Scalable Metadata-Hiding for Privacy-Preserving IoT Systems

Yunang Chen
University of Wisconsin–Madison
ychen459@wisc.edu

David Heath
University of Illinois Urbana–Champaign
daheath@illinois.edu

Rahul Chatterjee
University of Wisconsin–Madison
rahul.chatterjee@wisc.edu

Earlence Fernandes
University of California San Diego
efernandes@ucsd.edu

ABSTRACT

Modern cloud-based IoT services comprise an integrator service and several device vendor services. The vendor services enable users to remotely control their devices, while the integrator serves as a central intermediary, offering a unified interface for managing devices from different vendors. Although such a model is quite beneficial for IoT services to evolve quickly, it also creates a serious privacy concern: the vendor and integrator services observe all interactions between users and devices. Toward this, we propose Mohito, a privacy-preserving IoT system that hides such interactions from both the integrator and the vendors. In Mohito, we protect both the interaction data and the metadata, so that no one learns which user is communicating with which device. By utilizing oblivious key-value storage as a primitive and leveraging the unique communication graph of IoT services, we build a scalable protocol specialized in handling large concurrent traffic, a common demand in IoT systems. Our evaluation shows that Mohito can achieve up to 600× more throughput than the state-of-the-art general-purpose systems that provide similar security guarantees.

KEYWORDS

Internet of Things, privacy, metadata protection, anonymous communication

1 INTRODUCTION

IoT devices, ranging from smart home appliances, such as thermostats and security camera systems, to fitness trackers and medical equipment, have become integral to the daily lives of many individuals. These IoT devices are connected to backend servers operated by their manufacturing vendors, enabling users to remotely interact with them through smartphones or browsers. Such interactions however allow vendors to accumulate extensive data about the activities of their devices and users.

Studies [25, 60] have shown that users are generally concerned about the privacy implications of data collected from IoT devices, as it can be utilized to infer sensitive aspects of users' lives. For example, health and fitness trackers can record users' physical activities or sleep patterns, and may even expose details about users' overall health, daily routines, and potential medical conditions. The privacy risks associated with IoT data call for a robust privacy protection mechanism that hides interactions between users and devices from vendor servers.

To enhance the data privacy of IoT systems, recent works [22, 24] have focused on encrypting data communicated through the systems. However, the mere *existence* of communication can still pose privacy risks. Such metadata associated with communications is often considered a viable means for surveillance [47], and encrypted messaging services like Signal are adding multiple layers of metadata protection to conceal who is communicating with whom for privacy-conscious users [1]. This problem is even more pronounced in IoT systems, as metadata can be used to infer the actual data. Consider a scenario where the server sees a message from a user to a smart door lock. Even if it does not know the content of the message, it may still deduce what the user is sending, as the types of the command are typically limited to locking and unlocking. Indeed, researchers [17, 49] and government regulations [9, 10] have warned that IoT metadata can be utilized to infer user data and therefore must be protected. For example, the Office of the Privacy Commissioner of Canada issued guidance to IoT vendors stating that, to adhere to Canada's federal privacy law, vendors should categorize metadata as personal information for privacy protection purposes [9].

Thus a major challenge in designing privacy protection mechanisms for IoT systems is *metadata* privacy. Although a long line of works has proposed anonymous communication systems that provide metadata privacy [37–39, 51–53], their system model and assumptions do not work for the IoT setting. These general-purpose systems typically assume that servers are pairwise connected, but in modern IoT systems, vendors do not share a direct line of communication. Instead, they all connect to a centralized service known as the *integrator* service. Integrator services, such as Amazon Alexa [2], Google Home [5], and IFTTT [6], are cloud-based IoT platforms that serve as a bridge between users and vendors by collecting commands from users and forwarding these commands to the corresponding vendors. Integrators play an essential role in modern IoT ecosystems, as they unify the heterogeneous communication interfaces of different vendors and streamline device management. Therefore, a privacy-preserving IoT system should abide by the communication structure that centers around the integrator. This structure presents challenges for hiding metadata, as we must ensure that the integrator can relay messages between users and vendors efficiently and accurately, while never allowing it to learn the identities of message senders and receivers.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–16

