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 [72]. It is challenging for users to have any assurances of privacy *after* their sensitive data leaves the confines of their home [55]. 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 [61]. 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 [73], [77], but again there is no easy way to verify this behavior. Yet another alternative is to leverage new mechanisms for trusted cloud computing [26], [42], [47]. 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 [33]. We developed the operators as a set of Node-Red

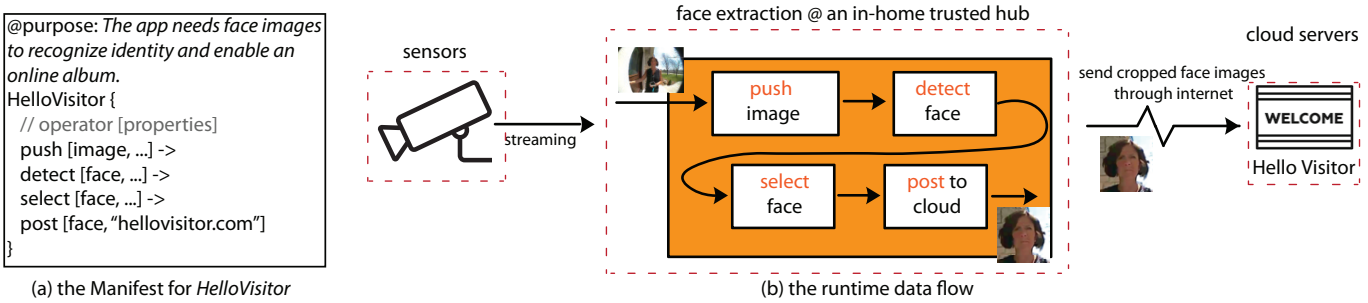

Figure 1: The architecture of the "HelloVisitor" app. A developer creates a manifest making use of four chainable operators (i.e., *push*, *detect*, *select*, and *post*) and adds a text string to specify the purpose. Once deployed, a camera streams the video to the hub, which pre-processes the video 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 [17]). 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 [20] 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. We performed a preliminary developer study to evaluate Pekaboo's usability, but due to space constraints we discuss it in our technical report [37].

We make the following contributions in this paper:

- 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 open-source prototype implementation[1] of a Peekaboo hub on a Raspberry PI platform with a TPU accelerator.
- 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 [22], [57] and Manufacturer Usage Description (MUD) whitelists [13] 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,

[1] https://github.com/CMUChimpsLab/Peekaboo

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 and only allow whitelisted outgoing data flows, similar to the MUD [13]. 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 [44], 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., [46]). Second, we surveyed the mobile privacy research literature (e.g., [21], [38], [49]) to understand common data use patterns by mobile app developers since many of them apply to IoT scenarios. Third, we conducted design fiction interviews [19] 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**

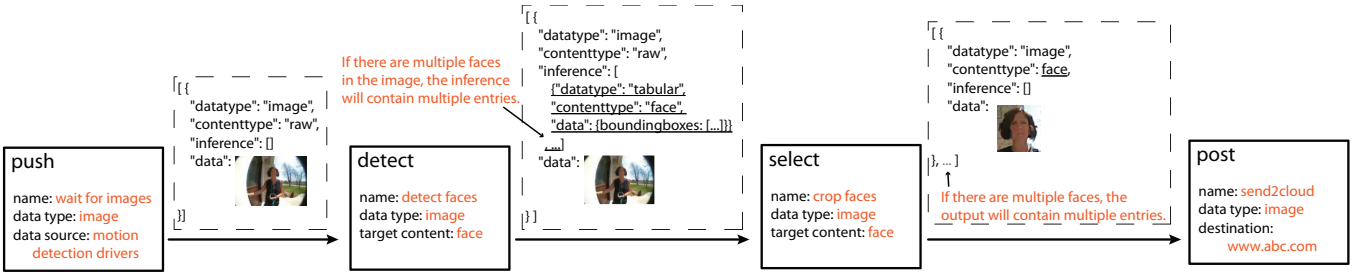

Figure 2: The hub program for "HelloVisitor" (Fig. 1). Operators come with a few configurable parameters and have clearly defined ways to interact with a uniform data model (dashed boxes). For example, the *detect* operator only modifies the "inference" field. The *select* operator modifies the "data" 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.

lightweight and non-proprietary, suggesting that they can be run on a low-end smart home hub. For example, pre-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 [24], 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.

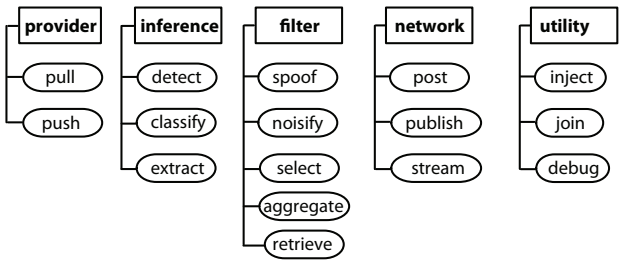

Figure 3: The taxonomy of Peekaboo operators. Each operator corresponds to a "verb" statement relative to the operator itself. For example, the *pull* operator pulls data from the hub runtime.

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], [18], [36], [43] 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

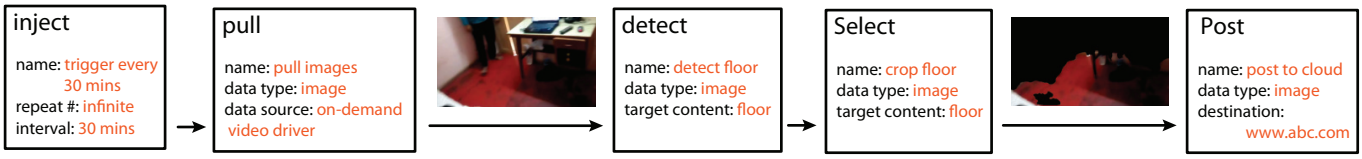

Figure 4: A water leak detection app pulls an image from the camera every 30 minutes, detects the floor area using image segmentation algorithms, and sends the image containing only the floor to the server. The app protects users' privacy by only sending the floor pixels to the cloud.

derived data.

The abstractions of operators are somewhat analogous to abstract classes in object-oriented programming, and developers 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], [50] (e.g., face, person, floor, table), 632 audio classes [25] (e.g., baby crying), and many other individual categories (e.g., body pose [51], 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 [67] 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 [15].

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.

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

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

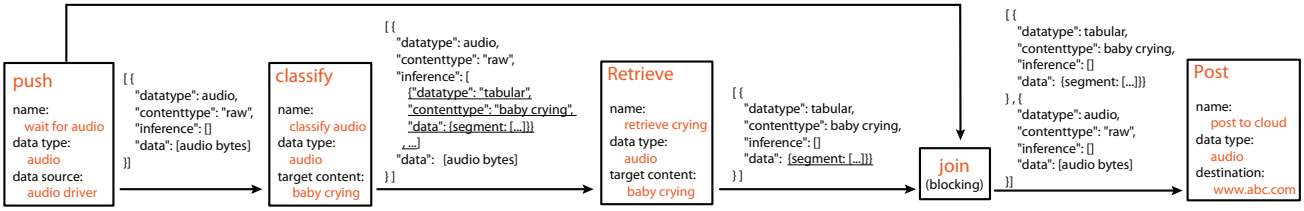

Figure 5: A baby monitoring app only sends audio containing a crying sound to the server. The *retrieve* operator replaces the "data" field with the "inference" field. The *join* operator merges the outputs from *wait for audio* and *retrieve crying*, passing its output onward when both input streams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

this is 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, 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. Fig. 7 presents a manifest of a smart voice assistant that we developed, which can offer a speech anonymization feature that protects users from exposing undesired voice fingerprints. In contrast to Google's "Guest Mode" [29], this manifest can **assure** users that their voice fingerprint identities are protected.

