My House, My Rules: A Private-by-Design Smart Home Platform

Igor Zavalyshyn
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
UCLouvain, ICTEAM
igor.zavalyshyn@uclouvain.be

Nuno Santos
INESC-ID / Instituto Superior Técnico, Universidade de Lisboa
nuno.m.santos@tecnico.ulisboa.pt

Ramin Sadre
UCLouvain, ICTEAM
ramin.sadre@uclouvain.be

Axel Legay
UCLouvain, ICTEAM
axel.legay@uclouvain.be

ABSTRACT

Smart home technology has gained widespread adoption. However, several instances of massive corporate surveillance and episodes of sensor data breaches have raised many privacy concerns amongst potential consumers. This paper presents PatrIoT, a private-by-design IoT platform for smart home environments. PatrIoT revisits the typical architecture of existing IoT platforms, and provides an alternative design where the home owner retains full ownership and control of smart device generated data. It leverages Intel SGX to prevent unauthorized access to the data by untrusted IoT cloud providers, and offers homeowners an intuitive security abstraction named *flowwall* which allows them to specify easy-to-use policies for controlling sensitive sensor data flows within their smart homes. We have built and evaluated a PatrIoT prototype. Most of the participants in a field study considered PatrIoT to be easy to use, and the supported policies to be useful in protecting their privacy.

ACM Reference Format:
Igor Zavalyshyn, Nuno Santos, Ramin Sadre, and Axel Legay. 2020. My House, My Rules: A Private-by-Design Smart Home Platform. In *MobiQuitous 2020 - 17th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous '20), December 7–9, 2020, Darmstadt, Germany*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3448891.3450333

1 INTRODUCTION

Despite the growth and popularity of smart home devices and systems, this technology remains overshadowed by a cloud of security and privacy concerns. Today, by relying on IoT platforms like Samsung SmartThings, Amazon Alexa, or Apple HomeKit, homeowners can seamlessly control smart devices, such as smart locks, virtual assistants, or baby cams, and run third-party applications (*apps*). However, falling under the control of antagonist actors, these systems can be turned into authentic spying platforms. In fact, once installed various third-party apps can collect highly sensitive data, e.g., video, audio, or sensor events of the surrounding environment, which can be abused in harmful ways [21].

MobiQuitous '20, December 7–9, 2020, Darmstadt, Germany

ACM ISBN 978-1-4503-8840-5/20/11…$15.00
https://doi.org/10.1145/3448891.3450333

Various mitigation techniques have been proposed for verifying apps' security and safety properties [3] and improving access control mechanisms [8]. However, common across all these efforts is the assumption that IoT platform providers are to be considered fully trusted. Currently, the platform providers can fully control the IoT cloud backend and collect, store, and / or share users' sensor data. Unfortunately, such privileges have already caused serious data misuse incidents that fall under the direct responsibility of IoT platform providers, involving targeted advertisement [6], surveillance and forensic investigations [9], insider-related eavesdropping or massive data leakage [14].

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

To achieve the first goal, PatrIoT relies on a hardened cloud backend service that runs inside a trusted execution environment (TEE) supported by Intel SGX technology. SGX secure enclaves offer memory-isolated environments that provide confidentiality and integrity protection against untrusted privileged system processes. By processing sensor data inside SGX secure enclaves, PatrIoT can effectively restrict the data access privileges of the cloud provider. To reach the second goal, in analogy to a "firewall", PatrIoT introduces the notion of *flowwall* which controls *how* third-party apps use the sensor data they request access to. Flowwall consists of an information flow control (IFC) monitor that controls the global device policies specified by the users. In contrast to existing permission-based smart home systems [7], that are either too coarse-grained or require certain expertise from the users to evaluate the potential risks on a *per-app* basis, PatrIoT's flowwall allows users to think in terms of *devices* they have and how those devices' data may or may not be used by *any* app they install.

PatrIoT makes two central contributions involving its policy specification and enforcement mechanisms. As for policy specification, many existing privacy-oriented solutions have failed to provide an adequate user interface, overwhelming the users with low-level details and causing the decision fatigue [2]. To address

this usability challenge, PatrIoT's UI was designed to make the process of privacy policy specification intuitive and easy to follow for a regular user. To define a policy, users operate with familiar device names, meaningful data types, e.g. audio or video, and destinations where these data types can or cannot flow to. The policy rules are defined once and applied to all the apps installed in the future.

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

We built a prototype of PatrIoT by leveraging SCONE [1], which allows us to deploy the PatrIoT backend securely in a Docker container running inside an SGX enclave. PatrIoT provides a JavaScript API for app developers and runs on top of Node.js. We use Prolog predicates to generate and check the apps' data flow models.

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

2 A PRIVATE-BY-DESIGN SMART PLATFORM

Our idea of a private-by-design IoT smart platform is one where the home user is the only party that retains exclusive ownership rights over the sensor data generated by the smart devices deployed at home: IoT platform provider and smart apps can acquire only the access rights that a user will explicitly decide to grant to them. Next, we present the system and security models, and then our design goals and a threat model.

2.1 System Model

The proposed system model is presented in Figure 1. Its central component is the TEE-protected Smart App Runtime (TSAR). It consists of a software stack which runs on a cloud infrastructure and provides the basic backend services for managing smart devices and hosting smart apps. With a management mobile app, a homeowner (user) can securely interact with the TSAR service in order to manage his smart devices, and install and configure smart apps downloaded from an app store. Once installed, smart apps run inside sandboxes, and can access sensor data based on permissions and a global user-defined security policy.