https://doi.org/XXXXXXX.XXXXXXX

In addition, an IoT system typically supports millions of devices deployed worldwide and generates a significant amount of concurrent traffic. This extensive data flow requires an efficient protocol that emphasizes high throughput. However, the cryptographic primitives used in many metadata-hiding systems are not designed to handle messages in large batches [27, 30, 32], making them less compatible with the demands of the IoT setting. Furthermore, the majority of such IoT traffic is generally caused by devices belonging to a few large vendors. The number of devices operated by smaller vendors is only a fraction of what larger vendors have [44]. We should assume that these smaller vendors only have the infrastructure capable of supporting traffic for their own devices. Many general-purpose metadata-hiding protocols split the system processing load evenly across all servers; if we were to apply such protocols in an IoT system, it would overburden smaller vendors. Hence, one must ensure that a vendor's operational cost scales with its number of devices.

Motivated by the above challenges, we propose Mohito, a privacy-preserving IoT system that hides user/device interaction metadata from vendors and from the integrator service. In Mohito, users transmit commands to devices through the IoT cloud servers, and devices respond with status updates. In addition to preventing the IoT servers from deciphering the contents of commands or responses, Mohito ensures they do not learn the metadata, i.e., which user is interacting with which device. Specifically, the integrator does not know the destination of each user's command, while the vendor does not know the source of the command each device receives (and vice versa for responses).

Mohito achieves high throughput by leveraging the centralized structure of IoT systems. At a high level, we organize communication into rounds and, in each round, the integrator gathers commands from users into a batch. Then the integrator utilizes an oblivious key-value store (OKVS) [34, 45], a cryptographic primitive that allows efficient and private batch-encodings without decoders learning which key-value pairs are encoded. For each batch of commands, the integrator processes and encodes them into several OKVSs based on the destination vendor of each command. These OKVSs are forwarded to the corresponding vendor, which decodes the commands and sends them to the target devices. Our protocol ensures that the vendor does not learn the source of the commands they have decoded.

Mohito also protects metadata information from an honest-but-curious integrator. We achieve this by first ensuring our privacy guarantee holds in a single round, and then we prevent cross-round attacks. For the first step, Mohito instructs vendors to shuffle commands for the integrator. By outsourcing the shuffling process to a vendor, we ensure that, within each round, the integrator cannot trace a command back to its user.

Like other communication systems, IoT systems are susceptible to cross-round attacks, also known as intersection attacks. In an intersection attack, the servers observe traffic patterns over multiple rounds of communication to infer relationships between message senders and receivers [28, 36, 42]. Defending against intersection attacks is integral to protecting metadata. Specifically in an IoT system, the integrator observes two pieces of information: (1) which users send a command and (2) how many commands are sent to each vendor. By recording this information over multiple rounds, the integrator can infer which users communicate with which vendors, breaking our privacy goal. Therefore, in Mohito, when a vendor shuffles the commands, it also injects a number of fake commands to hide the traffic pattern. We design the injection protocol in a way that the integrator cannot learn how many commands each vendor actually receives in each round, even if the integrator controls a small number of users and devices.

We implement and benchmark Mohito. As our main performance goal is to handle highly concurrent traffic, we are primarily interested in the system's throughput. We estimate that our proof-of-concept implementation can handle 24,000 commands per second. For comparison, Express [32], the state-of-the-art general-purpose metadata-hiding system, can handle only 40 messages per second under similar settings.

In sum, we offer the following contributions:

- We derive the desired properties of a privacy-preserving IoT system, in terms of security and real-world considerations for efficiency, which demands a different set of design requirements than general-purpose metadata-hiding communication systems.
- Based on these properties, we design Mohito, a metadata-hiding communication system tailored for privacy-preserving IoT interactions.
- We build a proof-of-concept implementation of Mohito and evaluate its performance to show Mohito's high throughput.

2 BACKGROUND & MOTIVATION

2.1 IoT Ecosystems and Privacy Concerns

The widespread adoption of IoT devices has led users to own multiple devices from various manufacturing vendors, each specializing in a particular product category or functionality. For example, a user of smart home devices may have bought a smart light bulb from Phillips Hue, a smart thermostat from Nest, and then a smart oven from LG. Such diverse device provenance presents challenges in terms of device management, interoperability, and user experience To address these challenges, users often leverage *integrator services*. An integrator service – e.g., Amazon Alexa, Google Home, IFTTT – is a centralized cloud platform that allows users to remotely interact with their devices through a unified interface, regardless of the device vendor and communication protocol. Therefore, integrator services have become an essential part of modern IoT ecosystems. Fig. 1 depicts the dataflow in these systems.

