Peekaboo: A Hub-Based Approach to Enable Transparency in Data Processing within Smart Homes

Haojian Jin, Gram Liu, David Hwang, Swarun Kumar, Yuvraj Agarwal, Jason I. Hong
Carnegie Mellon University

***Abstract*—We present Peekaboo, a new privacy-sensitive architecture for smart homes that leverages an in-home hub to pre-process and minimize outgoing data in a structured and enforceable manner *before* sending it to external cloud servers. Peekaboo's key innovations are (1) abstracting common data pre-processing functionality into a small and fixed set of chainable operators, and (2) requiring that developers explicitly declare desired data collection behaviors (e.g., data granularity, destinations, conditions) in an application manifest, which also specifies how the operators are chained together. Given a manifest, Peekaboo assembles and executes a pre-processing pipeline using operators pre-loaded on the hub. In doing so, developers can collect smart home data on a need-to-know basis; third-party auditors can verify data collection behaviors; and the hub itself can offer a number of centralized privacy features to users across apps and devices, without additional effort from app developers. We present the design and implementation of Peekaboo, along with an evaluation of its coverage of smart home scenarios, system performance, data minimization, and example built-in privacy features.**

I. Introduction

For many smart home products - such as smart speakers, cameras, and thermostats - the "brains" of these systems are typically in the cloud. However, a key concern with cloud-based software architectures is data privacy [73]. It is challenging for users to have any assurances of privacy *after* their sensitive data leaves the confines of their home [56]. Further, it is challenging for companies to *legitimately avoid collecting unnecessary smart home data while also re-assuring users and independent auditors that this is indeed the case*.

For example, imagine a developer of a smart TV claims to only send aggregated viewing history data to their servers once a week. **How can outsiders validate this claim, since the hardware, firmware, and backend servers are proprietary black-boxes?** Today, an independent auditor would need to use arduous reverse engineering techniques to validate devices' data collection behavior [62]. One alternative is for the device to do everything locally without sending any data [4], though given that many devices only have basic CPU and storage capabilities, this approach severely limits the functionality they can provide. Doing everything locally also limits many kinds of rudimentary analytics that users might find acceptable, e.g. developers may want to know how many hours the TV is on per week. As another alternative, the device can aggregate or denature the data itself before sending it out [74], [78], but again there is no easy way to verify this behavior. Yet another alternative is to leverage new mechanisms for trusted cloud computing [27], [43], [48]. However, these approaches are challenging for users and auditors to understand and verify, and do not necessarily perform data minimization before sensitive data leaves one's control.

This paper introduces Peekaboo, a new privacy-sensitive architecture for developers to build smart home apps. Peekaboo has three key ideas. First, app developers must declare all intended data collection behaviors in a text-based manifest (see Fig. 1a), including under what conditions data will be sent outside of the home to cloud services, where that data is being sent to, and the granularity of the data itself. Second, to specify these behaviors, developers choose from a small and **fixed** set of operators with well-defined semantics, authoring a stream-oriented pipeline similar to Unix pipes. This pipeline pre-processes raw data from IoT devices in the home (e.g. sensor data or usage history) into the granularity needed by the cloud service. Third, an in-home trusted Peekaboo hub mediates between all devices in the home and the outside Internet. This hub enforces the declared behaviors in the manifests, and also locally runs all of the operators specified in these manifests to transform raw data before it is relayed to any cloud services. Combined, these ideas make it so that developers can reduce data collection by running pre-processing tasks on the in-home trusted hub, and users and third-party auditors can inspect data behaviors by analyzing these manifests as well as any actual data flows. Our approach also facilitates a number of privacy features that can be supported by the hub itself, such as adding additional conditions or transformations before data flows out, or transforming parts of the manifest into natural language statements to make it easier for lay people to understand what data will be sent out, when, and to where.

For example, the "HelloVisitor" app (Fig. 1) identifies visitors using faces in images. Today's video doorbells often send captured raw photos to the cloud when they detect a scene change. By applying a pre-processing pipeline (i.e., face detection and image cropping), HelloVisitor can avoid sending images of people or vehicles simply passing by, thus minimizing data egress to just what the app needs to operate.

Peekaboo's architecture is based on an analysis (§III) we conducted of 200+ smart home scenarios drawn from the research literature and design fiction interviews we conducted. This analysis led to two insights. First, many apps do not need raw sensor or log data, but rather a transformed or refined version. Second, while it cannot support all smart home scenarios, our approach of using a small and fixed set of operators can support a surprisingly large number of scenarios.

We implemented a Peekaboo manifest authoring tool on top of Node-Red, a mature, web-based, visual programming platform [35]. We developed the operators as a set of Node-Red

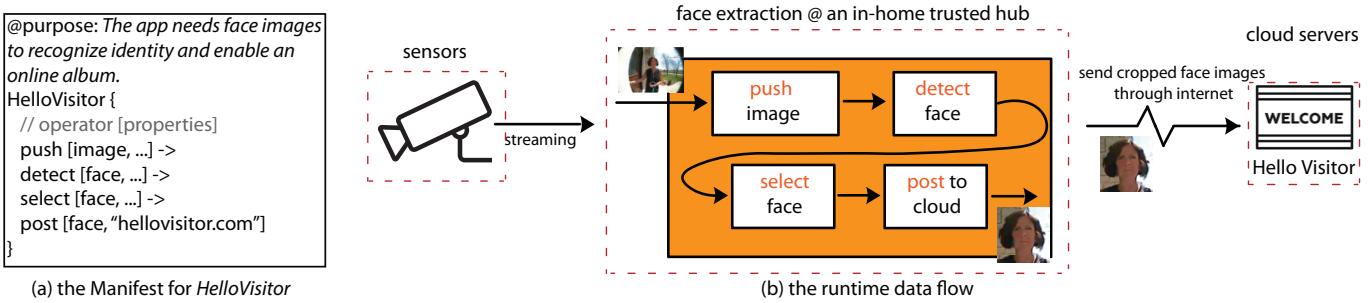

Figure 1: The architecture of the "HelloVisitor" app. A developer creates a manifest making use of four chainable operators (i.e., *push*, *detect*, *select*, and *post*) and adds a text string to specify the purpose. Once deployed, a camera streams the video to the hub, which pre-processes the video so that only portions of images with detected faces are sent to cloud servers. Since the semantics of each operator is known, it is easy to analyze the privacy-related behavior of apps (e.g. "this app only sends images of faces to HelloVisitor.com") and modify it if desired.

compatible building blocks. Further, we built the Peekaboo runtime, which parses a manifest, retrieves raw sensor or log data from IoT devices, sets up a data pipeline by assembling a chain of operators, and streams the data through this pipeline. Note that Peekaboo operators are device-agnostic, and rely on runtime drivers to handle heterogeneous device APIs (similar to HomeOS [18]). We deployed the Peekaboo runtime using a Raspberry Pi connected to a TPU accelerator.