In contrast to existing IoT platforms, the TSAR service is hardened in such a way that an IoT cloud administrator does not have any access privileges over the users' sensor data. This is achieved

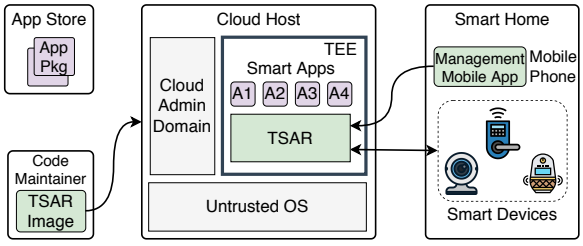


Figure 1: System model of our private-by-design IoT platform.

with two techniques: (i) by restricting the TSAR service external interfaces so that only the management app or smart devices are able to connect to it through TLS channels, and (ii) by running it inside a TEE so as to prevent privileged OS processes from accessing the TSAR memory where sensor data resides. A TEE is provided by dedicated hardware such as Intel SGX. Thus, a home user will only need to trust in the implementation of the TSAR software, and acquire a proof of its secure deployment in a cloud so as to obtain exclusive access rights in managing his smart home.

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

PatrIoT offers a clean slate IoT platform design which is not necessarily compatible with existing devices, apps and platforms. While disruptive in its nature, we argue there are strong economic incentives in favor of PatrIoT's adoption. First, there is a huge demand for privacy-preserving solutions among consumers and think tanks [5]. Second, there is an increasing pressure from lawmakers for stricter data protection measures (e.g., GDPR). Third, the smart home market is still very fragmented and lacking standards; as such, PatrIoT can make an important contribution to the consolidation of privacy-enhancing techniques for smart homes.

2.2 Security Model

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

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

- **S2S: Smart Device → App → Smart Device:** These are internal flows within home, e.g., app reads the status of a presence sensor to detect someone's arrival, and turns on a smart light.

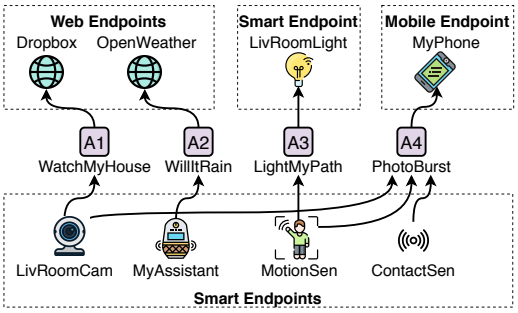


Figure 2: Smart home scenario with four installed apps: security surveillance app WatchMyHouse, voice-activated WillItRain app for weather forecast check, LightMyPath app for motion-triggered lights control, and a PhotoBurst app which notifies the user with the camera photo when motion or contact event is registered.

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

To characterize such flows and to easily specify policies for blocking or allowing them, the flowwall is based on several concepts that Figure 2 helps to introduce. This figure shows an example of a home scenario where four smart apps are installed (A_1-A_4). To perform their functions, smart apps may request access to certain objects named *endpoints*. Endpoints represent system resources that can act as producers (i.e. data *sources*) or as consumers (i.e. data *sinks*) of sensor data. Each endpoint fits into one of three classes:

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

Data flows can then be represented by the arrows shown in Figure 2. A *flow* is defined by the transfer of a specific sensor data type between source and sink endpoints. For instance smart app A_1 reads frames from the user's camera located in the living room (LivRoomCam) and uploads them to a user's Dropbox account, generating a Image data flow between these two endpoints. Collectively, apps A_1-A_4 illustrate all three data flow patterns, i.e., S2S, S2M, and S2W. The flowwall will i) keep track of all possible apps' data flows, and ii) allow or block specific flows according to the rules specified in a *security policy*. For instance, by blocking all flows from the living room's camera (LivRoomCam), apps A_1 and A_4 would necessarily be blocked. Next, we clarify our requirements to build an IoT system based on a flowwall security monitor.

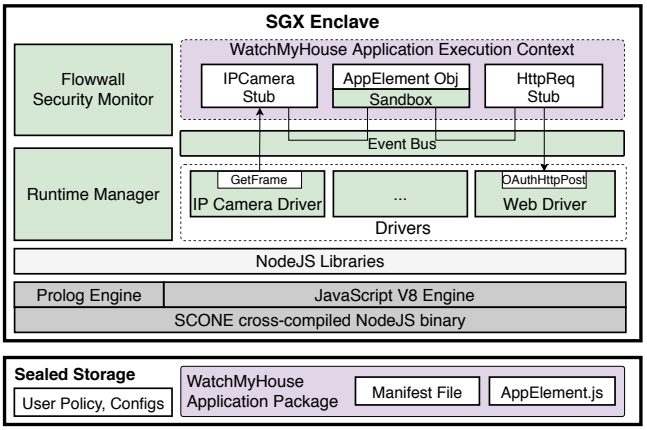


Figure 3: Components of PatrIoT TSAR.

2.3 Design Requirements and Threat Model

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

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

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

3 DESIGN

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

3.1 TEE-protected Smart App Runtime

The core of our system is the PatrIoT TSAR service (see Figure 3). It was built by leveraging SCONE [1], which offers a secure Docker container execution environment on top of SGX-enabled CPUs and protects the container processes from external attackers. It implements a Library OS with a small trusted computing base. The TSAR

service is provided by a containerized process that runs a Node.js binary cross-compiled against the SCONE libraries, and with a native Prolog engine add-on that is used for checking flowwall policies. Node.js then runs the PatrIoT TSAR-specific components, which are written in JavaScript.