Privacy concerns. Whenever users remotely interact with their IoT devices, they inadvertently expose their activity data to the corresponding vendors. For example, a vendor that manufactures smart home security systems can collect the entry and exit times of every person in the user's home. Due to the nature of many IoT devices, such information can be sensitive, as it can reveal details of the user's lifestyle and habits, including sleeping patterns, child behaviors, medical information, and sexual activities. Therefore, many users are worried about the privacy implications of IoT devices [25, 60].

Integrator services, while providing many benefits to users, pose even greater privacy concerns. To allow unified access to devices, integrator services require device permissions. Users typically provide

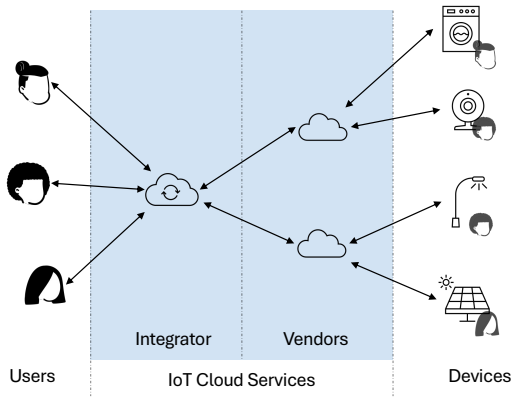

Figure 1: Overview of IoT Ecosystem. Users communicate with an integrator, which communicates with various vendors. Each vendor communicates with its own devices.

permissions via login credentials, authorizing integrator services to interact with devices on their behalf. Then, when a user issues a command to a device through the integrator service, the integrator service uses its permissions to forward the command to the vendor associated with the device. Therefore, the integrator service gains unfettered access to all of a user's IoT activities, allowing it to accumulate a more comprehensive view of the user's personal life.

2.2 Towards a Privacy-Preserving IoT System

The above privacy concerns motivate the design of IoT systems that provide quality user experience while also *preserving user privacy*. In a privacy-preserving IoT system, neither the vendor nor the integrator should learn of the user's interaction with their devices. There are four main challenges in designing such system.

Metadata-hiding. Privacy protection for metadata is crucial for many types of communication systems [1, 46], and it holds the same importance for IoT systems [9, 10, 17, 49]. It often does not suffice to simply encrypt messages; the mere existence of messages from a user to a vendor can compromise privacy. Consider a scenario where a user owns a smart door lock manufactured by August. The system could try to hide the user's interaction data with end-to-end encryption. Nevertheless, the integrator service can still observe that the user sends a message to August. In this case, August – like many other IoT vendors – manufactures only one type of device, so the integrator implicitly knows the user is communicating with a smart lock. By viewing the existence of a message in context (e.g., the user sends the command in the evening), the integrator service can piece together troubling information about the user (e.g., the user is likely unlocking their door as they come home from work). Therefore, such *metadata* information can lead to privacy leakage.

Scalability. A key characteristic of IoT systems is the scale of data. With millions of IoT devices deployed worldwide, they collectively generate a significant amount of traffic every second. Therefore, it is crucial for our system to focus on scalability first to accommodate the continuous growth of devices in the IoT landscape. We note that there are many works [27, 32] on anonymous communication systems with similar security goals, but they mostly focus on single message performance, whereas we need to handle large volumes of concurrent messages and achieve high throughput.

Load-balancing. In many anonymous communication systems, all server nodes are assumed to have similar processing powers. However, we cannot make the same assumption for vendors in IoT systems. Based on the dataset in [44], the top 20 vendors in IFTTT on average have 3,130 times more connected devices than the bottom 20, which means that, if we were to split the processing load equally, smaller vendors may not have the resources prepared to handle the overhead caused by larger vendors' traffic. Therefore, we need to ensure the overhead of each vendor scales proportionally to the number of devices it owns.