We conducted detailed experiments to validate the design of Peekaboo. To understand the range and limitations of its architecture, we first implemented 68 different manifests to cover over 200 use cases and analyzed the types of pre-processing in these manifests (§VII-A). We then used three example manifests to demonstrate the feasibility of using simple algorithms to reduce privacy risks while having little impact on utility (§VII-B). We built five end-to-end Peekaboo apps, covering 5 data types (video, image, audio, tabular, and scalar) and used these apps to evaluate system performance. We also evaluated the scalability of our low-cost Rasberry Pi setup ($100), showing it can support more than 25 inference tasks and 100 filtering transformations per second (§VII-C). Finally, we demonstrated how Peekaboo's architecture, and the hub specifically, can support three kinds of privacy features across all apps (§VII-D). Specifically, we show how a static analysis tool can generate natural language descriptions and privacy nutrition labels [21] of behaviors based on operator pipelines, how an arbitrary manifest can be extended with time-based scheduling features, and how rate limiting can be easily added to a data flow.

Our paper focuses on introducing Peekaboo's architecture and demonstrating its mechanisms. We defer a user evaluation of the usability of its privacy mechanisms to future work. More specifically, this paper makes the following contributions:

- A novel software architecture, Peekaboo, that helps developers collect sensitive smart home data in a fine-grained and flexible manner, while also making the process transparent, enforceable, centrally manageable, and extensible.
- A study of over 200 unique smart home use cases to design a taxonomy of reusable pre-processing operators that can feasibly be implemented at the Peekaboo hub to enforce privacy requirements.
- An end-to-end prototype implementation of a Peekaboo hub on a Raspberry PI platform with a TPU accelerator, which we plan to open source upon publication.
- A detailed evaluation of Peekaboo's expressiveness based on coverage of smart home scenarios, system performance, data minimization, and application-independent privacy features.

II. Peekaboo Design Overview

Peekaboo has three main components. The first is developer-specified **manifests** that declare all data pre-processing pipelines. Note that the simplest manifest can just describe getting data from IoT devices and sending it to the cloud if no pre-processing is needed. The second is a **fixed** set of reusable and chainable **operators** to specify these pipelines. The third is an in-home trusted **hub** that enforces these manifests and mediates access between edge devices and the wider Internet. We present more details about our design rationale and tradeoffs below.

Peekaboo's manifest is a natural evolution of Android permissions [23], [58] and Manufacturer Usage Description (MUD) whitelists [14] for IoT devices. In Android, developers must explicitly declare permissions in an app's manifest so that it can access protected resources (e.g., location or SMS messages). Similarly, MUDs allow IoT device makers to declare a device's intended communication patterns. The rationale is that many IoT devices are expected to communicate with only a few remote servers known *a priori*, and so declaring the device's behaviors allows the network to blacklist unknown traffic requests.

However, a weakness with Android's permission system is that it is binary all-or-nothing access. For instance, an app developer might need to access SMS messages from just one phone number for two-factor authentication, but Android only offers access to all SMS messages or none. Furthermore, while the developer can display text in the app or in a privacy policy that the app will only access messages from one source once, there is no easy way for a user to verify this behavior. One

solution is to offer many fine-grained permissions for every potential use case, but this would lead to an explosion of permissions that would be onerous for developers to program, unwieldy for users to configure, and complex for platform builders to support.

Peekaboo proposes an alternative approach, requiring developers to declare the data collection behavior inside a text-based manifest using operators, with the hub only allowing declared flows. With Peekaboo, a user can install a new smart home app by simply downloading a manifest to the hub rather than a binary. This approach offers more flexibility than permissions, as well as a mechanism for enforcement. It also offers users (and auditors) more transparency about a device's behavior, in terms of what data will flow out, at what granularity, where it will go, and under what conditions.

Threat model: We envision that future third-party developers can build and distribute sophisticated ubiquitous computing apps through smart home App Stores, similar to today's app stores for smartphones. However, these smart home app developers, similar to mobile app developers, might deliberately or inadvertently collect more data than is necessary. Peekaboo's goal is to limit data egress by such developers, while also making it easier for users and auditors to verify the intended behaviors and data practices of their IoT apps.

We make the following assumptions: (1) *Trusted Hub:* The Peekaboo Hub, developed by platform providers (e.g., Apple, Google), is trusted and uncompromised. (2) *IoT Devices:* We assume that the Peekaboo hub can isolate IoT devices similar to the MUD [14]. All actual IoT Devices are required to send data through the Peekaboo hub, and do not circumvent the Peekaboo hub, either through an independent or covert side-channel. (3) *Operators:* We assume that the operators that we have created are themselves secure and do not have vulnerabilities. The source code of operators will be made open source and allow for verification, audits, and updates as needed. We note that while our threat model assumes that devices within the home are trusted, it does *not* make a similar assumption about cloud services.

III. Analysis of Smart Home Scenarios

To inform the design of Peekaboo, we collected over 200 smart home use cases and examined the feasibility and trade-offs of doing pre-processing on a hub. Overall, our analysis suggests that, while it cannot support all smart home scenarios, pre-processing data locally before it leaves a smart home via a small set of operators can indeed support a wide range of smart home apps.

Collecting smart home use cases. Since smart home applications are not yet as popular as those on smartphones [45], we chose to study use cases beyond today's commercially available products, sourcing applications through three approaches. First, we conducted a literature review of sensor-based scenarios for smart homes (e.g., [47]). Second, we surveyed the mobile privacy research literature (e.g., [22], [39], [50]) to understand common data use patterns by mobile app developers since many of them apply to IoT scenarios. Third, we conducted design fiction interviews [20] with 7 participants (3 female, 4 male) across different age groups (min=21, max=60, avg=32) to broaden our set of use cases (See details in Appendix §A). In total, we gathered over 200 unique smart home use cases.

Method. We clustered similar use cases, resulting in 37 smart home apps spanning different sensors and locations in a home (Appendix Table V). We then analyzed what data these apps needed to operate. Specifically, we enumerated why these apps need to collect data (e.g., sensing, analytics, system diagnostics), analyzed potential requirements and constraints in using this data (e.g., compute load, proprietary algorithms, business models), and concluded with what we felt were reasonable outgoing data granularity for each data collection purpose. For example, suppose that the developer of a video doorbell app wants to build an online album for visitors to the home. A reasonable data requirement of this app is face images. We enumerated these data requirements for different purposes in Appendix Table VI, organizing them by data type.

