-Design} for remote attestation. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1429–1446, 2019. 3.7

[153] NVIDIA. Determinism in deep learning. https://developer.nvidia.com/gtc/2019/video/s9911, 2019. 5.7.2

[154] NVIDIA. Confidential computing. https://www.nvidia.com/en-us/data-center/solutions/confidential-computing/, 2023. 2.3.1

[155] Open Connectivity Foundation. Upnp standards and architecture. https://openconnectivity.org/developer/specifications/upnp-resources/upnp, 2021. 3.3.3

[156] OpenSSL. Release strategy. https://www.openssl.org/policies/releasestrat.html, 2020. 4.3.2, 4.3.2

[157] openssl-changelog. OpenSSL changelog. https://www.openssl.org/news/changelog.html, 2020. 4.2.2, 4.3.2

[158] Nouha Oualha and Kim Thuat Nguyen. Lightweight attribute-based encryption for the internet of things. In *2016 25th International Conference on Computer Communication and Networks (ICCCN)*, pages 1–6, 2016. doi: 10.1109/ICCCN.2016.7568538. 2.1.2, 3, 3.1

[159] Shijia Pan, Carlos Ruiz, Jun Han, Adeola Bannis, Patrick Tague, Hae Young Noh, and Pei Zhang. Universense: IoT device pairing through heterogeneous sensing signals. In *Proceedings of the 19th International Workshop on Mobile Computing Systems & Applications*, pages 55–60, 2018. 3.3.2

[160] Nicolas Papernot, Patrick McDaniel, Arunesh Sinha, and Michael Wellman. Towards the science of security and privacy in machine learning. *arXiv preprint arXiv:1611.03814*,

2016. 5.1

[161] Simon Parkin, Trupti Patel, Isabel Lopez-Neira, and Leonie Tanczer. Usability analysis of shared device ecosystem security: Informing support for survivors of iot-facilitated tech-abuse. In *Proceedings of the New Security Paradigms Workshop*, 2019. 2.1.1

[162] Bryan Parno. Bootstrapping trust in a" trusted" platform. In *HotSec*, 2008. 3.7

[163] Bryan Parno, Jonathan M McCune, and Adrian Perrig. Bootstrapping trust in commodity computers. In *2010 IEEE Symposium on Security and Privacy*, pages 414–429. IEEE, 2010. 3.7

[164] Bryan Parno, Jon Howell, Craig Gentry, and Mariana Raykova. Pinocchio: Nearly practical verifiable computation. In *2013 IEEE Symposium on Security and Privacy*, pages 238–252. IEEE, 2013. 2.3.2

[165] particlel-device-os. Particle device OS. `https://www.particle.io/device-os/`, 2020. 1, 2.2.3, 4.1, 4.5.2

[166] phodal/awesome-iot. A collaborative list of great resources about IoT framework, library, OS, platform. `https://github.com/phodal/awesome-iot#library`, 2020. 4.1

[167] Raspberry Pi. Issue: cpuset disabled #1950. `https://github.com/raspberrypi/linux/issues/1950`, 2020. 4.3.5

[168] Otto Julio Ahlert Pinno, Andre Ricardo Abed Gregio, and Luis C. E. De Bona. Controlchain: Blockchain as a central enabler for access control authorizations in the iot. In *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, pages 1–6, 2017. doi: 10.1109/GLOCOM.2017.8254521. 2.1.2, 3, 3.1

[169] Rishabh Poddar, Ganesh Ananthanarayanan, Srinath Setty, Stavros Volos, and Raluca Ada Popa. Visor:{Privacy-Preserving} video analytics as a cloud service. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1039–1056, 2020. 2.3.1

[170] Qualcomm. With help from qualcomm technologies, the irobot roomba i7+ robot vacuum delivers more intelligent, effective cleaning. `https://www.qualcomm.com/news/onq/2018/12/help-qualcomm-technologies-irobot-roomba-i7-robot-vacuum-delivers-more`, 2018. 5.1, 5.9.1