On-device operations. The computations a device can execute are often limited by factors such as the device's battery life and computational capacity. Therefore, we should ensure that the on-device overhead remains small and does not scale with the total number of users or devices in the system.

3 DESIGNING A PRIVACY-PRESERVING IOT SYSTEM

We present Mohito, a clean-slate design for privacy-preserving systems tailored to the IoT setting. To ensure Mohito can be adopted by existing IoT services, we adhere to these platforms' communication structure and API design when designing Mohito. In this section, we describe Mohito's system model as well as our threat model and security goals.

3.1 System Model

Following the design of today's commercial IoT ecosystems [2, 5, 6], we model the system as a set of IoT cloud services connected to various users and IoT devices (as shown in Fig. 1). The cloud services consist of a single integrator service and many device vendors.

Vendor. A device vendor, such as LG and Philips Hue, provisions IoT devices and periodically communicates with its devices. In this paper, we use this term to denote the backend server of a vendor.

Integrator. An integrator service, such as Alexa and IFTTT, is a cloud platform authenticated to connect to various vendors and issue commands on behalf of users. All communications between users and vendors pass through the integrator.

User. A user owns the devices in their home purchased from different vendors and controls those devices using an integrator-provided interface (e.g. a web interface or phone app). For simplicity, we consider the user and the interface as a single entity. We do *not* expect users to be always online; they only participate when they wish to send a command.

Device. Each device is owned by a single user and must be set up by the user before it comes online. Once set up, the device starts communicating with its vendor. We assume each device has a globally unique id, such as a MAC address, that is also known to the device's user and vendor (but not to the integrator).

We model the interaction between the user and its device as a single operation: the user first sends a command to its device and

then receives a status update from the device as a response. This interaction is achieved as follows: the user sends the integrator a message, which is forwarded to the appropriate vendor and then the device; similarly, the device's response goes through the vendor first, then the integrator, and finally the user. We believe this operation is generic enough to cover all basic functionalities in an IoT system. For example, if the user needs to check the current state of a device, they can issue a "read" command that tells the device to report its state in the status update response.

We do *not* allow communication between vendors. It is not realistic to assume that each vendor knows all other vendors, nor that a particular vendor should build infrastructure compatible with other vendors. This aligns with the practices of existing IoT platforms, as none of them requires a vendor to directly communicate with another vendor.

We model the system to proceed in a round-based fashion. Each round takes a fixed amount of time, during which the integrator gathers all messages from users participating in the current round and delivers them to the corresponding vendor, before advancing to the next round. This round-based approach closely aligns with the API design of today's commercial integrator services, such as IFTTT [11], which requires vendors to process messages in batches.

3.2 Threat Model

In our threat model, IoT services (i.e., the integrator and vendors) are *honest-but-curious*,[1] but users and devices can be *malicious*. In practice, IoT services are regulated such that they cannot deviate arbitrarily from a protocol. Even in cases where these services are attacked by an adversary, the attack often causes data breaches, rather than ceding control of the service. Users and devices are more vulnerable, and they might be fully compromised by an adversary.

Non-colluding services. Given that many vendors may be owned by the same company, we allow collusion among multiple vendors. However, we do assume that the integrator remains an independent platform and does *not* collude with any of the vendors. This is a common model adopted by many related security works on IoT integrators [13, 21, 22, 24, 33, 56, 58], due to the legal restrictions on sharing user data between different IoT services [4] and the low likelihood of an adversary successfully compromising two IoT services simultaneously.

In addition, the integrator or a vendor may register accounts with other vendors or buy devices from them to gain information about its competitors. Therefore, we assume some users and devices may collude with an IoT service. However, we assume that these colluding users and devices do not add more than δ messages (a parameter for our construction, see Section 5.2 for more on this) within a round.

Transparent communication. We assume that when two parties communicate, they learn each other's identity (through the IP address or as a username due to an authentication process). When a user contacts the integrator (or any IoT service), the integrator (or the IoT service) learns some identity of the user. We believe this assumption is necessary, as real-world services often rely on such identity to implement rate limiting or to prevent denial-of-service attacks [3, 7]. Note that other information, such as how many vendors the user uses, is not revealed to the integrator; otherwise, it trivially learns the destination of messages from users who only have one device/vendor.