Results. We make the following key observations. First, **most cloud services do not need raw sensor or log data**. For example, an app that uses images to detect potential water leaks likely only needs the parts of the original image showing the floor. A smart lighting app which needs to infer the brightness of a room from a camera only needs derived brightness values rather than the raw image. Table VI offers a quantitative view of data granularity requirements across various scenarios. Only 4 out of 37 camera usages require raw data, while the rest only need either derived or partial data. Beyond cameras, we found that only 19 out of 61 cloud services require raw data, while 8 were scalar values (e.g., a binary door status sensor).

Second, **most pre-processing functions share similar data-agnostic data actions**, suggesting the feasibility of using a small set of reusable operators to replace these repetitive pre-processing implementations. At first blush, the number of pre-processing functions appears huge, since we need to cover various data types, output data granularity (e.g., face, objects, audio events, abnormal data), and filter/transform operations (e.g., image cropping, audio spectrum extraction). However, by enumerating the 200+ smart home uses, we find that there exists consistent, simple, and data-agnostic semantics behind most of the functions. In Table VI, we categorize all the output data granularity into two classes. *Partial original data* is a subset of the raw sensor data in the original data representation, such as the parts of an image with faces extracted, audio recording segments containing human speech only, and a column in a table. *Derived data* is computed from the raw data, often resulting in a new data representation (e.g., the keypoints of a human pose). We use two verbs, "select" and "extract," to summarize all the functions that output "partial original" and "derived data," respectively.

Third, **many of these pre-processing functions are lightweight and non-proprietary**, suggesting that they can be run on a low-end smart home hub. For example, pre-

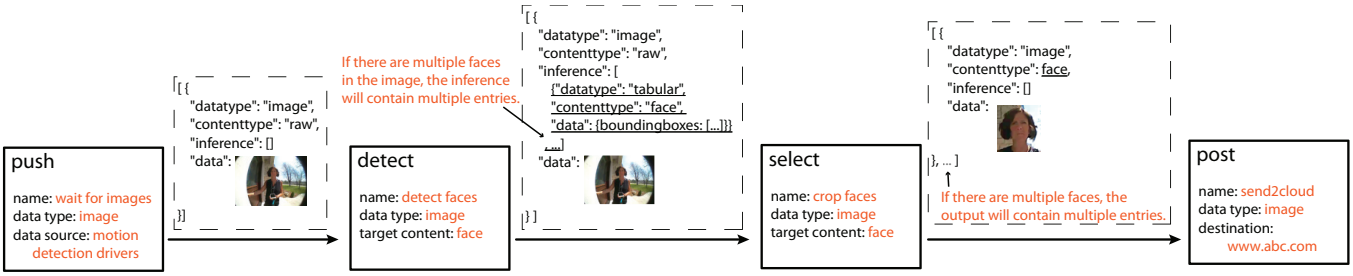

Figure 2: The hub program for "HelloVisitor" (Fig. 1). Operators come with a few configurable parameters and have clearly defined ways to interact with a uniform data model (dashed boxes). For example, the *detect* operator only modifies the "inference" field. The *select* operator modifies the "data" field based on the "inference" field. We highlight the affected data fields with underscores. Operators are open source to help with verification of their behaviors.

trained machine learning models like face detection, functions like inferring brightness, as well as database-like operations on tabular data (project, select, sum, average) are relatively straightforward, openly available, and fast to compute.

IV. Programming Manifests using Operators

We discuss the design of Peekaboo's manifest and operators, plus the rationale and tradeoffs behind our design. The manifest and operators need to be expressive enough to support a wide range of scenarios, easy to author for developers, easy to comprehend for auditors, and beneficial for privacy protection.

Peekaboo uses a Pipe-and-Filter software architecture [25], modeling a hub program as a set of connections between a set of stateless operators with known semantics. Figure 2 presents a data pre-processing pipeline from the *HelloVisitor* app, which can be abstracted into a directed acyclic graph (DAG) of operators. The first operator, a *push* operator named "wait for images", specifies the raw image retrieval behavior (e.g., pull v.s. push, frequency, resolution) and gets raw image from the Peekaboo runtime when there is a motion event. The second operator, *detect*, annotates the bounding boxes of faces inside the raw image. Next is "crop face", a *select* operator that crops the image to output a set of face images based on annotated bounding boxes. Finally, "send2cloud" is a *post* operator with outgoing network access, which posts cropped faces to a remote cloud service.

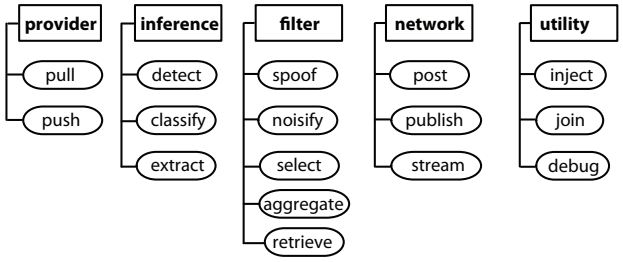

Figure 3: The taxonomy of Peekaboo operators. Each operator corresponds to a "verb" statement relative to the operator itself. For example, the *pull* operator pulls data from the hub runtime.

A. Abstracting Reusable Operators

At a high level, pre-processing pipelines do three things: collect raw data from edge devices, transform that data into the targeted granularity, and send the processed data to external servers. Although the exact desired data actions vary across use cases, the high-level data pre-processing semantics are surprisingly similar across data types. For example, a *noisify* data action denatures the original data slightly by imposing some noise, without changing the original data representation format, such as blurring an image, changing pitch/tempo of audio, or distorting numerical values by a small amount.

We used best practices in API design (e.g. [9]) to guide the design of these operators. We started with a few use cases, programmed them using our initial API, and iterated on the API as we expanded the supported use cases. Based on the data transform behaviors, we created sixteen operators grouped into five categories (Fig. 3): `provider`, `inference`, `filter`, `network`, and `utility`. Developers can specify the behavior of an operator by configuring its associated properties.

Inferring and filtering target content. In contrast to the binary all-or-nothing data access control, Peekaboo aims to enable a new fine-grained semantic-based data access control, which requires developers to declare when the app collects data (e.g., when a baby is crying) and what data content would be collected (e.g., face images, speech audios). To achieve this, we introduce two sets of operators: `inference` operators that can annotate data contents (e.g., the bounding boxes of faces, the key points of a body pose), and `filter` operators that filter based on the annotations. For example, *HelloVisitor* first uses a *detect* operator to annotate the bounding boxes of faces and then uses a *select* operator to crop the image based on annotated bounding boxes.