[171] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. securetf: a secure tensorflow framework. In *Proceedings of the 21st International Middleware Conference*, pages 44–59, 2020. 2.3.1

[172] Hany Ragab, Alyssa Milburn, Kaveh Razavi, Herbert Bos, and Cristiano Giuffrida. Crosstalk: Speculative data leaks across cores are real. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 1852–1867. IEEE, 2021. 2.3.1

[173] raspberry-pi-4-spec. Raspberry Pi 4 specification. `https://www.raspberrypi.org/products/raspberry-pi-4-model-b/specifications/`, 2020. 4.7.3

[174] Dark Reading. Over 80% of medical imaging devices run on outdated operating systems. `https://www.darkreading.com/iot/over-80--of-medical-imaging-devices-run-on-outdated-operating-systems/d/d-id/1337273`,

2020. 4.1

[175] Red Hat. What is the maximum number of interface aliases supported in Red Hat Enterprise Linux? https://access.redhat.com/solutions/40500, 2020. 4.7.3

[176] Muzammil Abdul Rehman and Paul Grosu. RPC is not dead: Rise, fall and the rise of remote procedure calls. http://dist-prog-book.com/chapter/1/rpc.html, 2017. 4.8

[177] M Sadegh Riazi, Christian Weinert, Oleksandr Tkachenko, Ebrahim M Songhori, Thomas Schneider, and Farinaz Koushanfar. Chameleon: A hybrid secure computation framework for machine learning applications. In *Proceedings of the 2018 on Asia conference on computer and communications security*, pages 707–721, 2018. 5.2.2, 5.4

[178] M Sadegh Riazi, Kim Laine, Blake Pelton, and Wei Dai. Heax: An architecture for computing on encrypted data. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 1295–1309, 2020. 2.3.3

[179] Brian Rogers, Siddhartha Chhabra, Milos Prvulovic, and Yan Solihin. Using address independent seed encryption and bonsai merkle trees to make secure processors os-and performance-friendly. In *40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 2007)*, pages 183–196. IEEE, 2007. 5.5.1

[180] RTInsights. Malware attacks IoT devices running windows 7. https://www.rtinsights.com/malware-iot-windows-7/, 2020. 4.1

[181] RTInsights. IoT devices still exposed, vast majority of traffic unencrypted. https://www.rtinsights.com/iot-security-remains-lacklustre/, 2020. 1, 4.1

[182] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015. doi: 10.1007/s11263-015-0816-y. 5.7.2, 5.9.1

[183] Nikola Samardzic, Axel Feldmann, Aleksandar Krastev, Srinivas Devadas, Ronald Dreslinski, Christopher Peikert, and Daniel Sanchez. F1: A fast and programmable accelerator for fully homomorphic encryption. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 238–252, 2021. 2.3.3

[184] Samsung SmartThings. Direct-connected device SDK. https://smartthings.developer.samsung.com/docs/devices/direct-connected-devices/overview.html, 2020. 2.2.3, 4.5.2, 4.6.2

[185] Samsung SmartThings. SmartThings device SDK. https://github.com/SmartThingsCommunity/st-device-sdk-c, 2020. 4.6.2

[186] Felix Schuster, Manuel Costa, Cédric Fournet, Christos Gkantsidis, Marcus Peinado, Gloria Mainar-Ruiz, and Mark Russinovich. Vc3: Trustworthy data analytics in the cloud

using sgx. In *2015 IEEE symposium on security and privacy*, pages 38–54. IEEE, 2015. 3.7

[187] Roei Schuster, Vitaly Shmatikov, and Eran Tromer. Situational access control in the internet of things. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1056–1073, 2018. 2.1.2, 3, 3.1

[188] Senrio. 400,000 publicly available IoT devices vulnerable to single flaw. `https://blog.senr.io/blog/400000-publicly-available-iot-devices-vulnerable-to-single-flaw`, 2016. 1, 4.1

[189] Amazon Web Services. Wyze case study - AWS. `https://aws.amazon.com/solutions/case-studies/wyze/`, 2022. 1, 5.1