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 [66] 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.

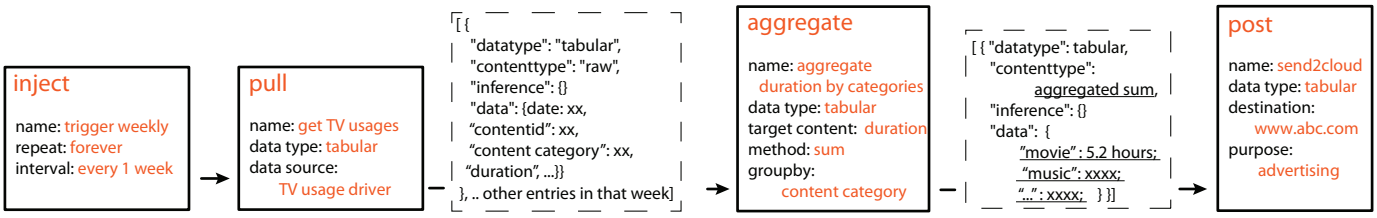

Figure 6: A SmartTV usage summary app queries users' watching patterns and uses the data to generate a weekly summary for video consumption across different channels and categories. The *aggregate* operator works similarly to SQL aggregation and replaces the value in the "data" field.

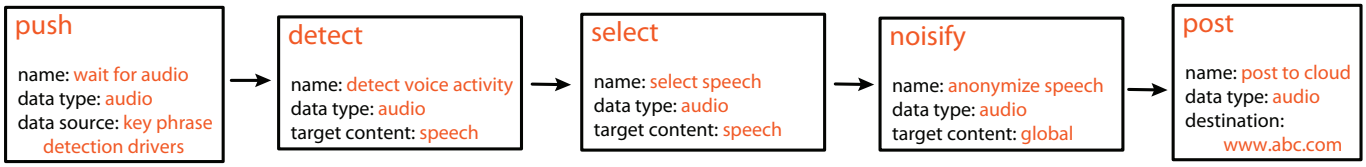

Figure 7: The incognito voice assistant only collects short audio segments containing speech and hides the speaker's identity by changing the pitch. This hub program can protect users' identity without breaking the speech recognition functionality.

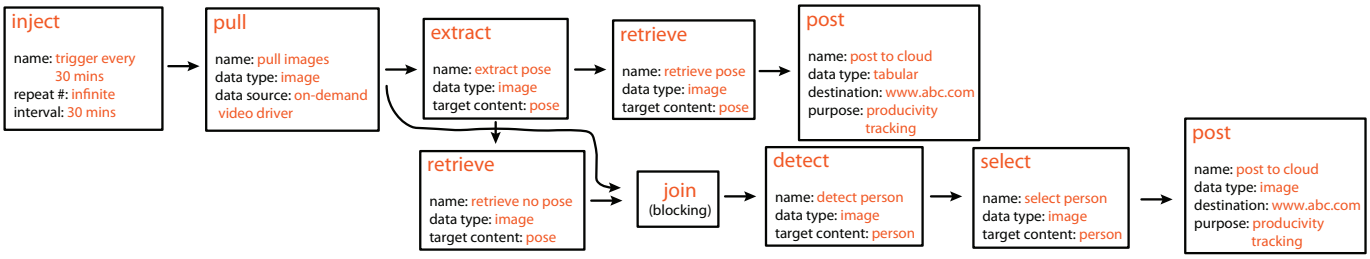

Figure 8: The productivity tracking app pulls an image from the camera every 30 minutes, analyzes the pose of the person, and uploads the pose to the server for further analysis. Pose information is usually unavailable when the person sits at a desk. The app then only sends the person pixels to the cloud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

This example shows use of Peekaboo's flexible conditional flow control. Peekaboo operators support an intrinsic condition 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 [33], 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). The source code is available at https://github.com/CMUChimpsLab/Peekaboo.

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 [17]. These drivers retrieve raw sensor data (video, image, audio, tabular,

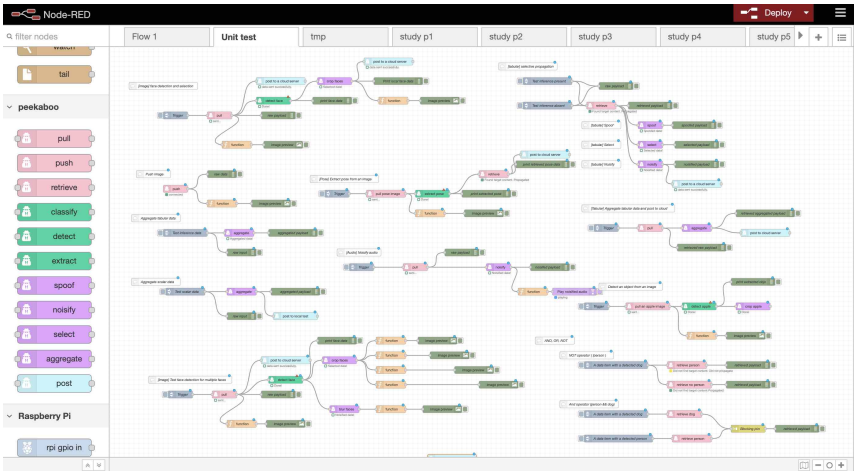

Figure 9: The editor interface and the unit test tab presented to the developers. Developers can inspect the unit tests to understand operators' behavior and the connection grammar.

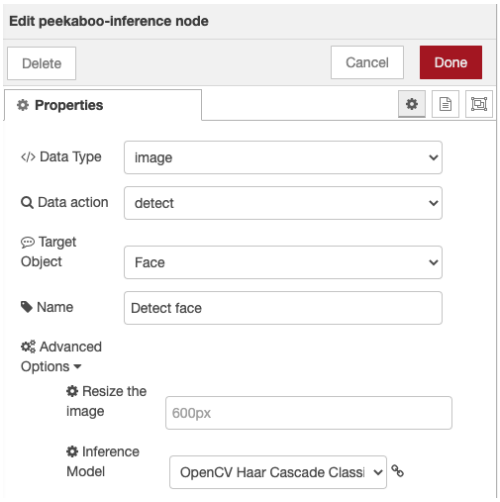

Figure 10: The configuration interface of an inference operator.

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) 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 $\approx$ $100 USD. Most of our deep-learning-based inference tasks use pre-trained MobileNet [68] 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 [28], 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). 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 [46]. 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.