We summarize inference tasks into three primitives: *detecting* instances of objects, *classifying* the dominant content category, and *extracting* derived data (e.g., audio frequency spectrum). We also examined common privacy countermeasures [2], [19], [38], [44] to determine five types of *filter* operators: *spoof* for replacing the payload with an artificial replacement, *noisify* for injecting a configurable random noise, *select* for keeping partial raw data, *aggregate* for summarizing statistics, and *retrieve* for overwriting the payload data with derived data.

The abstractions of operators are somewhat analogous to abstract classes in object-oriented programming, and devel-

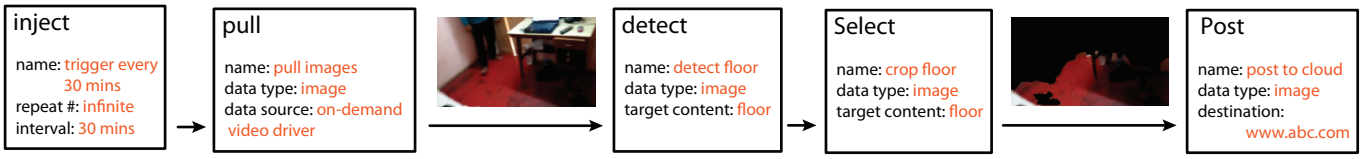

Figure 4: A water leak detection app pulls an image from the camera every 30 minutes, detects the floor area using image segmentation algorithms, and sends the image containing only the floor to the server. The app protects users' privacy by only sending the floor pixels to the cloud.

opers can specify the exact data transformation they want via configuring the properties of each operator. The runtime then maps the manifest specification to the concrete subclass implementations based on the properties. For example, the detect operators in Figs. 2 and 4 have different target content properties, so they are mapped to two different data transformations. Further, there may be multiple face detection operator implementations, and developers can let the runtime determine which one to use or specify one explicitly. We enumerate supported data transformations in Appendix Figure IV.

The key property for the inference and filter operators is *target content* (e.g., face in Fig. 2 and floor in Fig. 4). The target content is a type of semantic annotation supported by integrated inference algorithms, which can then be filtered accordingly using filter operators. Peekaboo currently provides several built-in state-of-the-art inference algorithms using pre-trained machine learning models, which support over 90 visual categories [10], [51] (e.g., face, person, floor, table), 632 audio classes [26] (e.g., baby crying), and many other individual categories (e.g., body pose [52], heart rate [8], audio frequency spectrum, brightness). Developers can choose among these options using a dropdown menu in the authoring interface (Fig. 10).

Note that a Peekaboo runtime only supports a fixed set of operators and property options enabled by the pre-loaded implementations. We do not allow operators to be dynamically loaded because we would not know the semantics of that operator, and it may have undesired behaviors.

Collecting raw and relaying processed data. To install a Peekaboo app, users need to bind the manifest to compatible devices, similar to how users install a SmartThings App [68] today. Developers have to specify required device drivers in `provider` operators (i.e., *push* and *pull*), so the hub runtime can determine compatible devices.

Here, *push* and *pull* represent two styles of data access: passively waiting for pushed data from drivers and actively pulling from drivers. For example, *HelloVisitor* (Fig. 2) starts with a *push* operator. When there is a significant change in the visual scene, the Peekaboo runtime sends an image to the *push* operator to trigger subsequent operators. In contrast, Fig 4 shows the hub program of a *water leak detection* app, which pulls an image from the camera every 30 minutes.

Finally, network operators are the only group of operators that can send data outside. Peekaboo currently has three operators: *post* for HTTP/S post, *publish* for MQTT Pub/Sub, and *stream* for RTSP video streaming. Developers can configure the `provider` and `network` operators to enable SSL connections between the IoT devices and the hub, and between the hub and remote servers, similar to the Network security configuration in Android [16].

B. Chaining Operators Together

In this section, we present more details on how operators are connected together.

Data model. Peekaboo uses a uniform data structure between operators (see Figures 2 and 4). The basic data structure (i.e., a Peekaboo data item) is a map, and the message transmitted between operators is an array of such items. An example map is shown below.

$$\{\text{“}datatype\text{”} : \text{“}video \mid image \mid audio \mid tabular \mid scalar\text{”},$$
$$\text{“}contenttype\text{”} : \text{“}raw \mid face \mid person \mid dog \mid speech \mid ...\text{”},$$
$$\text{“}inference\text{”} : [\{...\} \{...\} ...],$$
$$\text{“}data\text{”} : \{raw\ sensor\ data\},$$
$$\text{“}process\text{”} : \{device\ information, operation\ history\}\}$$

A Peekaboo data item only stores one unit of the data (e.g., an image, audio file, tabular row, or scalar value), while the inference field contains a list of annotations. Suppose the input of *HelloVisitor* is an image with multiple faces. Here, *detect face* will write multiple face annotations to the inference field, and each face annotation is a tabular Peekaboo data item. *crop face* will then generate a list of Peekaboo data items, where each corresponds to one face image (Fig. 2).

Composing pipelines. To build a data pre-processing pipeline, developers connect operators' outputs to others' inputs and assemble them into a directed acyclic graph. All operators, except *join* (described more below), take only input from one prior operator. This design avoids synchronization issues since the execution of operators is asynchronous.

In contrast, developers can connect an operator's output to multiple operators' inputs. The preceding operator creates a copy of the output message for each connection to avoid interference and potential multithreading problems.

Supporting more complex logic. Figs. 2 and 4 show two simple pipelines. However, these linear pipelines are insufficient for many applications. Imagine a baby monitoring app that only sends input audio to an external server when the hub program detects a baby crying (Fig. 5). With a linear pipeline this is infeasible since when we use a *retrieve* operator to check if the audio contains baby crying, the subsequent operators no longer have access to the original data. More fundamentally,

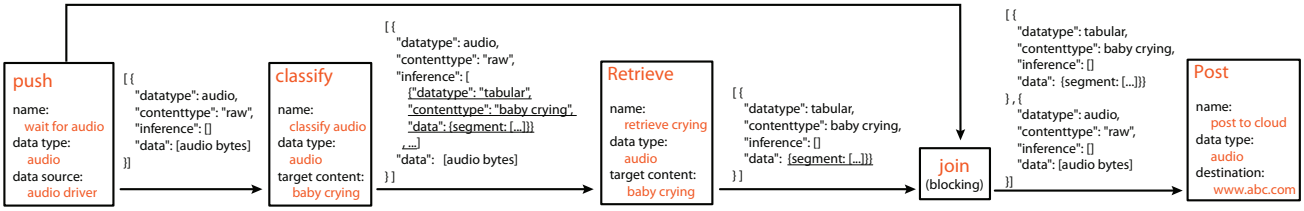

Figure 5: A baby monitoring app only sends audio containing a crying sound to the server. The *retrieve* operator replaces the "data" field with the "inference" field. The *join* operator merges the outputs from *wait for audio* and *retrieve crying*, passing its output onward when both input streams arrive. Finally, the *post* operator only sends the audio data item outside, since its data type parameter is "audio".

this is a common constraint of many dataflow programming models, where the data itself controls the program's flow.