[190] Mohit Sethi, Elena Oat, Mario Di Francesco, and Tuomas Aura. Secure bootstrapping of cloud-managed ubiquitous displays. In *2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp, 2014. 4.4

[191] Hossein Shafagh, Lukas Burkhalter, Anwar Hithnawi, and Simon Duquennoy. Towards blockchain-based auditable storage and sharing of IoT data. In *Proceedings of the 2017 on Cloud Computing Security Workshop*, pages 45–50, 2017. 2.1.2, 3, 3.1

[192] Hossein Shafagh, Anwar Hithnawi, Lukas Burkhalter, Pascal Fischli, and Simon Duquennoy. Secure sharing of partially homomorphic encrypted IoT data. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, pages 1–14, 2017. 2.1.2, 3, 3.1

[193] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. Droplet: Decentralized authorization and access control for encrypted data streams. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2469–2486, 2020. 2.1.2, 3, 3.1

[194] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979. 1.2, 3.3, 3.3.4

[195] Shu Shi, Varun Gupta, Michael Hwang, and Rittwik Jana. Mobile vr on edge cloud: A latency-driven design. In *Proceedings of the 10th ACM Multimedia Systems Conference*, MMSys '19, page 222–231, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450362979. doi: 10.1145/3304109.3306217. URL `https://doi.org/10.1145/3304109.3306217`. 6.2

[196] Amit Kumar Sikder, Leonardo Babun, Z Berkay Celik, Abbas Acar, Hidayet Aksu, Patrick McDaniel, Engin Kirda, and A Selcuk Uluagac. Kratos: multi-user multi-device-aware access control system for the smart home. In *Prelavantroceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, 2020. 2.1.2

[197] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 5.9, 5.9.1

[198] Anna Kornfeld Simpson, Franziska Roesner, and Tadayoshi Kohno. Securing vulnerable home IoT devices with an in-hub security manager. In *2017 IEEE International Conference on Pervasive Computing and Communications Workshops*, PerCom, 2017. 2.2.1

[199] A.S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In *Proceedings of the Euteco '88 Conference*, pages 775–783, 1988. 4.8

[200] TensorFlow. Difference in output between cpu and gpu. `https://github.com/tensorflow/tensorflow/issues/19200`, 2018. 5.7.2

[201] TensorFlow. Dependence of output on batch size. `https://github.com/tensorflow/tensorflow/issues/25625`, 2019. 5.7.2

[202] TensorFlow. Tensorflow. `https://www.tensorflow.org/`, 2022. 5.8

[203] TensorFlow. Tensorflow lite. `https://www.tensorflow.org/lite`, 2022. 5.8

[204] Thread Group. Thread. `https://www.threadgroup.org/`, 2021. 3.3.3

[205] Youliang Tian, Ta Li, Jinbo Xiong, Md Zakirul Alam Bhuiyan, Jianfeng Ma, and Changgen Peng. A blockchain-based machine learning framework for edge services in iiot. *IEEE Transactions on Industrial Informatics*, 18(3):1918–1929, 2021. 5.5.1

[206] tomoyo. TOMOYO linux. `https://tomoyo.osdn.jp/index.html.en`, 2020. 4.3.4, 4.6.1

[207] tool-smem. smem(8) - linux man page. `https://linux.die.net/man/8/smem`, 2020. 4.7.3

[208] Florian Tramer and Dan Boneh. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287*, 2018. 2.3.1, 3.7, 5.1, 5.3, 5.5.1, 5.9.3

[209] Rahmadi Trimananda, Ali Younis, Bojun Wang, Bin Xu, Brian Demsky, and Guoqing Xu. Vigilia: Securing smart home edge computing. In *2018 IEEE/ACM Symposium on Edge Computing*, SEC, 2018. 2.2.2, 3.7, 4.1, 4.3.4