The *runtime manager* is the heart of the TSAR service. It manages smart devices, apps, and user configurations for a given home environment. In particular, it controls the life cycle of apps and maintains their execution contexts. Apps interact with the environment through an API, which leverages an internal *event bus* for interfacing with *drivers*. There are multiple drivers responsible for interacting with smart, mobile and web endpoints, and for offering other services (e.g., timers). The flowwall *security monitor* tracks all apps' data flows and enforces a user-defined security policy. The persistent state consists of TSAR-specific files (e.g., security policy and configuration files), and app packages installed by the user. It is protected by sealed storage encryption techniques.

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

3.2 PatrIoT API

PatrIoT app development API was designed not only to offer easy-to-use programming abstractions, but also to enable the implementation of a sound, meaningful, and efficient taint tracking mechanism for flowwall policy checking purposes.

Element-based programming: In order to make all internal data flows within a given app explicit and easy to analyze, PatrIoT adopts a dataflow-based programming model where an app is written not as a monolithic program (e.g., a single JavaScript function), but as an *element graph* whose nodes consist of *elements*, and edges correspond to *connections* between elements. Elements represent functional units offered by the PatrIoT API (e.g., to interact with TSAR drivers) or implemented by the app developer. The connections define the only paths for data to flow between the app elements.

Elements can receive or send data through an interface consisting of one or multiple input/output *ports* with attached data types. Two elements can be connected using asynchronous unidirectional or synchronous bidirectional links. Some elements are natively provided by the PatrIoT API. They have well-defined specifications, both in terms of interface and expected behavior, and may require

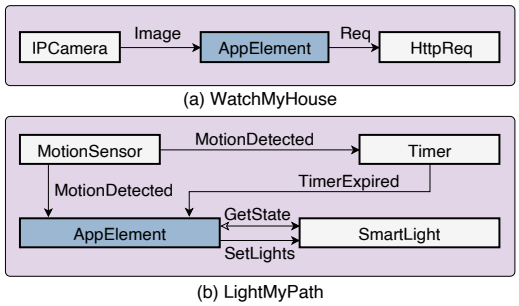


Figure 4: Element graphs of two PatrIoT apps

specific permissions (e.g., to access smart devices). For this reason these elements are named *trusted*. App developers can write customized elements in JavaScript to implement application-specific logic. Because their code and internal behavior are not trusted, these elements are named *untrusted* and run inside sandboxes.

PatrIoT API by example: To illustrate these concepts, imagine we want to develop a simple surveillance app named WatchMyHouse. Its goal is to take a periodic picture from an IP camera selected by the user and upload that picture to the user's Dropbox account. We can implement this app by creating an element graph consisting of three elements (boxes) connected as shown in Figure 4 (a).

Trusted elements are colored in white, and the untrusted one is in dark shade. Arrows represent elements' pairwise connections between input-output ports. PatrIoT API provides a rich library of built-in trusted elements for writing a variety of different apps. The trusted `IPCamera` element links to the IP Camera driver. It takes a picture from a camera endpoint and forwards the picture to a specific output port. The exact camera endpoint is selected by the user at app installation time. Another trusted `HttpReq` element links to the Web driver. Whenever it receives a data blob in one of its input ports, it sends a request to the Web driver which in turn issues an HTTPS post request to an OAuth2 authenticated web endpoint. The target endpoint is explicitly indicated by the developer, and it is validated by the web driver. The untrusted `AppElement` element connects two other elements. The app developer needs to write the JavaScript code for this element to read the picture from `IPCamera`, and prepare and push a request to `HttpReq`. This app package will then consist of a *manifest file* written in JSON that describes the app's element graph, and the *JavaScript code* of `AppElement`.

At runtime (see Figure 3), the TSAR service creates an application execution context which consists of (i) element stubs that point to the drivers that implement the trusted elements used by the app, and (ii) stateless sandboxed instances of untrusted element code (i.e., `AppElement` code). These objects communicate through the event bus according to the paths that have been declared in the app's manifest. The flowwall security monitor oversees these flows, and decides whether or not the app is allowed to execute depending on the rules specified in the security policy.

3.3 Flowwall Security Policies

A flowwall security policy consists of a sequence of allow or block *rules* which are evaluated sequentially and applied atomically by the security monitor. The flowwall is initialized with an implicit default rule (R0) which blocks all possible flows, i.e., no app will be able

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

Figure 5: Example of a policy for the scenario in Figure 2.

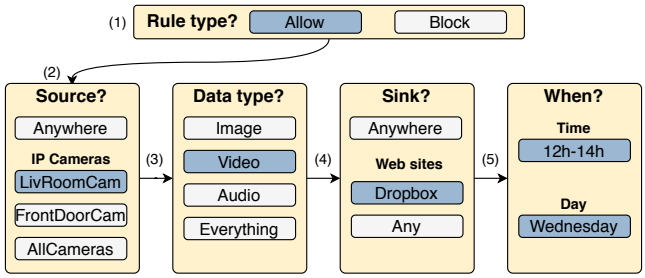


Figure 6: Schematic representation of the UI workflow to specify rule R4 (see Figure 5). To avoid overwhelming the user with too much information, in step 2, the system only displays existing valid source endpoints. Once the user selects the source endpoint (3), PatrIoT shows only the data types that can be generated by that endpoint. Similarly, in step 4, only valid sink endpoints are displayed.

to communicate unless R0 is overridden by a user-defined security policy. Next, we illustrate how security policies are specified.