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

Content selection is the most common pre-processing (64 of 68 scenarios). Examples include cropping faces from images, 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 [74] 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 [75], 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 [70] to evaluate the person-activated cameras. Finally, we used speech recordings from the CMU PDA database [58] 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

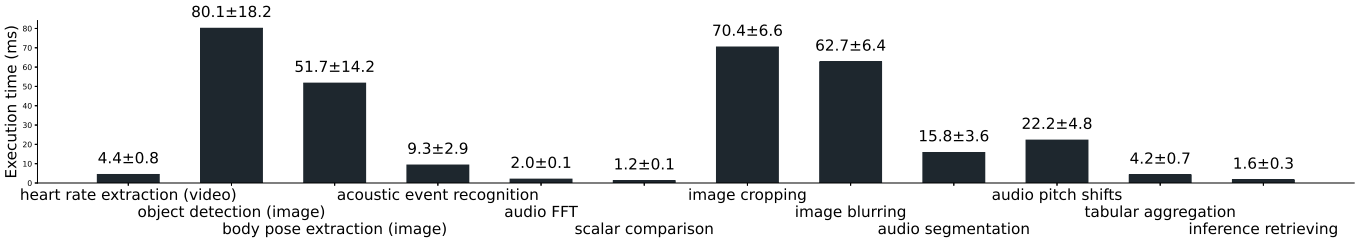

Figure 11: The execution times (in milliseconds) for transforming 1-second videos/audios (normalized), 800x600 images, and a 5-column table with 100 entries. Our low-cost setup can support 25 inference tasks and more than 100 filtering transformations per second, which we believe is sufficient to support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

reduces speaker identification accuracy from 100% to 27.68% (lower is better since it provides more anonymity) while only 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 [34] (~2 min) to test video transformations, 3 sound clips from Google AudioSet [25] (~10 seconds) to test audio transformations, 10 images from the ChokePoint dataset [76] (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. According to prior work [6], the frequency of network events from typical sensors (e.g., sleep monitors, nest cameras, switches) varies from a few per minute to a few per hour. We stress tested the requests at an interval of 0.5 seconds and measured the average latency of 1000 repetitions.

Results: Figure 11 presents the individual completion times on a Raspberry Pi 4B for common data transformations. Most filtering tasks take 5 ms to 80 ms to complete, while the inference tasks on multimedia data are generally more expensive. Although inference tasks (e.g., object detection) take around 80 ms to complete, they consume little CPU resources since the core computation are outsourced to the TPU device, and the operator runs asynchronously. On average, most of the ML models we use (MobileNet [68]) take around 40 ms on the TPU per inference. So we estimate that our low-cost setup can support 25 inference tasks and more than 100 filtering transformations per second, which we believe is sufficient to simultaneously support many smart home scenarios.

Figure 12 presents the average latency for different apps, showing that Peekaboo can achieve a latency comparable to conventional cloud-based approaches. The TV log app's hub

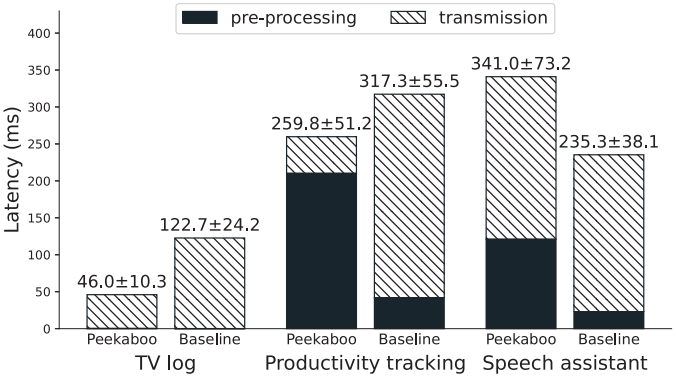

Figure 12: The latencies for the three apps are comparable to the conventional cloud-based approaches. The pre-processing times for TV logs are negligible. Although Peekaboo apps spend more time pre-processing data on the hub than on the cloud, they may spend less time on data transmission since pre-processing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

program spends negligible time on log aggregation but reduces the outgoing data size significantly, thus experiencing lower latency with Peekaboo. Similarly, Peekaboo's productivity tracking app spends 210.2 ms (std=49.6 ms) to extract the pose, reducing the network transmission time from 275.3 ms (std=50ms) to 40.1 ms (std=11.1ms). The Peekaboo speech assistant app takes 106ms more to pre-process and send a 10-second sound clip to the cloud than the conventional approach since the pre-processing in this case does not reduce the outgoing data size. In summary, the latency for pre-processing outgoing data on the hub is comparable to the conventional cloud-based approach. Some Peekaboo apps have reduced data transmission time due to the reduced size of outgoing data.

D. Hub Privacy Features

An advantage of Peekaboo's chained operators is that their structure and well-known semantics make it fairly easy to analyze and modify app behavior. We developed four built-in features to demonstrate the mechanisms.

We built a simple static analyzer to **generate natural language privacy descriptions** based on the manifest automatically, which can support users' decision-making in installing a new Peekaboo app. For any manifest, we can describe its data collection behavior using a four-element template: *[trigger], the app sends [content data] to [destination], if [condition].* We derived a set of heuristics to annotate the above properties for each edge in the pipeline based on the operator behavior. For example, the analyzer annotates [*content data*] as *face images* after processing by a cropping face operator. The analyzer annotates the [*condition*] if there is a join operator in the manifest and [*trigger*] based on the *inject* or *push* operator. The analyzer stops the annotation when it traverses the whole graph and uses the derived properties to explain the data collection behavior of each network operator. Table III enumerates explanations generated for apps in §V.

Table III: Auto-generated explanations for 3 case studies using a simple template. We highlight the properties of the template using underlines: trigger, content data, destination and conditions.

App #	Generated privacy explanations
# 1	For every week, the app sends duration data aggregated by content category to www.abc.com.
# 2	When the microphone detects a trigger phrase, the app sends anonymized speech audios to www.abc.com.
# 3	For every 30 minutes, the app sends extracted poses to www.abc.com. For every 30 minutes, the app sends cropped person images to www.abc.com if the app cannot recognize poses from the raw image.

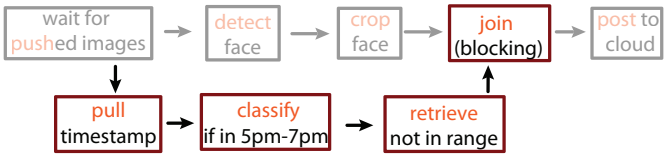

Figure 13: By inserting the subgraph of time checking (highlighted in red boxes), the modified HelloVisitor manifest will not send data outside between 5 pm and 7 pm.

We also implemented a **time-based scheduling** feature, which can pragmatically modify a Peekaboo manifest by inserting a subgraph of operators (highlighted in red boxes in Fig. 13), so the app cannot send data outside at certain times. Imagine that a family does not want their faces captured by the video doorbell when they arrive home, and most family members usually arrive between 5 pm and 7 pm. Based on users' time specifications, this feature can automatically insert a time-checking branch (i.e., pull->classify->retrieve) to check the time, and merge the two branches using a blocking join operator. In doing so, the data flow can only reach the network operator if the time condition qualifies.

Another feature we built is **pull rate-limiting**, which allows users to control the frequency at which a smart home app pulls data by modifying the operator properties. Figure 4 illustrates a water leak detection app, which pulls an image from an existing smart camera every 30 minutes. Using *pull rate limiting*, the runtime can modify the "interval" property of the *inject* operator from *30 minutes* to *120 minutes*, making the app only pull images every two hours.