[210] Emily Tseng, Rosanna Bellini, Nora McDonald, Matan Danos, Rachel Greenstadt, Damon McCoy, Nicola Dell, and Thomas Ristenpart. The tools and tactics used in intimate partner surveillance: An analysis of online infidelity forums. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1893–1909. USENIX Association, August 2020. ISBN 978-1-939133-17-5. 2.1.1

[211] Random Nerd Tutorials. 70+ ESP32 projects, tutorials and guides with Arduino IDE. `https://randomnerdtutorials.com/projects-esp32/`, 2020. 4.6.2

[212] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the intel {SGX} kingdom with transient {Out-of-Order} execution. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 991–1008, 2018. 2.3.1

[213] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017. 4.8

[214] Steve Vinoski. Convenience over correctness. *IEEE Internet Computing*, 12(4):89–92, 2008. 4.8

[215] Stavros Volos, Kapil Vaswani, and Rodrigo Bruno. Graviton: Trusted execution environments on {GPUs}. In *13th USENIX Symposium on Operating Systems Design and*

Implementation (OSDI 18), pages 681–696, 2018. 2.3.1

[216] Jim Waldo, Geoff Wyant, Ann Wollrath, and Sam Kendall. A note on distributed computing. In *International Workshop on Mobile Object Systems*, pages 49–64, 1996. 4.8

[217] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 611–626, 2016. 1.2, 3, 3.3, 3.3.1, 3.3.4, 3.3.6, 3.5

[218] Chenkai Weng, Kang Yang, Xiang Xie, Jonathan Katz, and Xiao Wang. Mystique: Efficient conversions for zero-knowledge proofs with applications to machine learning. In *USENIX Security Symposium*, pages 501–518, 2021. 5.1

[219] Wikipedia. OpenSSL - major version releases. `https://en.wikipedia.org/wiki/OpenSSL#Major_version_releases`, 2020. 4.3.2

[220] Wikipedia. IEEE 802.11i-2004. `https://en.wikipedia.org/wiki/IEEE_802.11i-2004`, 2020. 4.4

[221] Ann Wollrath, Roger Riggs, and Jim Waldo. A distributed object model for the javaˆ tˆ m system. *Computing Systems*, 9:265–290, 1996. 4.8

[222] Rich Wolski, Chandra Krintz, Fatih Bakir, Gareth George, and Wei-Tsung Lin. CSPOT: Portable, multi-scale functions-as-a-service for IoT. In *4th ACM/IEEE Symposium on Edge Computing*, SEC, 2019. 2.2.2

[223] Wyze. Wyze cam. `https://www.wyze.com/products/wyze-cam`, 2022. 5.1

[224] Wyze. Wyze cam plus subscription. `https://services.wyze.com/detail/camplus`, 2022. 1, 5.1

[225] Miguel G. Xavier, Marcelo V. Neves, Fabio D. Rossi, Tiago C. Ferreto, Timoteo Lange, and Cesar A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, 2013. 4.3.5

[226] Bin Cedric Xing, Mark Shanahan, and Rebekah Leslie-Hurd. Intel® software guard extensions (intel® sgx) software support for dynamic memory allocation inside an enclave. *Proceedings of the Hardware and Architectural Support for Security and Privacy 2016*, pages 1–9, 2016. 5.5.1

[227] Guowen Xu, Hongwei Li, Hao Ren, Jianfei Sun, Shengmin Xu, Jianting Ning, Haomiao Yang, Kan Yang, and Robert H Deng. Secure and verifiable inference in deep neural networks. In *Annual Computer Security Applications Conference*, pages 784–797, 2020. 2.3.3

[228] Yaxing Yao, Justin Reed Basdeo, Oriana Rosata Mcdonough, and Yang Wang. Privacy perceptions and designs of bystanders in smart homes. *Proceedings of the ACM on Human-Computer Interaction*, 3(CSCW):1–24, 2019. 2.1.1, 3.1

[229] Tianlong Yu, Vyas Sekar, Srinivasan Seshan, Yuvraj Agarwal, and Chenren Xu. Handling a trillion (unfixable) flaws on a billion devices: Rethinking network security for the internet-of-things. In *14th ACM Workshop on Hot Topics in Networks*, HotNets, 2015. 1,