Overview by example: Unauthorized sensor data sharing with Internet destinations or arbitrary mobile phones may lead to potential data exfiltration. Figure 5 shows a simple policy that aims to whitelist the web and mobile endpoints considered to be trustworthy for the hypothetical scenario presented in Figure 2. It contains five rules (R1-R5) which are interpreted sequentially. The policy first overrides R0 by allowing flows of any kind to occur (R1), and then blocks all flows to the web and to mobile endpoints (R2 and R3); this allows only data flows to occur within the home environment. Next, two exceptions are opened: R4 lets camera frame images to be collected from the living room's camera and uploaded to the user's Dropbox account during a certain time of the day (e.g. when the cleaning staff has access to the house), and R5 allows sensor data flows to the user's own mobile phone.

Rule syntax: In general, the format of a rule is as follows:

allow | **block** ⟨data type list⟩ **from** ⟨source endpoint list⟩ **to** ⟨sink endpoint list⟩ [**at** ⟨time period list⟩]

The keywords "allow" or "block" indicate the *rule type*, i.e., whether the rule allows or blocks the data flows matched by the rule, respectively. The data type list indicates one or multiple comma separated types of data to be matched. They can be simple types, e.g., Video, or the wildcard Everything to indicate all possible simple types. The keywords "from" and "to" are followed by a list of source and sink endpoints, respectively, which may specify individual endpoints, e.g., LivRoomCam, and/or include wildcards, such as Anywhere for all valid endpoints, and driver-specific terms, e.g., IPCamera to refer to all IP camera endpoints. Optionally, it is possible to specify time restrictions by using the keyword "at" followed by a time period, e.g., "12:00-14:00", and days of the week.

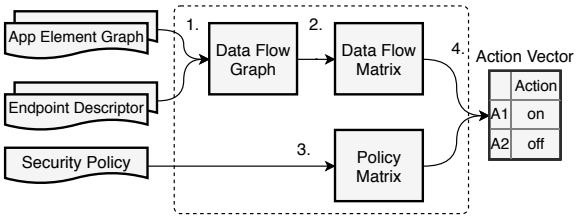


Figure 7: Data structures operated by the security monitor. The policy evaluation algorithm is executed strictly on occasions where the configuration of data flows or policy restrictions may change.

User interface for policy specification: Specifying policy rules using the syntax presented above can be cumbersome for untrained users. To help with this procedure, PatrIoT's management app provides a simple UI that guides the user along a five step process (see Figure 6) which helps the user to reason in terms of privacy-sensitive/insensitive data flows he intends to allow/block. To create a new rule, he starts by selecting the rule type, i.e., "allow" or "block" (1). Then, he picks the source endpoint (2), tells what data types from that source he wants to allow or block (3), indicates the sink endpoint (4), and optionally provides a temporal restriction for the rule (5). In a different UI view, the user can manage the security policy, list all rules, change their order, modify them, or delete them.

3.4 Policy Enforcement

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

Policy evaluation algorithm: Figure 7 shows the inputs, the output, and intermediate data structures generated by the policy evaluation algorithm. For inputs, it takes the element graphs of all installed apps, descriptors of existing endpoints, and the security policy. Based on these inputs, the algorithm generates two data structures which aim to model all possible data flows generated by the apps – the *data flow graph* and the *data flow matrix*; as well as a data structure that expresses the policy rules in an efficient manner – the *policy matrix*.

To explain how the algorithm works, consider the scenario of Figure 2. Assume that PatrIoT is configured with the policy shown in Figure 5 and that only two of the apps are installed: WatchMyHouse (A1) and PhotoBurst (A2). To ease the explanation, we follow the algorithm along the four steps shown in Figure 7, assuming that the intermediate data structures are built from scratch:

1. Modeling of data flows: The security monitor generates a model of all data flows that can potentially exist. This model consists of a set of Prolog predicates that specify a global *data flow graph* (DFG) based on the installed apps and existing endpoints. Figure 8 represents the resulting DFG for our example scenario. Nodes

consist of the aggregate elements (represented as boxes) pertaining to all installed apps (A1 and A2) and the endpoints that these apps have access to (represented in circles). Directed edges connecting two nodes n_1 and n_2 indicate that data can flow from n_1 to n_2. The type of data and its provenance is indicated in the labels attached to the edge. Each label consists of a pair $\langle d, e \rangle$ which indicates the data type d and its provenance e, i.e., d's source endpoint.

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

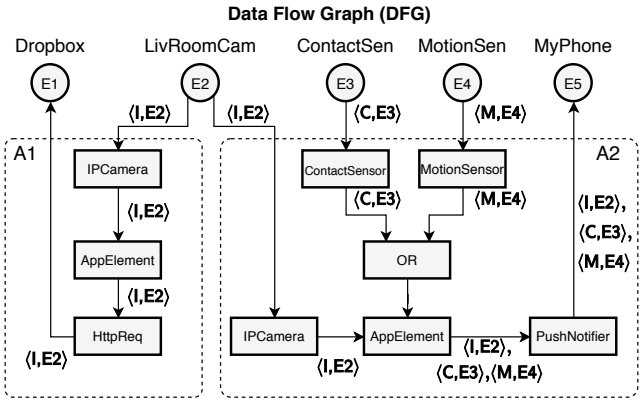
If n_1 and n_2 are both elements, then the edges reflect the connections of the respective app's element graphs and the possible types of data that can be transferred through these connections. These data types are indicated by the label attached to the edge and determined by the output of element n_1.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

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

2. Extraction of data flows: The DFG model is used to determine all possible data flows between source and sink endpoints, and record that information in the form of a (sparse) data flow matrix (DFM). The resulting matrix for our example scenario is shown in Figure 8. Rows and columns indicate source and sink endpoints, respectively. DFM$[e_1, e_2]$ is empty if no flow exists from e_1 to e_2; otherwise, it contains a list of pairs $\langle d, a \rangle$ which indicate the data type d that can flow between them and identify the app a responsible for that flow. To build this matrix, the security monitor executes a DFG Prolog query which computes the labels of the ingress edges of every sink e_2. From these labels, d and e_1 are extracted; from the element linked to e_2, the app a is identified.