We address this limitation by introducing the *join* operator to support AND, OR, NOT logic. *Join* is the only operator that can take input from multiple operators and fuse asynchronous data flows into one pipeline. A key property for *join* is whether it is blocking or non-blocking. A non-blocking *join* forwards incoming data whenever it becomes available, equivalent to an **OR** logical operator. In contrast, a blocking *join* only merges and forwards incoming messages if they meet specified criteria (e.g., all the incoming messages arrive within a small time window). Note that incoming messages can be from the same prior operator. For example, a blocking *join* can block the data flow until there are multiple "baby crying" events, to confirm that a certain accuracy level is met.

C. Error Handling & Debugging Support

It is possible for the flow between operators to be Peekaboo data items with different data types due to flow fusion. Without careful design, connecting arbitrary operators together may result in unpredictable errors. An essential property of our API design is that each operator is associated with a target data type. An operator only processes corresponding data items selectively, so the operators will not try to detect faces in a scalar value object.

Another important design issue is that *inference* and *filter* operators handle unmatched data types differently. Inference operators will leave unmatched items untouched, while filter operators will filter them out. This design strengthens Peekaboo's annotation capability since developers now can apply multiple inference operators to the same data item. Meanwhile, it enhances privacy as well: data can flow to the next operator only if the developer matches the data type explicitly.

To help with debugging, developers can append a *debug* operator after any operator, which will print output data to a console window. We also incorporated example data (e.g., test videos, photos, audios, and tabular data) as options of the *pull* operator and designed the *inject* operator to support both manual and interval triggers, so developers can quickly test pipelines within the editor without actual hardware.

V. Illustrating Pre-processing using Examples

We present three example manifests to illustrate the use of Peekaboo's APIs.

Smart TV logs. A smart TV developer is interested in collecting users' viewing history for advertising purposes. This smart TV stores viewing history in a simple tabular form. Here, we assume developers want to do better with respect to privacy, perhaps for legal compliance, market competition, or because hubs like Peekaboo are widely adopted in the future and companies want to assure customers that they are only collecting minimal data. Fig. 6 shows an example pipeline we built to show how a developer can compute an aggregate view of video consumption on the hub, thus sending out a less sensitive summary. This example also shows Peekaboo's support for database-like queries (e.g., SELECT, AGGREGATE, JOIN, WHERE), which also work in other tasks (e.g., counting people in an image).

Incognito voice assistant. While smart voice assistants are increasingly common, users have expressed multiple privacy concerns, such as nontransparent data collection [33] and identity tracking [11], [30], [33]. Fig. 7 presents a manifest of a smart voice assistant that we developed, which can mitigate these two privacy concerns. In contrast to current cloud-based implementations, this app is both transparent and offers a speech anonymization feature that protects users from exposing undesired voice fingerprints.

Beyond database-like queries, Peekaboo introduces two important extensions, content-based selection and explicit noise injection, to accommodate the smart home context and privacy-preserving goals. This example illustrates how the combination of inference and filter operators can filter out non-speech audio segments. Further, this example uses a *noisify* operator to hide speaker identity, which changes the tempos and pitch of the captured audio with a configurable random variation (e.g., 5%).

Productivity tracking. PC Applications like RescueTime [67] help users be more productive by helping them understand how they spend their time. As working from home becomes increasingly common, we envision a smart home version that track a person's productivity beyond PCs. Implementing such an app in a conventional cloud-based architecture can be worrisome due to privacy concerns similar to those in the voice assistant app. Fig. 8 presents an example pre-processing pipeline of a productivity tracking app using a camera, which transmits extracted poses to the cloud.

This example shows use of Peekaboo's flexible conditional flow control. Peekaboo operators support an intrinsic condition

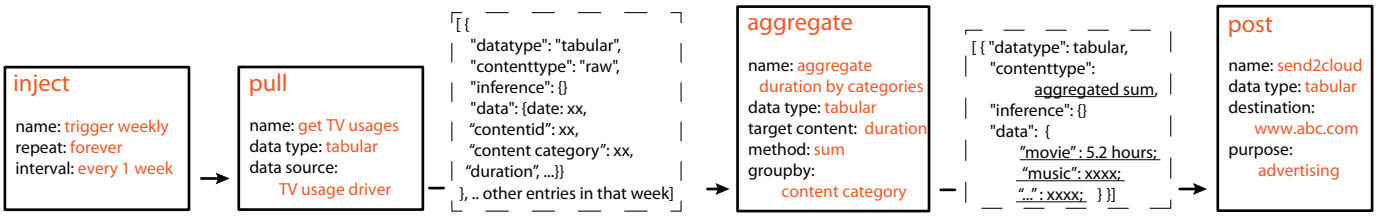

Figure 6: A SmartTV usage summary app queries users' watching patterns and uses the data to generate a weekly summary for video consumption across different channels and categories. The *aggregate* operator works similarly to SQL aggregation and replaces the value in the "data" field.

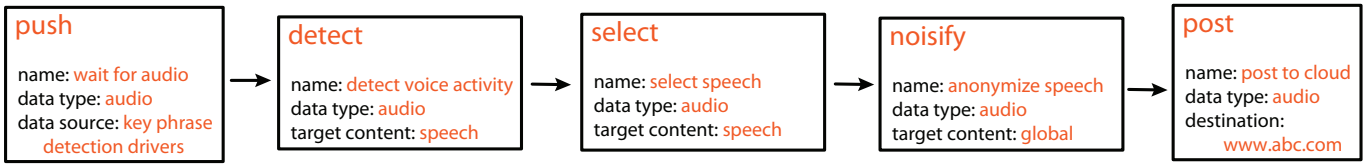

Figure 7: The incognito voice assistant only collects short audio segments containing speech and hides the speaker's identity by changing the pitch. This hub program can protect users' identity without breaking the speech recognition functionality.