2.2.1, 4.1

[230] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, XiaoFeng Wang, and Yuqing Zhang. Shattered chain of trust: Understanding security risks in cross-cloud IoT access delegation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1183–1200, 2020. 2.1.2

[231] Gina Yuan, David Mazières, and Matei Zaharia. Extricating iot devices from vendor infrastructure with karl. *arXiv preprint arXiv:2204.13737*, 2022. 3.7

[232] ZDNet. CallStranger vulnerability lets attacks bypass security systems and scan LANs. https://www.zdnet.com/article/callstranger-vulnerability-lets-attacks-bypass-security-systems-and-scan-lans/, 2020. 4.1, 4.8

[233] ZDNet. Hacker leaks passwords for more than 500,000 servers, routers, and IoT devices. https://www.zdnet.com/article/hacker-leaks-passwords-for-more-than-500000-servers-routers-and-iot-devices/, 2020. 1, 4.1

[234] ZDNet. Ripple20 vulnerabilities will haunt the IoT landscape for years to come. https://www.zdnet.com/article/ripple20-vulnerabilities-will-haunt-the-iot-landscape-for-years-to-come/, 2020. 4.1

[235] 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)*, 2019. 2.1.1

[236] Zeroconf. Zero configuration networking. http://www.zeroconf.org/, 2021. 3.3.3

[237] Han Zhang. synergylabs/iot-capture. https://github.com/synergylabs/iot-capture, 2021. 4, 4.6.1

[238] Han Zhang. synergylabs/teo-release. https://github.com/synergylabs/TEO-release, 2022. 3, 3.5

[239] Han Zhang, Abhijith Anilkumar, Matt Fredrikson, and Yuvraj Agarwal. Capture: Centralized library management for heterogeneous {IoT} devices. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 4187–4204, 2021. 3.7, 4

[240] Han Zhang, Yuvraj Agarwal, and Matt Fredrikson. TEO: Ephemeral ownership for iot devices to provide granular data control. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 302–315, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391856. doi: 10.1145/3498361.3539774. URL https://doi.org/10.1145/3498361.3539774. 3

[241] Han Zhang, Yuvraj Agarwal, and Matt Fredrikson. Protecting user data through ephemeral ownership of iot devices. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, MobiSys '22, page 620–621, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391856. doi: 10.1145/3498361.3538664. URL https://doi.org/10.1145/3498361.3538664. 3

[242] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. HoMonit: Monitoring smart home apps from encrypted traffic. In *2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS, 2018. 2.2.1, 4.8

[243] Wensheng Zhang and Trent Muhr. TEE-based selective testing of local workers in federated learning systems. In *2021 18th International Conference on Privacy, Security and Trust (PST)*, pages 1–6. IEEE, 2021. 2.3.2, 5.5.1

[244] Xiaoli Zhang, Fengting Li, Zeyu Zhang, Qi Li, Cong Wang, and Jianping Wu. Enabling execution assurance of federated learning at untrusted participants. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 1877–1886. IEEE, 2020. 2.3.1

[245] Lingchen Zhao, Qian Wang, Cong Wang, Qi Li, Chao Shen, and Bo Feng. Veriml: Enabling integrity assurances and fair payments for machine learning as a service. *IEEE Transactions on Parallel and Distributed Systems*, 32(10):2524–2540, 2021. 2.3.2

[246] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11): 2348–2359, 2018. 5.1

[247] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. Opaque: An oblivious and encrypted distributed analytics platform. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 283–298, 2017. 3.7

[248] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In *28th USENIX Security Symposium*, 2019. 4.2

[249] Jianping Zhu, Rui Hou, XiaoFeng Wang, Wenhao Wang, Jiangfeng Cao, Boyan Zhao, Zhongpu Wang, Yuhui Zhang, Jiameng Ying, Lixin Zhang, et al. Enabling rack-scale confidential computing using heterogeneous trusted execution environment. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1450–1465. IEEE, 2020. 2.3.1