Lastly, as a proof of concept, we combined the 3 above features to generate live "privacy nutrition labels" that summarizes an app's behaviors (Fig. 14). Apple now requires iOS app developers to fill in a form to create a self-reported "nutrition label" for privacy disclosures. However, it can be hard for developers to accurately fill out these forms. Furthermore, Apple and third parties cannot easily verify these declared behaviors. In contrast, Peekaboo's labels can be auto-generated, thus requiring less effort from developers, and always reflect

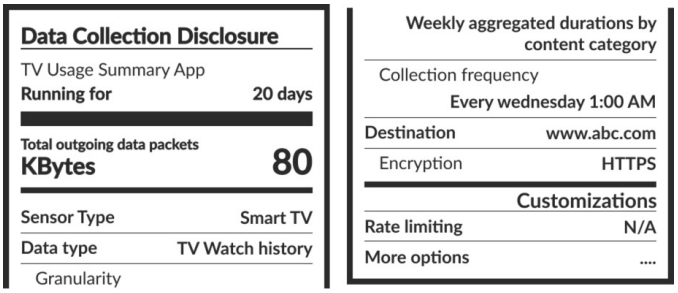

Figure 14: A "live" privacy nutrition label generated by the Peekaboo runtime automatically.

the actual data practice.

VIII. Related Work

Manifest & operators: There are many examples of using manifests to create a sandboxed environment to contain untrusted applications (e.g., Janus [27], Android permissions [5], MUDs [13]). In contrast to most existing manifests, which often confine applications' access to system resources (e.g., network, storage, sensors), Peekaboo's manifest confines access to data content (e.g., face images, speech audio) in a fine-grained manner. Due to the vast number of data granularities, the traditional manifest representation (i.e., raw access enumeration) is no longer feasible. Peekaboo addresses this challenge by allowing developers to assemble desired APIs by wiring together a fixed set of operators.

The design of Peekaboo operators is inspired by Unix pipes and PrivacyStreams [49]. PrivacyStreams splits single-pipeline Android data processing into a number of reusable SQL-like operators (e.g., sortby, filter, groupby), which can make data processing more transparent. The key innovations of Peekaboo are the integrated designs (a manifest, a fixed set of operators, and a trusted runtime with pre-loaded implementations) and the demonstration of using a small and fixed set of operators to support a large number of data pre-processing scenarios. These two ideas allow Peekaboo to offer many features that PrivacyStreams cannot offer, such as OS-level enforcement (i.e., developers can only collect data they claimed in the manifest), additional built-in privacy features, and explicit declarations of fine-grained data granularity (e.g., conditions).

Privacy-sensitive software architectures: A number of privacy-sensitive architectures for IoT have been proposed as alternatives to conventional cloud-based designs. One example is to process all sensor data at the "edge", thereby avoiding sending data to the cloud [77]. Another approach is to use trusted cloud computingand perform data minimization through technologies like DIY hosting [60], privacy-sensitive machine learning [12], [26], [47], [54], and federated learning techniques [42]. However, these approaches often come with tradeoffs such as sacrificing computational efficiency [47], [48], development flexibility [26], [48], or service quality [71].

In contrast, Peekaboo offers a hybrid approach that does some pre-processing locally on the hub while also allowing developers to use cloud services in a manner that they are accustomed to [31]. Peekaboo's hub is similar in spirit to cloudlets [69], but the key difference is that the computation running on the hub is structured and enforced using the operator-based manifest. Peekaboo is also inspired by past smart home hub/gateway/firewall projects [11], [14], [41], [53], [78]. Peekaboo has two major differences. First, Peekaboo enables data minimization at the data content level (e.g., only sending face images), while existing projects can only block individual outgoing network requests. Second, Peekaboo requires developers to explicitly declare the data collection behaviors, facilitating auditing and enforcement.

Privacy awareness and control: A complementary approach to privacy is better mechanisms for notice and choice [2], [17], [32], [52], [40], e.g. at runtime for mobile apps [16], [57], often extended to smart home contexts [17]. In the case of IoT privacy, merely allowing or denying sensor access is insufficient and this all-or-nothing access control either exposes sensitive data or breaks the app functionality. In response to this, recent efforts offer finer-grained control and transparency, particularly on the fidelity of the data [1], [36], [65], [73].

However, privacy support in these systems is often built on an individual basis with no common structure, imposing challenges for both app development and user privacy management. Third-party auditors also cannot easily verify if these features work as claimed [56], [61]. Peekaboo addresses these needs through a novel privacy architecture, which can enable transparent and enforceable data collection, and offer centrally manageable hub privacy features.

IX. Discussion

Architecture adoption: Many requirements for the Peekaboo hub are well aligned with the roles of recent commercial "hub" products, which may facilitate adoption. For example, many hubs (e.g., Philips Hue Smart Hub) serve as a central in-home gateway to connect devices, mediating access between the internal devices and the Internet. In addition, most hubs (e.g., Nest Hub Pro) have a moderate amount of computing power to pre-process out-going data and offer a centralized hub user interface. A few hubs (e.g., Samsung SmartThings Hub) even behave like an early app store.

Alternative implementations: Peekaboo's manifest is a new program representation that offers more flexibility than all-or-nothing access, while being more structural and verifiable than arbitrary code (e.g., Java). Future work can also implement the manifest in other flow-based programming frameworks (e.g., NoFlo, Pyperator) [45]. We chose Node-Red since it is popular in the home automation hobbyist community, provides many open-source device drivers, and has a mature user interface.

Design pattern adoption: The design of Peekaboo can be generalized as a reusable design pattern for cases where first- and third-parties are trying to access sensitive data, e.g., browser plugins, calendar APIs, and smartphones [64]. Our core ideas of a fixed set of operators, a text-based manifest where all outgoing data flows must be declared, and a trusted

computing platform with pre-loaded implementations can thus be useful in these cases. For example, Google Calendar only allows users to grant all-or-nothing access to third-party developers. However, most apps (e.g., Zoom) do not need full access. A future calendar API might offer a set of common operators instead and allow developers to program their access in Peekaboo-like manifests. This design pattern can make data flows transparent, enforce data transformations, and allow third-parties to build independent privacy features.

Role of users, developers, auditors in determining privacy-utility trade-offs: Peekaboo has the potential to disincentivize overcollection of data. We expect Peekaboo manifests to be public, making it possible for app stores and third-party auditors (e.g., Consumer Reports) to analyze manifests programmatically at scale. Users can also see if the required data granularity make sense, and flag items in a review if they do not, block certain outgoing data, or choose not to install an app. Altogether, this kind of transparency has the potential for nudging developers to collect less data.

Hosting proprietary algorithms: Some of the best implementations of inference mechanisms such as keyword spotting, keypoint tracking, and biometric authentication can be proprietary. Platform/hub builders may implement these algorithms inside their hub drivers in the future, or hardware developers can provide the functionality directly on the edge devices.

Extensibility: Peekaboo assumes a fixed set of operators and a stable data model to support its data pre-processing pipelines. Although the taxonomy in Fig. 3 may not be complete, we anticipate that the list and semantics of operators will converge quickly and remain stable for years. Further, platform builders can extend the operator options by expanding supported data transformations (e.g., removing all audible frequencies from the audio [35]) and adding new pre-trained models. Platform builders can also develop new drivers to support more devices.

We expect the runtime to be updated over time, analogous to getting a new version of Linux or Java. Also, similar to Android's manifest, a Peekaboo manifest will need to specify a minimum required runtime version. We do not expect the pre-loaded operators to grow into a large library of data transformation functions. Instead, we believe a few simple and common data transformations (e.g., tabular data aggregation, image cropping/blurring) can cover many common scenarios and go a long way towards improving privacy.