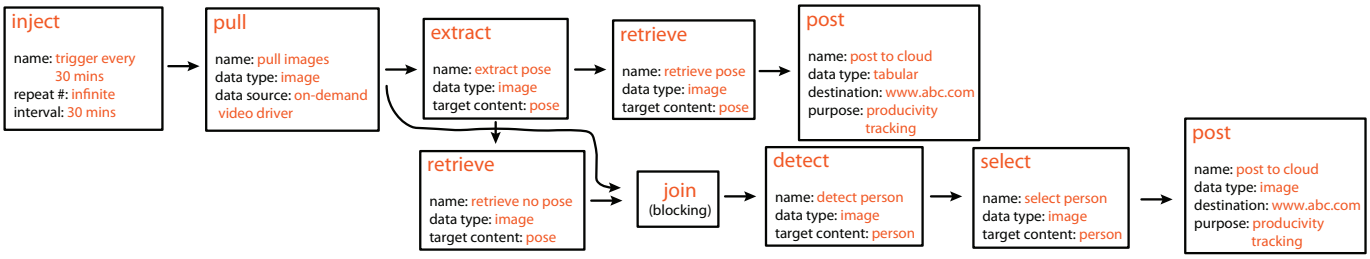

Figure 8: The productivity tracking app pulls an image from the camera every 30 minutes, analyzes the pose of the person, and uploads the pose to the server for further analysis. Pose information is usually unavailable when the person sits at a desk. The app then only sends the person pixels to the cloud with more sophisticated algorithms.

flow control similar to Unix Pipes. The runtime executes an operator only if its input is ready, and the filter operators only forward non-empty processed data to subsequent operators. For example, if the retrieve pose operator (Fig. 8) cannot find any extracted poses in its input, the flow stops propagating. This simple design makes it easy for Peekaboo APIs to support IFTTT-like trigger-action programs natively.

This example also shows how the output of a single operator can be forwarded to multiple operators, similar in spirit to Unix's *tee* command. The blocking and non-blocking *join* operators allow Peekaboo to accommodate distributed asynchronous sensors, supporting smart home apps with complex logic across distributed devices within the same home.

VI. Implementing the Hub Runtime

We implemented Peekaboo by leveraging Node-Red [35], a visual programming platform developed by IBM to wire together devices using a library of customized blocks. The notion of "blocks" and directional "connections" in the visual programming language are well suited to Peekaboo's chainable operators. We implemented Peekaboo operators as a set of Node-Red compatible building blocks, so developers can use a Node-Red web-based flow editor UI as the programming environment and leverage its built-in debugging utilities (e.g., displaying images, playing an audio, printing output data).

Operators: Each Peekaboo operator is written in JavaScript and executed by a Node-red runtime once deployed. One challenge in Peekaboo is the relative lack of support for machine learning algorithms in JavaScript. Our experiments with multiple JS implementations (e.g., opencv4nodejs) showed them to be slow and inaccurate. Instead, we design the Peekaboo runtime using a microservice architecture, where several ML inference algorithms are hosted in containerized services running on the Peekaboo hub, and only Peekaboo inference operators can communicate with these services locally through web sockets.

Drivers to obtain sensor data: Each Peekaboo hub program gets data from hub runtime drivers rather than querying hardware directly. Developers may implement customized runtime drivers in arbitrary code, similar to HomeOS [18]. These drivers retrieve raw sensor data (video, image, audio, tabular, or scalar) from edge devices, format it (e.g., b64string for images, bytes for video/audio, JSON for tabular data), and publish the data in a local socket. The hub program can either pull the latest data (e.g., retrieving a real-time photo)

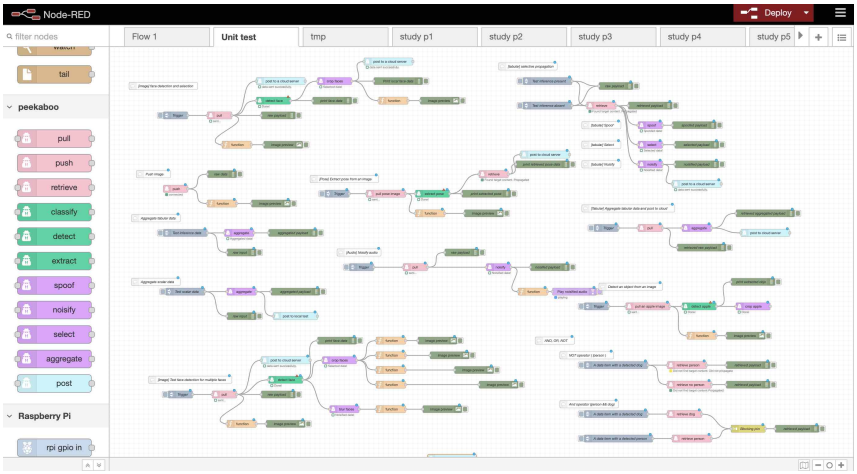

Figure 9: The editor interface and the unit test tab presented to the developers. Developers can inspect the unit tests to understand operators' behavior and the connection grammar.

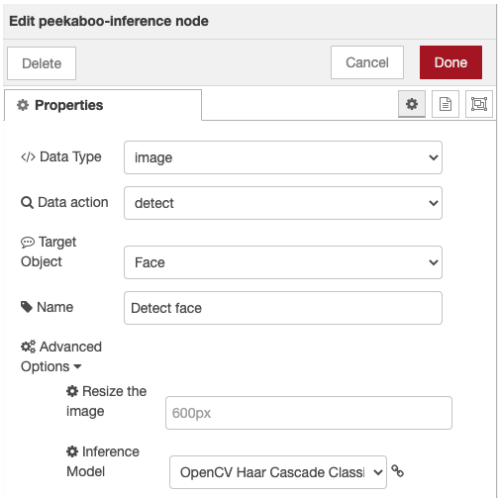

Figure 10: The configuration interface of an inference operator.

from the driver or register a push subscription on the driver (e.g., receiving a photo if there is a significant scene change). Note that these runtime drivers handle device heterogeneity and specifics of getting data from them, while the Peekaboo operators are device-agnostic data stream operators.

Runtime: We deployed the Peekaboo runtime (i.e., Node-Red, drivers, containerized inference services) on a Raspberry Pi 4B (4GB), along with a Google Coral TPU and a Zigbee adaptor, with a total cost of $100 USD. Most of our deep-learning-based inference tasks use pre-trained MobileNet [69] models and are outsourced to the TPU.

End-to-end applications: We developed drivers for four Peekaboo IoT devices, a customized smart camera and smart speaker using Google AIY Kits [29], a simulated RESTFul API that generates tabular smart TV logs, and an Aqara Zigbee humidity sensor, covering the five data types in Table VI (video, image, audio, tabular, and scalar).