3. Expansion of the policy rules: Before the final stage of policy evaluation, it is necessary to create an adequate representation of the security policy that allows to match the policy rules against the data flows described in the DFM. In particular, it is necessary to properly parse the references to groups of endpoints (e.g., Anywhere) and take into account the temporal restrictions in the rules (if any). This is the role of the Policy Matrix (PM) shown in Figure 8.

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



Data Flow Matrix (DFM)

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

Policy Matrix (PM)

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

Figure 8: Intermediate data structures for policy evaluation in the running example. The DFM shows that the images from the user's living room camera (E2) can be sent to Dropbox via A1 or to his mobile phone via A2, and that all possible destinations of motion and contact sensor readings are limited to his mobile phone only via A2. The PM shows that the action value for R4 is *off*: this means that this PM version is covering a time span where that targeted flow is not allowed, i.e., outside the 12h-14h time slot on Wednesdays.

3.5 Data Flow Graph Model Generation

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

OTP predicates for trusted elements are statically defined as part of the PatrIoT API. For each trusted element of the API, along with its JavaScript implementation, there is an accompanying file containing the element's OTP predicates. For instance, an OTP for an `IPCamera` element can be expressed as follows:

IPCamera of app a_1 **can output to** port `OutputFramePort` label $\langle \text{Image}, e_k \rangle$ **if** $e_k \in$ {valid source IP Camera endpoint}, **and** user gave a_1 the permission to access e_k.

For an untrusted element it is not possible to specify an accurate OTP predicate that reflects the actual behavior of the element. Consequently, untrusted elements are modeled as *'funnels'*, i.e., the labels from all the element's inputs will be forwarded to every single output port. This behavior can be expressed as follows:

Untrusted element u **can output to** every output port p^{out} the label set L, such that $L = \bigcup${labels received from every input port p^{in}} **and** p^{out} and p^{in} belong to u.

Thus, in Figure 8, e.g., we can see that the untrusted elements in apps A1 and A2 – both named `AppElement` – can propagate all labels between their respective input and output ports.

When generating the DFG, the security monitor loads the OTP predicates for trusted and untrusted app elements into the DFG model. Based on these predicates, the security monitor can model the tainted labels propagation within the app. Finally, to determine all the data flows between any given source and sink endpoints, PatrIoT uses another predicate:

Data type X **can flow** from endpoint e_1 to endpoint e_2 **if** exists a label $l = \langle \text{X}, e_1 \rangle$ such that l reaches e_2.

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

4 IMPLEMENTATION

We implemented a full prototype of the PatrIoT system. In total, we wrote ~20K lines of JavaScript code. The TSAR container was built using a SCONE Docker image featuring a Node.js v.8.9.4 binary cross-compiled against SCONE libs. Node.js includes a native addon that implements a Prolog query engine based on SWI-Prolog v.7.7.8. We developed in total 17 drivers responsible for the implementation of 35 trusted elements. The management app consists of a React-based frontend that serves a dynamic web application to connected clients. The backend was implemented as a REST API server provided by the runtime manager of the TSAR service.

To sandbox untrusted app elements, we rely on the VM2 implementation of a VM sandbox module for Node.js. Sandboxed code cannot import external modules, nor any global variables or classes from the main PatrIoT context.

5 EVALUATION

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

5.1 Case Study

To evaluate our system, we recreate the smart home scenario displayed in Figure 9. This home belongs to a family of three: Samantha, John and their baby. A nanny comes occasionally to babysit. There is also a predefined schedule for cleaning staff to access the home.

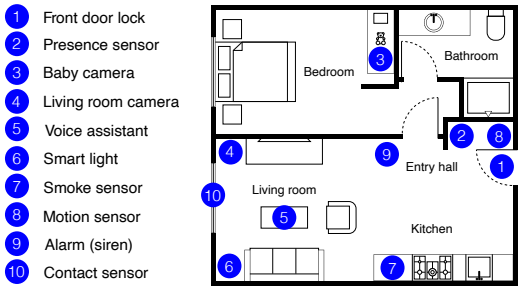


Figure 9: Emulated smart home setup.

We emulated ten devices (see Figure 9). The front door lock was emulated using an Arduino-based contact sensor. Presence, motion, smoke sensors and an alarm were emulated using corresponding Arduino-based sensors (HC-SR501 PIR, MQ-2, piezo buzzer). IP cameras were emulated using a USB camera attached to a Raspberry Pi device and streaming an MJPEG video. The same Raspberry Pi equipped with a microphone and a speech-recognition software running on it was used to emulate a voice assistant device. Finally, we used a Philips Hue light bulb as a smart light device.

We implemented a total of 20 PatrIoT apps that feature a device-to-device, device-to-mobile and device-to-web interaction, as well as a set of voice-activated apps that can either interact with local devices or web services.