Benefits of Peekaboo: Peekaboo has three important advantages over building data minimization features individually. (1) **Transparency**. Individually built data minimization practices are black boxes to outsiders. Indeed, even if developers open-source their products or allow a third-party auditor to access their codebases, inspecting the actual data collection behavior is difficult. (2) **Ease of development**. Building data minimization algorithms and user interfaces for privacy requires significant effort. By authoring a Peekaboo manifest, Peekaboo developers can leverage many built-in features for free. (3) **Centralized privacy management**. If all developers build management interfaces individually, users would have to deal with potential inconsistencies between these user interfaces and their different semantics. In contrast, Peekaboo can offer centralized, fine-grained features across devices.

X. Future work & Limitations

End-to-end encryption: Peekaboo currently requires two separate encrypted connections (i.e., devices-to-hub and hub-to-servers) for its operation rather than end-to-end encryption from device to server directly. While SSL proxy mechanisms (e..g, [23], [39], [59], [63]) may provide a way to support end-to-end encrypted connections with the ability to verify what data is being sent, it is not clear whether they can support Peekaboo's operators that transform data. This is a current limitation, and we defer this to future work.

More hub features: Beyond the four hub privacy features introduced in §VII-D, further work may explore many other hub privacy features by analyzing and rewriting the manifest program. For example, the hub may aggregate the installed manifests, make privacy nutrition labels interactive, enable centralized privacy dashboards, and allow users to query what/when/how data flows out. The hub can also potentially allow users to apply global filters (e.g., blocking outgoing face images) across manifests, e.g. to make guests more comfortable with cameras around the house.

Manifests for other communication patterns: Peekaboo focuses on whitelisting edge-to-cloud data flows to improve privacy. A promising research direction is to generalize the manifest for other communication patterns, such as device-to-device, cloud-to-device, or even physical actuation. Ideally, we may have a set of Peekaboo-like manifests for each IoT device, whitelisting its interactions with the rest of the world in a common, structured, useful, and understandable way. Such an infrastructure can help users establish a correct mental model of device behaviors, allow the hub to offer built-in protection for the physical events (e.g., a bread toaster cannot run for more than an hour), and ease software development.

XI. Conclusion

This paper introduces Peekaboo, a new IoT app development framework to help developers build privacy-sensitive smart home apps. Peekaboo offers a hybrid architecture, where a local **user-controlled hub** pre-processes smart home data in a structured manner before relaying it to external cloud servers. In designing Peekaboo, we propose three key ideas: (1) factoring out repetitive data pre-processing tasks from the cloud side onto a user-controlled hub; (2) supporting the implementation of these tasks through a fixed set of open-source, reusable, and chainable **operators**; (3) describing the data pre-processing pipelines in a text-based **manifest** file, which only stores the operators' specifications, but not the operators' actual implementation.

Acknowledgement

This research was supported in part by the National Science Foundation under Grant No. CNS-1801472, CNS-1837607,

and CNS-2007786, Cisco, Intel, Infineon, and Air Force Research Laboratory under agreement number FA8750-15-2-0281. We thank the anonymous reviewers for their constructive feedback.

REFERENCES

[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. ACM, 2016.

[2] Yuvraj Agarwal and Malcolm Hall. Protectmyprivacy: detecting and mitigating privacy leaks on ios devices using crowdsourcing. In *Proceeding of the 11th annual international conference on Mobile systems, applications, and services*, pages 97–110, 2013.

[3] Aseem Agarwala. Google ai blog: Automatic photography with google clips. https://ai.googleblog.com/2018/05/automatic-photography-with-google-clips.html, 05 2018. (Accessed on 06/08/2020).

[4] Shahriyar Amini, Janne Lindqvist, Jason Hong, Jialiu Lin, Eran Toch, and Norman Sadeh. Caché: caching location-enhanced content to improve user privacy. In *Proceedings of the 9th international conference on Mobile systems, applications, and services*, pages 197–210, 2011.

[5] Android Developers. App manifest overview | android developers. https://developer.android.com/guide/topics/manifest/manifest-intro#perms, 02 2021. (Accessed on 02/04/2021).

[6] Noah Apthorpe, Dillon Reisman, and Nick Feamster. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. *arXiv preprint arXiv:1705.06805*, 2017.

[7] Microsoft Azure. Cognitive services—apis for ai developers | microsoft azure. https://azure.microsoft.com/en-us/services/cognitive-services/, 11 2020. (Accessed on 11/15/2020).

[8] Guha Balakrishnan, Fredo Durand, and John Guttag. Detecting pulse from head motions in video. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3430–3437, 2013.

[9] Joshua Bloch. How to design a good api and why it matters. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications*, pages 506–507, 2006.

[10] Liang-Chieh Chen, George Papandreou, Iasonas Kokkinos, Kevin Murphy, and Alan L Yuille. Deeplab: Semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs. *IEEE transactions on pattern analysis and machine intelligence*, 40(4):834–848, 2017.

[11] Haotian Chi, Qiang Zeng, Xiaojiang Du, and Lannan Luo. Pfirewall: Semantics-aware customizable data flow control for home automation systems. *arXiv preprint arXiv:1910.07987*, 2019.

[12] Jianfeng Chi, Emmanuel Owusu, Xuwang Yin, Tong Yu, William Chan, Patrick Tague, and Yuan Tian. Privacy partitioning: Protecting user data during the deep learning inference phase. *arXiv preprint arXiv:1812.02863*, 2018.

[13] Cisco. What is mud? - manufacturer usage description - document - cisco devnet. https://developer.cisco.com/docs/mud/#!what-is-mud, 02 2021. (Accessed on 02/05/2021).

[14] 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 *Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 122–133, 2017.

[15] Android Developer. Network security configuration | android developers. https://developer.android.com/training/articles/security-config. (Accessed on 09/07/2020).

[16] Apple Developer. Requesting permission - app architecture - ios - human interface guidelines - apple developer. https://developer.apple.com/design/human-interface-guidelines/ios/app-architecture/requesting-permission/, 04 2020. (Accessed on 04/16/2020).

[17] Colin Dixon, Ratul Mahajan, Sharad Agarwal, AJ Brush, Bongshin Lee, Stefan Saroiu, and Paramvir Bahl. An operating system for the home. In *Presented as part of the 9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, pages 337–352, 2012.

[18] Matt Duckham and Lars Kulik. Location privacy and location-aware computing. *Dynamic & mobile GIS: investigating change in space and time*, 3:35–51, 2006.

[19] Anthony Dunne and Fiona Raby. *Speculative everything: design, fiction, and social dreaming*. MIT press, 2013.

[20] Pardis Emami-Naeini, Yuvraj Agarwal, Lorrie Faith Cranor, and Hanan Hibshi. Ask the experts: What should be on an iot privacy and security label? In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 447–464, 2020.

[21] Kassem Fawaz and Kang G. Shin. Location privacy protection for smartphone users. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, page 239–250, New York, NY, USA, 2014. Association for Computing Machinery.

[22] Adrienne Porter Felt, Elizabeth Ha, Serge Egelman, Ariel Haney, Erika Chin, and David Wagner. Android permissions: User attention, comprehension, and behavior. In *Proceedings of the eighth symposium on usable privacy and security*, pages 1–14, 2012.