Message size. Prior works have shown that the size of IoT messages can be used to infer device activities [18]. To prevent this, messages can be padded to some fixed size. Padding messages efficiently is an important, albeit orthogonal, research problem [15, 16, 18]. In this work, we assume that messages generated by users and devices are padded to some fixed length, such that it is not possible to identify a user or device based on the size of a message.

Public key infrastructure. We assume IoT services exist within a public key infrastructure. Each service has its own key pair, and anyone can verify the public key of each service.

3.3 Security Goals

Our goal is to support privacy-preserving IoT interactions such that IoT services learn minimal information about users. Commands/responses should be hidden, and IoT services should not learn which user communicates with which device. In more detail, our system provides the following properties:

Data privacy. No party – other than the intended recipient – learns the content of a particular message.

Metadata privacy. We hide the communication pattern between users and devices from IoT services. No IoT services should be able to learn which user is interacting with which device. Specifically,

- For the integrator, each time it receives a message from a non-colluding user, it should not learn which vendor is the target of this message.
- For a vendor, it should not learn which non-colluding devices receive commands in each round. We enforce this requirement because vendors already know the identities of their devices (per our threat model).

The metadata privacy must hold even when an IoT service can observe communications for an arbitrary number of rounds.

Data integrity. Although we assume IoT services are honest-but-curious, a malicious user/device can attempt to corrupt the integrity of messages intended for other users/devices. We ensure that the message sent by a benign user/device cannot be tampered with by another user/device.

We provide a more formal definition in Appendix C.

3.4 Potential Solutions and Challenges

We examine alternative solutions to design a privacy-preserving IoT system and analyze why they fail to tailor to the specific demands of IoT ecosystems.

Local Protocols. In designing Mohito, we prioritize compatibility with current industry practices to ensure better integration with existing IoT infrastructures. While local protocols that use direct user-to-device communication within the user's home network may provide better efficiency, they do not align with the prevalent operating models of IoT systems. As discussed in Section 2.1, the

[1] While we adopt the honest-but-curious model, we discuss how Mohito's protocol can be upgraded to the malicious setting in Section 8.

integrator functioning as a cloud-based service has become an indispensable part of today's IoT ecosystem, because it enhances user experiences by allowing remote device management through a centralized interface. Therefore, to facilitate easier adoptability with vendors that are already following this communication paradigm, we need to assume a cloud-based system model.

Mixnet-based Protocols. General-purpose anonymous communication protocols that rely on mix-nets [37, 38, 53] do not satisfy some of the key requirements in the IoT setting outlined in Section 2. First, all participating servers in a mix-net are expected to communicate with each other directly, whereas communications in IoT systems flow through a central integrator and peer-to-peer interactions between vendors are not practical. Second, mix-nets assume a uniform load distribution across all servers, but IoT systems are characterized by a diversity of vendors with vastly different levels of computational resources. Finally, mix-net protocols can introduce high latency due to the multiple rounds of mixing required to anonymize messages, which contradicts the low-latency requirement in IoT environments.

PIR-based Protocols. Some anonymous systems based on Private Information Retrieval (PIR), such as Riposte [27] and Express [32], can be adapted to fit within the IoT server structure. However, these systems are often designed with a different set of functionality requirements, some of which (such as message persistence) are not applicable in the IoT context and therefore can introduce unnecessary overheads. In particular, to satisfy these different requirements, these systems utilize underlying cryptographic primitives like distributed point function [35] that are not optimized for batch processing of messages. Consequently, their application in IoT systems, which is characterized by a large volume of concurrent traffic, can lead to performance bottlenecks.

4 OVERVIEW OF Mohito ARCHITECTURE

We now start delving into the design of Mohito. First in this section, we outline the building blocks of Mohito that allow it to achieve our security goals within a *single* round. Then we show how to extend the protocol to prevent cross-round attacks in Section 5 and finally provide the full detailed protocol in Section 6.

One of the fundamental building blocks of Mohito is an Oblivious Key-Value Store (OKVS) [34]. Section 4.1 reviews the OKVS primitive and discusses how it helps protect metadata in our system. However, using OKVS alone cannot satisfy our design goals. In the following two subsections, we discuss how we can overcome the drawbacks of a naïve OKVS approach by incorporating *shuffling* (Section 4.2) and *ephemeral ids* (Section 4.3).