5.2 Performance

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

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

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

To analyze the performance of PatrIoT apps we used a benchmark based on the use-case apps described in Section 5.1. We measured the time it took to execute a complete app data flow graph: from the time a trigger event was generated until the time it was fully processed by the app. We also measured the Prolog query time for each app's DFG model. This is the most time consuming step of the policy enforcement algorithm (see Section 3.4).

For our testbed, we used two servers running 64bit Ubuntu 18.04.4 LTS with a 16-core 3.60GHz Intel i9-9900K CPU and 16GB

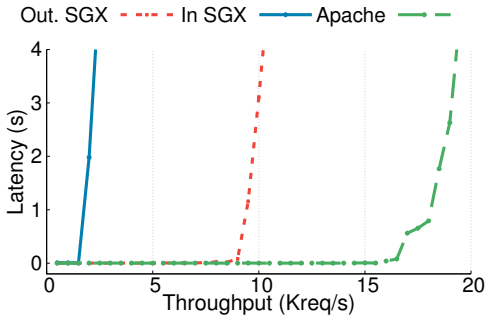


Figure 10: Throughput versus latency evaluation.

Table 1: Attestation, bootstrap, and app loading times.

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

of RAM. We adopted the 19.03.9 version of Docker engine to run PatrIoT. PatrIoT core files inside a Docker image were encrypted using SCONE's File Shield. We evaluated the performance of PatrIoT running inside and outside of SCONE SGX separately. Obtained values were averaged across 20 runs.

System initialization time: Table 1 presents PatrIoT's attestation, bootstrap and app loading times. It takes on average 13.5 seconds to attest PatrIoT running inside an SGX SCONE enclave. Most of this time is taken by communication with a CAS server and decryption of PatrIoT core files after a successful attestation. Additional delay comes from the fact that SCONE needs to allocate the required memory resources at enclave start time which depending on the specified heap size might take more time. However, considering that PatrIoT is a server component which needs to be started only once and run continuously such a one-time delay can be tolerated. The bootstrap time overhead of using SGX is just 10 ms which is mostly caused by enclave transitions during system calls. The app loading time overhead reaches 118 ms, which is the time it takes to decrypt the app files in the container's encrypted file system.

Load test: Figure 10 features the results of PatrIoT server test when run inside and outside SGX SCONE enclave. PatrIoT Server performed similarly in both settings until the load reached 1900 requests per second, at which point the latency of the PatrIoT's SGX version started to degrade. The standalone version of PatrIoT reached a saturation point at around 9000 requests per second. Since many smart devices generate low-rate network traffic, this limit is acceptable. We observed nearly 5x performance loss when running PatrIoT inside an SGX SCONE enclave. This is consistent with the original reports by SCONE authors [1]. SCONE is not optimized for network-intensive applications like PatrIoT. Apache outperformed the TSAR service, since the former is multi-threaded, while the latter's Node.js engine is single-threaded.

Application performance: The left side of Figure 11 displays the execution times for each use-case app. Execution times are tightly dependent on each app's workload, ranging between 32 and 690 ms

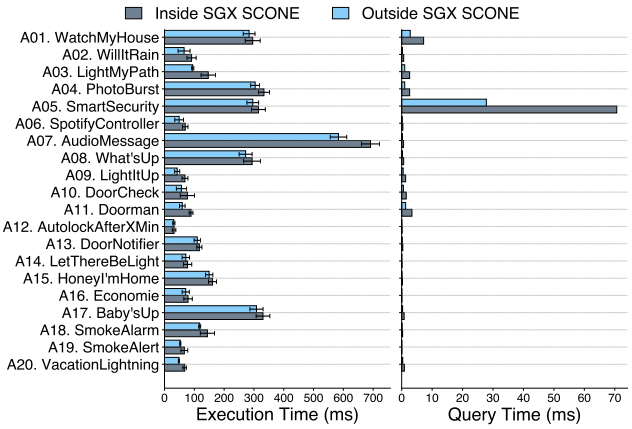


Figure 11: PatrIoT app benchmark performance.

(inside SCONE). Apps that send sensor data to the Internet or as part of the push notification (e.g. `AudioMessage`, `Baby'sUp`) often have the highest execution time due to the network latency and the data transfer rate. The right side of Figure 11 features the time needed to execute a Prolog query and extract flow information from a given app's DFG. The average query time is 4.7 ms and 1.84 ms (inside and outside SGX SCONE) for the apps with a simple DFG. If an app has a DFG with multiple data sources, Prolog's backtracking mechanism requires more time to inspect all possible data flows, e.g., `SmartSecurity` which consists of 11 elements 6 of which emit different data types. While its query time is in a stark contrast to the other apps it is still below 70 ms.

5.3 Policy Expressiveness

To assess the expressiveness of PatrIoT's flowwall policies, we have written several allow / block rules that make sense for our smart home scenario (see Figure 12). The first three block rules (RB1-RB3) are the most restrictive: RB1 blocks all the app flows, RB2 only blocks flows to the Internet, and RB3 blocks flows to mobile endpoints. RB4 rule displays how a S2S flow (see Section 2.2) can be effectively blocked. Rules RB5-RB6 prevent the most privacy sensitive data flows (voice assistant and baby cam) to the Internet. The allow rules (RA1-RA6) start with RA1, which allows all possible flows, followed by more restrictive ones based on certain conditions.