[23] Forcepoint. Tls inspection and how it works. https://help.stonesoft.com/onlinehelp/StoneGate/SMC/6.2.0/GUID-2B5CE217-C9FE-4D8F-915E-FEA1F8253BEC.html, 08 2021. (Accessed on 08/09/2021).

[24] David Garlan and Mary Shaw. An introduction to software architecture. In *Advances in software engineering and knowledge engineering*, pages 1–39. World Scientific, 1993.

[25] Jort F. Gemmeke, Daniel P. W. Ellis, Dylan Freedman, Aren Jansen, Wade Lawrence, R. Channing Moore, Manoj Plakal, and Marvin Ritter. Audio set: An ontology and human-labeled dataset for audio events. In *Proc. IEEE ICASSP 2017*, New Orleans, LA, 2017.

[26] 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, 2016.

[27] Ian Goldberg, David Wagner, Randi Thomas, Eric A Brewer, et al. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th conference on USENIX Security Symposium, Focusing on Applications of Cryptography*, volume 6, pages 1–1, 1996.

[28] Google. Aiy projects. https://aiyprojects.withgoogle.com/, 07 2021. (Accessed on 07/19/2021).

[29] Google. Guest mode: An easy privacy control for your home devices. https://blog.google/products/assistant/introducing-guest-mode/, 01 2021. (Accessed on 12/02/2021).

[30] Google AI. Google ai blog: Take your best selfie automatically, with photobooth on pixel 3. https://ai.googleblog.com/2019/04/take-your-best-selfie-automatically.html, 04 2019. (Accessed on 08/10/2020).

[31] Dominique Guinard, Iulia Ion, and Simon Mayer. In search of an internet of things service architecture: Rest or ws-*? a developers' perspective. In Alessandro Puiatti and Tao Gu, editors, *Mobile and Ubiquitous Systems: Computing, Networking, and Services*, pages 326–337, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[32] Danny Yuxing Huang, Noah Apthorpe, Frank Li, Gunes Acar, and Nick Feamster. Iot inspector: Crowdsourcing labeled network traffic from smart home devices at scale. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 4(2):1–21, 2020.

[33] IBM's Emerging Technology Services. Node-red. https://nodered.org/, 07 2020. (Accessed on 07/06/2020).

[34] Intel IoT Devkit. intel-iot-devkit/sample-videos: Sample videos for running inference. https://github.com/intel-iot-devkit/sample-videos, 08 2021. (Accessed on 08/18/2021).

[35] Yasha Iravantchi, Karan Ahuja, Mayank Goel, Chris Harrison, and Alanson Sample. *PrivacyMic: Utilizing Inaudible Frequencies for Privacy Preserving Daily Activity Recognition*. Association for Computing Machinery, New York, NY, USA, 2021.

[36] Suman Jana, Arvind Narayanan, and Vitaly Shmatikov. A scanner darkly: Protecting user privacy from perceptual applications. In *2013 IEEE Symposium on Security and Privacy*, pages 349–363, 2013.

[37] 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 (extended technical report), 2022.

[38] Haojian Jin, Minyi Liu, Kevan Dodhia, Yuanchun Li, Gaurav Srivastava, Matthew Fredrikson, Yuvraj Agarwal, and Jason I Hong. Why are they collecting my data? inferring the purposes of network traffic in mobile apps. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(4):1–27, 2018.

[39] Juniper. Ssl forward proxy. https://docs.paloaltonetworks.com/pan-os/8-1/pan-os-admin/decryption/decryption-concepts/ssl-forward-proxy, 08 2021. (Accessed on 08/19/2021).

[40] Rishabh Khandelwal, Thomas Linden, Hamza Harkous, and Kassem Fawaz. Prisec: A privacy settings enforcement controller. 2021.

[41] Ronny Ko and James Mickens. Deadbolt: Securing iot deployments. In *Proceedings of the Applied Networking Research Workshop*, pages 50–57, 2018.

[42] Jakub Konečnỳ, H Brendan McMahan, Daniel Ramage, and Peter Richtárik. Federated optimization: Distributed machine learning for on-device intelligence. *arXiv preprint arXiv:1610.02527*, 2016.

[43] John Krumm. Inference attacks on location tracks. In *International Conference on Pervasive Computing*, pages 127–143. Springer, 2007.

[44] Thomas Kubitza, Patrick Bader, Matthias Mogerle, and Albrecht Schmidt. Developing iot systems: It's all about the software. *Computer*, 53(4):58–62, 2020.

[45] Samuel Lampa. samuell/awesome-fbp: Awesome flow-based programming (fbp) resources. https://github.com/samuell/awesome-fbp, 06 2020. (Accessed on 12/02/2021).

[46] Eric C Larson, Mayank Goel, Gaetano Boriello, Sonya Heltshe, Margaret Rosenfeld, and Shwetak N Patel. Spirosmart: using a microphone to measure lung function on a mobile phone. In *Proceedings of the 2012 ACM conference on ubiquitous computing*, pages 280–289, 2012.

[47] Taegyeong Lee, Zhiqi Lin, Saumay Pushp, Caihua Li, Yunxin Liu, Youngki Lee, Fengyuan Xu, Chenren Xu, Lintao Zhang, and Junehwa Song. Occlumency: Privacy-preserving remote deep-learning inference using sgx. In *The 25th Annual International Conference on Mobile Computing and Networking*, pages 1–17, 2019.

[48] Tian Li, Anit Kumar Sahu, Ameet Talwalkar, and Virginia Smith. Federated learning: Challenges, methods, and future directions. *IEEE Signal Processing Magazine*, 37(3):50–60, 2020.

[49] Yuanchun Li, Fanglin Chen, Toby Jia-Jun Li, Yao Guo, Gang Huang, Matthew Fredrikson, Yuvraj Agarwal, and Jason I. Hong. Privacystreams: Enabling transparency in personal data processing for mobile apps. *Proc. ACM Interact. Mob. Wearable Ubiquitous Technol.*, 1(3), September 2017.

[50] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[51] TensorFlow Lite. Pose estimation | tensorflow lite. https://www.tensorflow.org/lite/models/pose_estimation/overview. (Accessed on 11/14/2020).

[52] Yabing Liu, Krishna P Gummadi, Balachander Krishnamurthy, and Alan Mislove. Analyzing facebook privacy settings: user expectations vs. reality. In *Proceedings of the 2011 ACM SIGCOMM conference on Internet measurement conference*, pages 61–70, 2011.

[53] Anna Maria Mandalari, Daniel J. Dubois, Roman Kolcun, Muhammad Talha Paracha, Hamed Haddadi, and David Choffnes. Blocking without breaking: Identification and mitigation of non-essential iot traffic. *Proceedings on Privacy Enhancing Technologies*, 2021(4):369–388, 2021.

[54] Yunlong Mao, Shanhe Yi, Qun Li, Jinghao Feng, Fengyuan Xu, and Sheng Zhong. A privacy-preserving deep learning approach for face recognition with edge computing. In *Proc. USENIX Workshop Hot Topics Edge Comput.(HotEdge)*, pages 1–6, 2018.

[55] GILES TURNER MATT DAY and NATALIA DROZDIAK. Thousands of amazon workers listen to alexa conversations | time. https://time.com/5568815/amazon-workers-listen-to-alexa/, 04 2019. (Accessed on 02/04/2021).