For simplicity, we focus only on the part of the protocol where users send commands to devices and omit the part where devices send responses back, since the latter can generally be achieved by reversing the process of the former. We detail the latter in Section 6.3.

4.1 Oblivious Key-Value Stores

Background. A *key-value store* is an encoding of a set of key-value pairs, and is defined by two algorithms:

- Encode takes as input a set of key-value pairs $\{(k_1, v_1), \ldots, (k, v_n)\}$ and outputs a store S;
- Decode takes as input a store S and a key k, and outputs a value v.

A key-value store is *oblivious* if it hides the keys when the values are random. When invoking Decode on some key k_i used to generate S, the result is the corresponding v_i; for any other key, the result is a value that appears to be random. Therefore, an observer seeing calls to Decode cannot tell whether a particular key is in S or not. More formally, consider two OKVS structures encoding random values where the first structure has keys $\mathcal{K}_0$ and the second has keys $\mathcal{K}_1$. The key-value store is oblivious if it is infeasible to distinguish these two structures.

One classic OKVS construction uses a polynomial P satisfying $P(k_i) = v_i$. The coefficients of P represent the encoded values, and we can decode key k by simply evaluating $P(k)$. However, this approach is computationally expensive, as encoding and (batch) decoding of n items require polynomial interpolation, requiring $O(n \log^2 n)$ operations. Recent works [34, 45] construct cuckoo-hash-table-based OKVSs that achieve encoding and decoding at cost $O(n\lambda)$, where λ is the security parameter.

Strawman approach with OKVS. We describe a naïve design for a metadata-hiding IoT system based on OKVS. We assume the system proceeds in a round-based fashion. In each round:

(1) Each user who wishes to send a command to their device encrypts the command with a key shared only between the user and the device. The user sends message $m = (id, \text{cmd})$ to the integrator, where id is the device id and cmd is the encrypted command.
(2) The integrator, after collecting messages $m_1, \ldots, m_n$ from users participating in this round, encodes them into an OKVS S, where device ids are keys and encrypted commands are values. The integrator sends S to every vendor.
(3) Each vendor tries to decode a cmd from S for each of the vendor's devices using the device id id as key, and forwards the decoding result to the device.
(4) Each recipient device decrypts its command.

It is easy to see that this strawman approach satisfies the metadata privacy property, as long as the device ids are randomly chosen and cannot be traced back to their vendors. From the integrator's view, it cannot learn which user's message ends up in which vendor because the same OKVS is sent to every vendor. From the vendor's view, it cannot learn which devices actually have incoming commands due to the obliviousness property of OKVS, so it calls Decode on every of its device ids; hence, devices that have no incoming command will still receive a message from their vendor, but this message will appear random to the vendor and indistinguishable from the real messages. In addition, this approach is efficient on the vendors' side. Each vendor only needs to make n Decode calls, where n is the number of devices owned by that vendor. So its computational cost only scales with n and is not affected by other vendors' devices in the systems.

However, this strawman approach comes with a huge communication cost. Indeed, each vendor receives an OKVS that encodes *all*

commands, effectively multiplying the amount of traffic in the system by the number of vendors and making the bandwidth overhead unbearable. Small vendors that own dozens of devices would still receive an OKVS containing millions of commands in every round. Still, this OKVS-based strawman serves as an excellent starting point to build a privacy-preserving IoT system, and we show how to extend this approach to design Mohito, which retains the same security but with significantly improved efficiency.

4.2 Chosen Vendor as Shuffler

To make the protocol more practical, we must ensure that each vendor's cost scales only with the number of commands intended for this vendor, not the total number of commands. Hence, instead of sending the same OKVS to each vendor, the integrator should send to each vendor a distinct OKVS, encoding only the commands intended for that vendor. The challenge here is to efficiently construct these OKVSs, given that the integrator should not learn the target vendor of each user's message (as stated by the metadata privacy goal in Section 3.3).

Mohito handles this problem by introducing the notion of a *shuffler*. At the beginning of each round, the integrator chooses one vendor to play the shuffler. We discuss the practicality of the shuffler and the strategy the integrator can use to choose the shuffler in Section 8. The shuffler functions similarly to a node in a mix-net — it receives a list of user messages from the integrator, shuffles them, and sends the shuffled list back, so that the integrator no longer knows which user sent which message in the shuffled list.