We built five end-to-end applications using these devices: video based heart rate measurement [8], HelloVisitor (Fig. 1), incognito Speech Assistant (Fig. 7), Smart TV logs collector (Fig. 6), and humidity-based irrigation reminder. We built customized cloud services and web UIs for all five apps.

VII. Evaluation

This section presents detailed experimental evaluations of the feasibility and benefits of Peekaboo. We first created 68 different manifests for all the 200+ smart home use cases (Section III). Note, these manifests only work with bare-bone server implementations as we do not have proprietary backend implementations from manufacturers. We then analyzed types of pre-processing opportunities that can be enabled by these manifests (§VII-A). We also investigated the feasibility of balancing the tradeoff between privacy protection and utility for each type of pre-processing (§VII-B). We later built five end-to-end real-world Peekaboo apps, covering five data types and used these apps to evaluate our system performance (i.e., latency and throughput) (§VII-C). To test the usability of Peekaboo APIs, we provided the Peekaboo manifest authoring interface to 6 developers and ask them to author four manifests with pre-configured sensors and cloud services (§**??**). Finally, we demonstrate several hub privacy features that both developers and users essentially get for free, which work across apps and devices (§VII-D).

A. Evaluating Peekaboo's coverage of smart home scenarios

To validate the feasibility of Peekaboo, we authored manifests to cover the use cases described in §III.

Method. A smart home use case can be realized through different types of sensors. For example, a dedicated occupancy sensor or a camera can enable an occupancy-based application, although required data collection behaviors can differ. Two authors collaboratively implemented one manifest for each usage-sensor pair to explore different pre-processing opportunities. We found that many manifests could be reused for multiple scenarios, and so this process resulted in 68 manifests.

Results. Of the 68 manifests, we only identified three that always need raw data: a smart cooking device that uses a camera to analyze ingredients, a smart toilet that analyzes poop for diseases, and a microphone that performs spirometry [47]. For these kinds of apps, developers can directly connect a `provider` operator to a `network` operator to send out the raw data. While Peekaboo does not reduce data egress in these cases, it still provides transparency of data collection (e.g., what data has been sent to the service provider, and how often) and a binary on-off control.

For the other 65 manifests, Peekaboo APIs can enable at least one of the following types of pre-processing: content selection (e.g., cropping faces), conditional filtering (e.g., only sending data if a person appears in the view), and explicit noise injection (e.g., changing the pitch of an audio recording). Table I enumerates the breakdown of supported pre-processing across all scenarios, as well as the unsupported reasons.

Content selection is the most common pre-processing (64 of 68 scenarios). Examples include cropping faces from im-

Table I: We implemented 68 unique manifests for over 200 smart home use cases, analyzed the types of pre-processing in these manifests, and if the data needs for that use case could not be supported, we examined why.

Pre-processing	#Scenarios supported	Why Peekaboo could not support some of the scenarios
Content selection	64 / 68	e.g., hard to select content of interest, need to be used for online albums
Conditional filtering	57 / 68	e.g., high-stake tasks, proprietary implementations, insufficient computing resources
Explicit noise injection	51 / 68	e.g., high-stake tasks, injected noise may break the intended tasks
Always need raw data	3 / 68	the intersection of the three categories listed above

ages, extracting audio frequency spectrum, and aggregating numerical values. We could not apply content selection to 4 apps for two reasons. First, the algorithms to select the content of interest cannot run on a Peekaboo hub. For example, the algorithm to recognize food ingredient is proprietary and may require significant computational resources and frequent model updates. Second, developers need raw data to fulfill the data collection purpose. For example, automatic photography (e.g., Google Clips [3]) needs to collect and store the original photo.

Conditional filtering is also common (57/68). Since many smart home apps are event-driven, adding a local event filter can significantly reduce data egress. For example, a manifest that filters images based on the presence of faces can reduce the number of outgoing images from a basic motion-activated camera. However, one constraint of Peekaboo's architecture is that the open-source hub algorithms may not be as accurate as their cloud counterparts. As a result, we cannot insert filters for high-stake tasks, e.g., elderly fall detection. Another constraint is that some conditional filters might be proprietary with no current open source equivalent (e.g., water leakage detection). Finally, some conditional filters may require extensive computational power and storage, which cannot run on a local hub (e.g., wanted criminal search using smart doorbells).

While it is possible to inject explicit noise into the data pre-processing pipelines, there is one crucial privacy-utility trade-off: the injected noise may break the intended functionality and reduce the service quality. So we cannot explicitly inject noise to apps for high-stake tasks. Besides, many scenarios collect only coarse data (e.g., binary occupancy) or need raw data, where explicit noise injection is not applicable.

B. Privacy-utility trade-offs

While some data transformations are unlikely to affect service quality (e.g., many kinds of tabular data aggregation), others might have negative impacts. A main concern for content selection and conditional filtering is that the open-source inference models on the hub may be less effective than large proprietary machine learning models on the cloud. Furthermore, for explicit noise injection, that noise might break intended functionalities. While the actual trade-offs depend heavily on the applications, we used three example apps to demonstrate the **feasibility** of using simple algorithms to reduce privacy risks while having little impact on utility.

Method. We chose three pre-processing tasks to evaluate: a face-only video doorbell (HelloVisitor, Fig. 1, content-selection), a person-activated camera (Fig. 8, conditional filtering), and an incognito voice assistant (Fig. 7, noise injection). We chose these tasks because they represent different types of pre-processing, the availability of labeled ground truth data, and the availability of publicly accessible cloud-based baselines.

The face-only video doorbell can improve privacy by only sending images of faces, although Peekaboo's operator might miss some faces that a more capable cloud algorithm could detect. Similarly, the person-activated camera reduces unnecessary outgoing images, but it may miss images that potentially contain a person. For these two apps, we quantified the privacy benefits using the percentage of data egress reduction in bytes and the impact on utility using F1 scores. Finally, our incognito voice assistant protects speakers' identity by changing the tempo and pitch of the captured audio with small random variations using the *noisify* operator (<10%). We quantify the privacy benefit by measuring speaker recognition error and the utility impact using speech recognition error. We measured accuracy using the Levenshtein function [75] to compare the recognized speech text with the ground truth text.