[56] Richard Mitev, Anna Pazii, Markus Miettinen, William Enck, and Ahmad-Reza Sadeghi. Leakypick: Iot audio spy detector. In *Annual Computer Security Applications Conference*, ACSAC '20, page 694–705, New York, NY, USA, 2020. Association for Computing Machinery.

[57] Mohammad Nauman, Sohail Khan, and Xinwen Zhang. Apex: extending android permission model and enforcement with user-defined runtime constraints. In *Proceedings of the 5th ACM symposium on information, computer and communications security*, pages 328–332, 2010.

[58] Yasunari Obuchi. Pda speech database. http://www.speech.cs.cmu.edu/databases/pda/README.html, 03 2003. (Accessed on 11/15/2020).

[59] Mark O'Neill, Scott Ruoti, Kent Seamons, and Daniel Zappala. Tls inspection: How often and who cares? *IEEE Internet Computing*, 21(3):22–29, 2017.

[60] Shoumik Palkar and Matei Zaharia. Diy hosting for online privacy. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 1–7, 2017.

[61] Valentina Palladino. Teardown shows nest cam is "always-on" even when you think it's off | ars technica. https://arstechnica.com/gadgets/2015/11/teardown-shows-nest-cam-is-always-on-even-when-you-think-its-off/, 11 2015. (Accessed on 01/24/2021).

[62] Seung-min Park, Daeyoun D Won, Brian J Lee, Diego Escobedo, Andre Esteva, Amin Aalipour, T Jessie Ge, Jung Ha Kim, Susie Suh, Elliot H Choi, et al. A mountable toilet system for personalized health monitoring via the analysis of excreta. *Nature biomedical engineering*, pages 1–12, 2020.

[63] PolarProxy. Polarproxy - a transparent tls proxy created primarily for incident responders and malware researchers. https://www.netresec.com/?page=PolarProxy, 08 2021. (Accessed on 08/19/2021).

[64] Amir Rahmati, Earlence Fernandes, Kevin Eykholt, Xinheng Chen, and Atul Prakash. Heimdall: A privacy-respecting implicit preference collection framework. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '17, page 453–463, New York, NY, USA, 2017. Association for Computing Machinery.

[65] Nisarg Raval, Animesh Srivastava, Ali Razeen, Kiron Lebeck, Ashwin Machanavajjhala, and Lanodn P Cox. What you mark is what apps see. In *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services*, pages 249–261, 2016.

[66] RescueTime. Rescuetime: Automatic time-tracking software. https://www.rescuetime.com/, 11 2020. (Accessed on 11/15/2020).

[67] Samsung. One simple home system. a world of possibilities. | smartthings. https://www.smartthings.com/, 08 2021. (Accessed on 08/19/2021).

[68] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4510–4520, 2018.

[69] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[70] Gunnar A Sigurdsson, Gül Varol, Xiaolong Wang, Ali Farhadi, Ivan Laptev, and Abhinav Gupta. Hollywood in homes: Crowdsourcing data collection for activity understanding. In *European Conference on Computer Vision*, pages 510–526. Springer, 2016.

[71] Lorenzo Valerio, Andrea Passarella, and Marco Conti. Accuracy vs. traffic trade-off of learning iot data patterns at the edge with hypothesis transfer learning. In *2016 IEEE 2nd International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*, pages 1–6. IEEE, 2016.

[72] Frank Wang, Ronny Ko, and James Mickens. Riverbed: Enforcing user-defined privacy constraints in distributed web services. In *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, pages 615–630, 2019.

[73] Junjue Wang, Brandon Amos, Anupam Das, Padmanabhan Pillai, Norman Sadeh, and Mahadev Satyanarayanan. A scalable and privacy-aware iot service for live video analytics. In *Proceedings of the 8th ACM on Multimedia Systems Conference*, pages 38–49. ACM, 2017.

[74] Wikipedia. Levenshtein distance. https://en.wikipedia.org/wiki/Levenshtein_distance, 02 2021. (Accessed on 02/05/2021).

[75] Yongkang Wong, Shaokang Chen, Sandra Mau, Conrad Sanderson, and Brian C Lovell. Patch-based probabilistic image quality assessment for face selection and improved video-based face recognition. In *CVPR 2011 WORKSHOPS*, pages 74–81. IEEE, 2011.

[76] Yongkang Wong, Shaokang Chen, Sandra Mau, Conrad Sanderson, and Brian C. Lovell. Patch-based probabilistic image quality assessment for face selection and improved video-based face recognition. In *IEEE Biometrics Workshop, Computer Vision and Pattern Recognition (CVPR) Workshops*, pages 81–88. IEEE, June 2011.

[77] Hyunwoo Yu, Jaemin Lim, Kiyeon Kim, and Suk-Bok Lee. Pinto: enabling video privacy for commodity iot cameras. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1089–1101, 2018.

[78] Han Zhang, Abhijith Anilkumar, Matt Fredrikson, and Yuvraj Agarwal. Capture: Centralized library management for heterogeneous iot devices. In *USENIX Security Symposium*, 2021.

APPENDIX

A. DESIGN FICTION INTERVIEW APPARATUS

We conducted design fiction interviews [19] with 7 participants (3 female, 4 male) across different age groups (min=21, max=60, avg=32) to broaden our set of smart home use cases. Design fiction is a speculative design method, where participants are asked to sketch diegetic prototypes in an imaginary story world. All participants had interacted with multiple smart home products before. For the interview, we presented each participant with a hypothetical scenario[2] and a floor plan of a home, asking each participant to play the role of different family members and brainstorm different smart home functionality they desired. We also asked participants to focus on the role's needs, rather than feasibility of sensing or hardware availability.

The hypothetical scenario: *Imagine a family of five members living in a single-family house. Jeff and Amy are husband and wife respectively. Dave and Rob are their eight-year and seven-month-old kids. Jeff's mother, Jen, helps look after the baby, and Jeff's brother Sam occasionally visits the house. The family plans to purchase a set of devices to enable different smart home applications.*

The role playing questions: *If you are [Jeff|Amy|...], what smart home usages do you want in the [basement|living room|...]?*

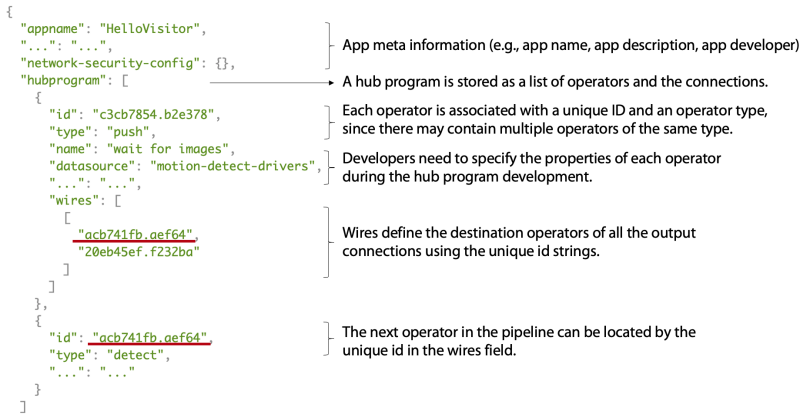

Figure 15: The manifest file of a hub program is stored in JSON format. The manifest file contains three types of information: app meta information, security configuration, and the hub program presentation.

Table IV: Supported data transformation across data types.