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

Figure 12 shows the results of these rules applied to our use-case apps. If we disregard an RB1 rule we can see that the majority of

Rule	A01	A02	A03	A04	A05	A06	A07	A08	A09	A10	A11	A12	A13	A14	A15	A16	A17	A18	A19	A20
RB1. **block** Everything **from** Anywhere **to** Anywhere																				
RB2. **block** Everything **from** Anywhere **to** Web																				
RB3. **block** Everything **from** Anywhere **to** Phone																				
RB4. **block** PresenceInfo **from** PresenceSensor **to** SmartLight																				
RB5. **block** Audio **from** SmartAssistant **to** Web																				
RB6. **block** Everything **from** BabyCam **to** Web																				
RA1. **allow** Everything **from** Anywhere **to** Anywhere																				
RA2. **allow** Image **from** LivRoomCam **to** Dropbox **at** 12:00-14:00,Wednesday																				
RA3. **allow** Everything **from** LivCam,Alarm,Smoke/Contact/Motion Sens. **to** ADTSecurity																				
RA4. **allow** Command **from** SmartAssistant **to** Spotify, NYTimes, BBCWeather																				
RA5. **allow** Everything **from** Anywhere **to** John'sPhone, Samantha'sPhone																				
RA6. **allow** Everything **from** BabyCam **to** Nanny'sPhone **at** 9:00-17:00, weekdays																				

Figure 12: Summary of policy evaluation for use-case apps. Red ( ) and green ( ) cells denote blocked or allowed apps respectively, apps with yellow cell ( ) are conditionally blocked, empty cell means the app flows are out of the rule's scope.

apps can operate nominally with all other rules in place. In fact, all of these apps operate with device-to-device flows which are usually deemed less privacy sensitive as compared to those that span across different domains (Internet, mobile, etc.). A quarter of apps that issue calls to mobile or web endpoints can be affected by rules RB2 and RB3. However, a set of custom endpoint-based allow rules could be added by the user to allow these apps' data flows.

5.4 Usability

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

Methodology: We conducted a two stage user study with 45 participants (computer department employees) with a goal to determine common privacy concerns of the smart device users, and their ability to express these concerns within a PatrIoT's UI. In a first stage, the participants were given the smart home scenario described in Figure 9 and asked to decide if a given device data flow should be allowed, blocked, or allowed only in a certain condition. All three data flow types were exercised: S2M, S2W, and S2S. The second stage of the survey was more practical. With the PatrIoT mobile app the participants had to register a new user account, define policy rules for a baby camera device, and then verify a given app data flows against those rules. In the end we asked the participants to tell us about their experience with PatrIoT.

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

As part of a second task, the participants were asked to decide if motion sensor data can flow to smart lights. On the one hand, the majority of participants (54.3%) decided to restrict such a data flow to a certain time of the day (when the user is at home). On the other hand, others (43.5%) decided to allow such a flow without any restrictions. Finally, only 2.2% opted for blocking it. The results are

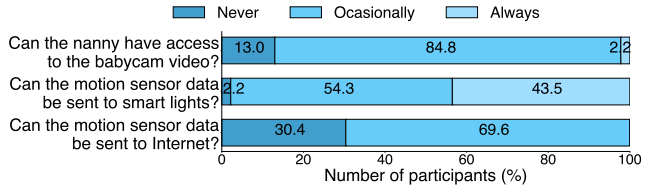


Figure 13: Survey results: privacy preferences.

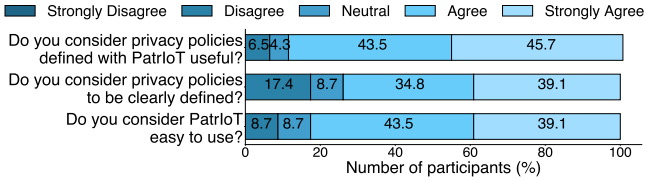


Figure 14: Survey results: user experience.

in line with our expectations: people are less concerned with the device-to-device data flows taking place within their home domain as long as there are no mobile or Internet endpoints involved.

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

Findings about user experience: Figure 14 presents our main findings in the second survey stage. The majority of participants agreed that PatrIoT rules are useful in protecting user privacy (89.2%). Only a small fraction remained neutral (4.3%) or disagreed with this statement (6.5%). These results highlight PatrIoT's ability to express smart home user privacy preferences of various complexity in a clear and practical way.

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

the average user to use from the start. These rules could be added automatically based on the connected devices.

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

6 SECURITY DISCUSSION

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

Currently, our system is dependent on a relatively large trusted computing base (TCB). In particular, PatrIoT's TCB comprises its API and runtime code, Node.js, Javascript engine, and SCONE's library. In spite of this, the total TCB size is comparable with other SGX-based systems, e.g. [10, 13, 15]. Studying ways for reducing the TCB size constitutes an interesting avenue for future work.

Finally, PatrIoT may be vulnerable to recently demonstrated side-channel attacks on SGX. While we consider such attacks to be out of scope, PatrIoT can take advantage of various mitigation techniques, e.g. [12], or use an alternative TEE, e.g. ARM TrustZone.

7 RELATED WORK

TEEs and SGX-shielded execution have been proposed to secure various workloads on untrusted clouds [10, 16]. PatrIoT is the first system that leverages these techniques to provide IoT service backend protections. Similarly to PatrIoT, others have also suggested to use IFC techniques for prevention of IoT privacy breaches in the cloud [17]. However, in contrast with our work, these authors propose classic IFC models operating at a very low level of abstraction and assume trusted platform providers.

A large number of systems has been proposed focusing on security and privacy of existing IoT smart platforms [3, 18, 19]. PatrIoT complements these systems by focusing exclusively on detection and prevention of privacy-sensitive data flows. Other systems offer mechanisms for tracking information leakage [4, 11]. However, in these systems the security policies are defined per app, which prevents tracking information flows across multiple apps. PatrIoT overcomes these limitations by providing an original IFC model that can globally track flows across all apps in home.