We ran the experiments with multiple benchmark data sets. First, we used the ChokePoint dataset [76], a person identification dataset under real-world surveillance conditions, to test the face-only video doorbell. Next, we selected 457 "home office" videos from an in-home activity dataset [71] to evaluate the person-activated cameras. Finally, we used speech recordings from the CMU PDA database [59] to test the incognito voice assistant (6 unique speakers, 112 unique audio files). For each speaker, we used half of the audio files (7-15 files) for speaker enrollment and the other half for speaker recognition and speech recognition. We used state-of-the-art cloud-based solutions from Microsoft Cognitive Services (i.e., face detection, person detection, speaker recognition, speech recognition) as the baseline [7].

Results. Table II presents the privacy-utility tradeoffs across our three example apps. The face-only camera reduces data egress from 1 million full-resolution images (12 GB) to 64,000 face images (200 MB), while the F-1 score of the local model is only 1% lower than the Microsoft Azure API. The person-activated app replaces 2868 images (62.8%) with less privacy-sensitive pose key points and removes unnecessary background pixels from 770 images (16.8%), resulting in a data reduction of 95.7% (from 164 MB to 7 MB). Meanwhile, the F-1 score of the local model is only slightly lower than the Microsoft Azure API (93.2% v.s. 96.2%). The incognito voice assistant reduces speaker identification accuracy from 100% to 27.68% (lower is better since it provides more anonymity) while only

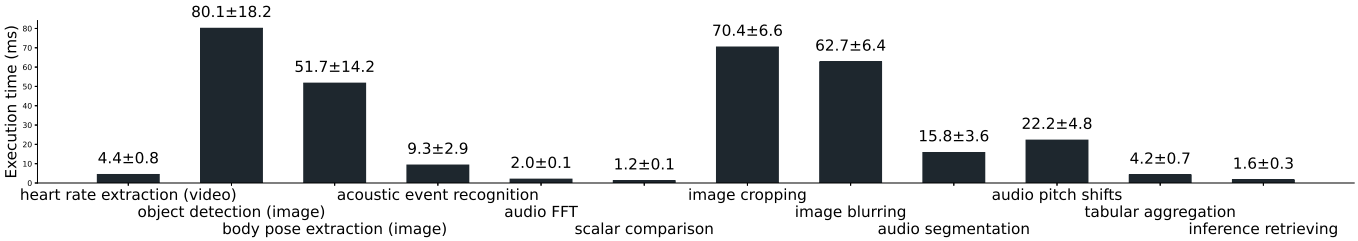

Figure 11: The execution times (in milliseconds) for transforming 1-second videos/audios (normalized), 800x600 images, and a 5-column table with 100 entries. Our low-cost setup can support 25 inference tasks and more than 100 filtering transformations per second, which we believe is sufficient to support many smart home scenarios.

Table II: Simple pre-processing algorithms provide significant improvement in the amount of potentially sensitive data sent, while maintaining utility.

	Privacy Metric	Utility Metric
	Outgoing data	F1 Score
Peekaboo face doorbell	**6.0**%	94.3%
Baseline doorbell	100%	**95.3**%
Peekaboo person camera	**4.3**%	93.2%
Baseline camera	100%	**96.2**%
	Speaker Recog Accuracy	Speech Word Error Rate
Peekaboo voice assistant	**27.7**%	11.88%
Baseline voice assistant	100%	**9.27**%

reducing speech recognition accuracy by 2.61%. Our results show that Peekaboo's pre-processing (e.g., small random pitch shifts, pre-filtering) can significantly reduce data egress with minor adverse impacts on the intended tasks.

C. System Performance

Peekaboo's architecture has two major constraints. The first is that the hub has more limited computing resources than cloud servers. Resource constraints on the hub may lead to pre-processing of data taking longer, or limit the ability to scale to many simultaneous pre-processing tasks in a smart home. Second, the pipe-and-filter architectural style introduces latency due to repetitive parsing and unparsing across filters. We conducted experiments to quantify the *computation load (i.e., throughput)* of data transformations and the *end-to-end latency* of different pipelines. We note, however, that our prototype is not highly optimized and is running on relatively low-end hardware, so this evaluation is intended to provide a rough lower bound on performance.

Method. We deployed the hub runtime to a low-cost setup and used the four sensors as the edge devices, both described in §VI. We also set up a cloud server on an AWS p2.xlarge instance, in an AWS region close to the authors institution.

We first characterized the computation load of common data transformations identified in §VII-A, including both inference and filtering. For inference, we profiled object detection (e.g., face, person, floor), audio event recognition (e.g., baby crying, water dripping), audio frequency spectrum extraction, body pose extraction, video heart rate extraction and scalar value comparison. For filtering, we profiled object-based image cropping, object-based image blurring, audio segmentation, audio pitch random shifts, tabular data aggregation, and inference results retrieving.

We used 3 videos from Intel's IoT Devkit [36] (~2 min) to test video transformations, 3 sound clips from Google AudioSet [26] (~10 seconds) to test audio transformations, 10 images from the ChokePoint dataset [77] (800x600) to test image transformations, and a synthetic dataset containing 100 entries to test tabular data transformations. We then measured the average computation time of 1000 repetitions.

Next, we compared the difference in latency when running pre-processing tasks on the hub vs running them on the cloud. With respect to the former, we measured the pre-processing latency on the hub and the transmission time of sending the processed data to the cloud. With respect to the latter, we measured the transmission time of sending the raw data to the cloud and the pre-processing latency on the cloud. We used the dataset mentioned above to test 3 end-to-end apps described in §V. We sent the requests at an interval of 0.5 seconds and measured the average latency of 1000 repetitions.

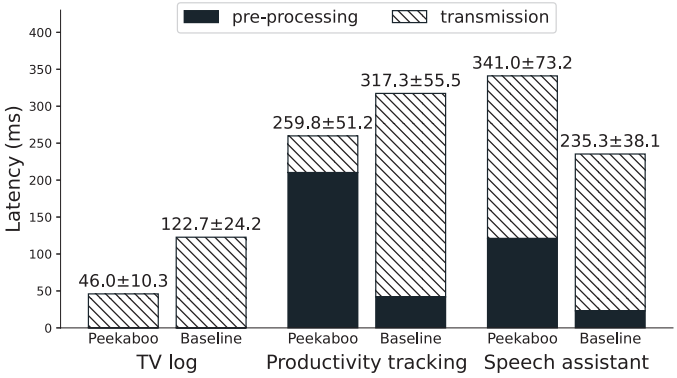

Figure 12: The latencies for the three apps are comparable to the conventional cloud-based approaches. The pre-processing times for TV logs are negligible. Although Peekaboo apps spend more time pre-processing data on the hub than on the cloud, they may spend less time on data transmission since pre-processing can reduce outgoing data size.

Results: Figure 11 presents the individual completion times on a Raspberry Pi 4B for common data transformations. Most