Data type	Operator	Supported transformations
Video	Extract	Extracting heart rates using [8]
	Retrieve	Keeping inference results but removing the video data
Image	Detect	Detecting the bounding box of 90 Common objects (Microsoft COCO [50]) and faces, Detecting the segmentation areas of 20 common object segmentation (PASCAL VOC2012 [10])
	Extract	Extracting the brightness, body poses [51]
	Noisify	Blurring an image
	Select	Cropping an image based on bounding boxes or a segmentation areas
	Retrieve	Keeping inference results but removing the image data
Audio	Detect	Detecting voice activity windows (i.e., starting time and ending time)
	Classify	Recognizing 632 audio events (AudioSet [25])
	Extract	Extracting frequency spectrum, speech text
	Noisify	Injecting a configurable random variation to the pitch and tempo.
	Retrieve	Keeping inference results but removing the audio data
Tabular	Select	Selecting a column with an optional where clause
	Aggregate	Aggregating (i.e., sum, count, average) the tabular entries by one field and projecting the output to a designated field.
Scalar	Classify	Comparing a value with a threshold
	Aggregate	Computing the sum, count, average of matched scalar items.
	Retrieve	Keeping inference results but removing the original scalar data

[2]Adapted from [17]

Table V: A summary of collected smart home use cases drawn from the research literature and design fiction interviews we conducted (§III). We obtained the initial list of use cases from the design fiction interview (Appendix §A), which asked participants to brainstorm use cases in each room using a single-family house floor plan. We then augmented this list by enumerating different sensors supporting these applications and potential data collections that developers may need. This list is not exhaustive, but our goal is to have a large sample covering many common smart home use cases.

ID	Description (Location)	Relevant sensors
#1	Water leak detection on the floor (Basement, Kitchen, Garden, Bathroom)	Camera, humidity, microphone
#2	Home inventory tracking (Storage room, Closet)	Camera, RFID
#3	Toxic Gas Alarm (Basement, Kitchen)	Specialized sensors
#4	Temperature tracking across rooms (The whole house)	Themometer
#5	Garage usage detection (Garage, garden)	Camera, occupancy, microphone
#6	Street parking spots detection (Driveway)	Camera, occupancy, radar
#7	Home arrival recording/prediction (Entrance doors, Garage)	Camera, RFID
#8	Garden irrigation tracking (Garden, driveway)	Camera, humidity
#9	Soil health tracking (Garden, driveway)	Specialized sensors
#10	Bath room activity (e.g., toileting for elderly) tracking (Bathrooms)	Humidity, occupancy, camera, pressure
#11	Smart Speaker & Voice control TV (Living rooms, Bedrooms)	Microphone, proximity
#12	Room occupancy statistics (The whole house)	Camera, occupancy, microphone
#13	Automatic temperature control based on the occupancy/identity (The whole house)	Camera, occupancy, microphone
#14	Office productivity tracking (Office room)	Camera
#15	Personalizing welcome message for visitors (Entrance doors, garage)	Camera
#16	Package delivery detection (Entrance doors)	Camera
#17	Fall detection for the elderly (The whole house)	Camera, microphone
#18	Automatic photography [3] (Living room, Kitchen, Garden)	Camera
#19	Automatic lighting based on occupancy and light intensity (The whole house)	Occupancy, light sensors, camera
#20	Wanted criminal search on the street (Entrance doors)	Camera
#21	Detecting strangers when no one is at home (The whole house)	Camera
#22	Ice detection (Entrance stairs, driveway)	Camera, specialized sensors
#23	Sunshine tracking for plants (Terrace, garden)	Camera, light sensors
#24	Detecting messiness of the home and calling a cleaning service (Living room, kitchen, closet)	Camera
#25	Smart cooking (Kitchen)	Smart appliances
#26	Freezer ice cleaning reminder (Kitchen)	Smart appliances
#27	Appliances electricity consumption statistics (The whole house)	Smart appliances and plugs
#28	Sleep tracking and sleep quality measurement (Bedrooms)	Camera, microphones, pressure sensors
#29	Smart toilet recognizes butt and analyzes poop for diseases [62] (Toilets)	Camera, pressure sensors
#30	Detecting baby crying (The whole house)	Camera, microphone
#31	Laundry service reminder (Closet, the whole house)	Camera, RFID
#32	Ubiquitous bio-metric measurement (e.g., height, heart rate) (the whole house)	Camera, heart rate sensor, Wi-Fi
#33	Pet barking detection (The whole house)	Camera, microphone
#34	Smart stylists that locates the clothes (Closet)	Camera, RFID
#35	Water activity detection (The whole house)	Specialized sensors
#36	Measuring lung function using a microphone	Microphone
#37	Acoustic ranging for games and cross-device interaction	Microphone

Table VI: What our team felt were reasonable outgoing data granularities for different smart home scenarios. We cluster these scenarios by their input and the output. If the output is Original, it implies no pre-processing functions can be done on the hub for these scenarios. The "(#scenario)" is the number of scenarios for each corresponding category, and the numbers in the "scenarios" column refer to the scenario id in Table V. Since a use case may have multiple reasonable app designs, the use case may be counted more than once.

Data type (# scenarios)	Pre-processed output (# scenarios)		Example usage	Scenarios
Image/Video (27)	**Original**			
	- raw video	1	Enabling online security camera album	#21
	- raw image	3	Enabling automatic photography (e.g. Google Clips [30])	#18
			Recognizing special activities (e.g., garden irrigation)	#8, #29
	Partial original			
	- only person/face	10	Recognizing human relevant activities (e.g., fall, sleep)	#7, #12, #13, #14, #17, #20, #21, #28, #30, #32
	- excluding person	7	Determining home messiness	#1, #2, #5, #19, #23, #31, #34
	- particular objects	3	Detecting certain objects (e.g., package)	#16, #22, #33
	Derived			
	- identity	2	Personalizing the welcome message	#7, #15
	- pose	2	Quantifying work activities	#14, #17
	- light intensity	2	Determining the lightness of the environment	#19, #23,
Audio (9)	**Original**			
	- raw audio	2	Supporting signal processing at per-frame level	#36, #37
		1	Supporting phone call, audio diary	#11
	Partial original			
	- voice audio	1	Supporting speech interaction	#11
	- activity sound	3	Detecting garage events	#5, #28, #30
	Derived			
	- speech text	1	Recognizing command intents	#11
	- FFT/MFCC	2	Detecting occupancy	#12, #13
	- audio events	3	Sending notification to owners when the dog barks	#12, #13, #33
Tabular (e.g., Channel state information) (5)	**Original**			
	- complete CSI	1	Enabling fine-grained sensing tasks using the Wi-Fi signals	#32, #34
	Partial original			
	- only UUID	3	Tracking the food/cloth inventory	#2, #7, #30
	Derived			
	- distance/positions	1	Helping users find the item in home	#34
Scalar (e.g. temperature, location, moisture) (13/37)	**Original**			
	- humidity	2	Detecting water leakages using precise humidity records	#1 , #8
	- water pressure	1	Detecting in-home activity using precise humidity records	#35
	- misc	8	Most scalar values are relatively safe to share	#3, #4, #5, #6, #22, #26, #27, #29
	Partial original			
	- coarse humidity	1	Detecting showering events in the bathroom	#10
	Derived			
	- out-of-town status	1	Detecting if are users out-of-town to adjust the AC temperature	#13