Lastly, we highlight two systems that propose data flow programming for home hub devices [8, 20]. Flowfence [8] allows IoT app developers to split their apps into modules that operate with sensitive data sources and those that do not, and to track the data flows between those parts. However, in contrast to PatrIoT, Flowfence employs dynamic taint techniques that are vulnerable to timing side channel attacks. From HomePad [20], we borrowed the idea of modeling IoT apps as an element flow graphs, and employing Prolog rules for checking privacy policies. However, HomePad's analysis is limited to per-app policies, which are harder to manage as the number of devices and apps grows. PatrIoT goes beyond HomePad by introducing a full fledged flowwall security monitoring abstraction that provides holistic control of the home environment.

8 CONCLUSIONS

We presented PatrIoT, a private-by-design IoT platform, which ensures secure data processing by leveraging SGX secure enclaves. PatrIoT's flowwall security monitor allows end-users to obtain fine-grained control of data flows generated by IoT devices, and prevent potential privacy violations by enforcing a privacy policy.

Acknowledgments: We thank the reviewers for their comments. This work was supported by funds provided by Fundação para a Ciência e a Tecnologia (FCT) (UIDB/50021/2020 project), Erasmus Mundus fellowship, and a CISCO research grant. We thank Etienne Riviere for the testbed access.

REFERENCES

[1] Sergei Arnautov, Bohdan Trach, Franz Gregor, et al. 2016. SCONE: Secure Linux Containers with Intel SGX. In *Proc. of OSDI*.
[2] Bram Bonné, Sai Teja Peddinti, Igor Bilogrevic, and Nina Taft. 2017. Exploring decision making with Android's runtime permission dialogs using in-context surveys. In *Proc. of SOUPS*.
[3] Z Berkay Celik, Patrick McDaniel, and Gang Tan. 2018. SOTERIA: Automated IoT safety and security analysis. In *Proc. of USENIX ATC*.
[4] Z. Berkay Celik, Gang Tan, and Patrick McDaniel. 2019. IoTGuard: Dynamic Enforcement of Security and Safety Policy in Commodity IoT. In *Proc. of NDSS*.
[5] Pardis Emami-Naeini, Henry Dixon, Yuvraj Agarwal, et al. 2019. Exploring how privacy and security factor into IoT device purchase behavior. In *Proc. of CHI*.
[6] Adam Clark Estes. [n.d.]. Yes, Your Amazon Echo Is an Ad Machine. https://gizmodo.com/yes-your-amazon-echo-is-an-ad-machine-1821712916.
[7] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. 2016. Security analysis of emerging smart home applications. In *Proc. of IEEE S&P*.
[8] Earlence Fernandes, Justin Paupore, et al. 2016. Flowfence: Practical data protection for emerging iot application frameworks. In *Proc. of USENIX Security*.
[9] Christine Hauser. [n.d.]. Police Use Fitbit Data to Charge 90-Year-Old Man in Stepdaughter's Killing. https://nyti.ms/2Oz8P5j. Accessed August 2020.
[10] Tyler Hunt, Zhiting Zhu, Yuanzhong Xu, et al. 2016. Ryoan: A Distributed Sandbox for Untrusted Computation on Secret Data. In *Proc. of OSDI*.
[11] Yunhan Jack Jia, Qi Alfred Chen, Shiqi Wang, Amir Rahmati, Earlence Fernandes, Z. Morley Mao, and Atul Prakash. 2017. ContexIoT: Towards Providing Contextual Integrity to Appified IoT Platforms. In *Proc. of NDSS*.
[12] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX enclaves from practical side-channel attacks. In *Proc. of USENIX ATC*.
[13] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. Safebricks: Shielding network functions in the cloud. In *Proc. of NSDI*.
[14] Threat Post. 2019. Amazon Sends 1,700 Alexa Voice Recordings to a Random Person. https://threatpost.com/amazon-1700-alexa-voice-recordings.
[15] Christian Priebe, Kapil Vaswani, and Manuel Costa. 2018. Enclavedb: A secure database using SGX. In *Proc. of IEEE SP*.
[16] Nuno Santos, Rodrigo Rodrigues, Krishna P. Gummadi, and Stefan Saroiu. 2012. Policy-Sealed Data: A New Abstraction for Building Trusted Cloud Services. In *Proc. of USENIX Security*.
[17] Jatinder Singh, Thomas Pasquier, Jean Bacon, Julia Powles, Raluca Diaconu, and David Eyers. 2016. Big Ideas Paper: Policy-driven Middleware for a Legally-compliant Internet of Things. In *Proc. of Middleware*.
[18] Yuan Tian, Nan Zhang, Yueh-Hsun Lin, XiaoFeng Wang, Blase Ur, Xianzheng Guo, and Patrick Tague. 2017. Smartauth: User-centered authorization for the internet of things. In *Proc. of USENIX Security*.
[19] Qi Wang, Wajih Ul Hassan, Adam Bates, and Carl Gunter. 2018. Fear and Logging in the Internet of Things. In *Proc. of NDSS*.
[20] I. Zavalyshyn, N. O. Duarte, and N. Santos. 2018. HomePad: A Privacy-Aware Smart Hub for Home Environments. In *Proc. of SEC*.
[21] Wei Zhou, Yan Jia, Yao Yao, et al. 2019. Discovering and Understanding the Security Hazards in the Interactions between IoT Devices, Mobile Apps, and Clouds on Smart Home Platforms. In *Proc. of USENIX Security*.