To prevent the integrator from learning the permutation used for shuffling by connecting identical messages in the two lists, users encrypt each message with the shuffler's public key, and the shuffler decrypts them before shuffling. More precisely, in each round, the user first attaches the id of their command's destination vendor v to its message m and then performs a two-layer encryption to compute $m' = \mathsf{Enc}(\mathsf{pk}_{V^*}, \mathsf{Enc}(\mathsf{pk}_I, m))$, where $m = (id, v, \mathsf{cmd})$, and pk_{V^*} and pk_I are the public key of the shuffler and integrator respectively. The purpose of the inner encryption with pk_I is to prevent the shuffler from learning device ids in plaintext; otherwise, the shuffler, which is also one of the vendors, will learn which of its devices receive commands, violating our goal of metadata privacy.

At the end of the shuffling process, the integrator obtains the shuffled list of messages, where each message consists of a device id, a vendor id, and an encrypted command, so it can group the device id and commands based on the vendor ids and encode them into separate OKVSs.

By outsourcing the shuffling process to the shuffler, we break the linkage between the commands and the users. The integrator can now learn the target vendor of each command without compromising the metadata privacy, as it no longer knows which user sent which command. Therefore, we are able to construct smaller OKVSs, each encoding only the commands intended for a specific vendor, greatly reducing the bandwidth cost of the protocol.

4.3 Ephemeral Command ID

The downside of allowing the integrator to learn the target vendor of each command is that it also learns which vendor owns which device id. Combined with the fact that the integrator can infer the relationship between users and device ids over time (e.g. by observing which user always participates in the rounds where a particular id appears), the integrator can use the device ids as identifiers of users and therefore deduce the user to vendor mapping — a violation of the metadata privacy.

To overcome this problem, Mohito creates a unique one-time id for each new command. Instead of attaching the static device id to the command, the user generates a fresh id which appears random and is tied to the current round. We refer to this id as an *ephemeral id*. More precisely, we use the device id as a seed to generate a key k for some PRF F and compute the ephemeral id $z = F_k(r)$, where r is a unique identifier that represents the current round. By this scheme, the user and the vendor can compute the ephemeral id for each device in a given round, but the integrator cannot. Each ephemeral id is globally unique; messages from the same user to the same device will have different ephemeral ids in different rounds, so that ids no longer serve as a way to identify users.

Security. We now briefly discuss that the Mohito protocol we have shown so far achieves our goal of *metadata privacy* within a single round. Against a curious integrator, the security reduces to (1) the uniform shuffle, (2) the security of the PRF F, and (3) the security of the public-key encryption scheme. Together they ensure that guessing which vendor a particular benign user is sending commands to is equivalent to guessing which index in the shuffled message list this user's message lands, of which the adversary has no better strategy than random guessing. Against a curious vendor, the security reduces to (1) the obliviousness property of OKVS, and (2) the the security of the public-key encryption scheme. They ensure the vendor cannot tell whether a command it decodes is from a real user or not and therefore cannot learn which devices have actually received commands. We provide a more formal security analysis in Appendix C.

5 PREVENTING CROSS-ROUND ATTACKS

One major challenge of designing a metadata-hiding system is to prevent cross-round attacks. Specifically, in a communication system, if the adversary can observe the traffic patterns across multiple rounds, it can make statistical inferences about the relationship between message senders and receivers. This type of attack is often referred to as an intersection attack or statistical disclosure attack [28, 36, 42].

Although intersection attacks affect many anonymity systems, they become more difficult or even impractical to carry out as the size of the anonymity set increases. As a result, these anonymity systems often choose to employ large anonymity sets to make them less vulnerable [27]. We note that Mohito may appear to belong to one of these systems, as it is designed to support IoT systems where there is usually a huge amount of concurrent traffic in every round and therefore a large anonymity set; however, there are two drawbacks to this assumption: first, to have a large anonymity set, Mohito should only end a round once enough users have participated, leading to uneven round durations and latencies; second, there are certain common scenarios in the IoT settings that can break this assumption. Section 5.1 gives an example attack, illustrating how intersection attacks in Mohito may lead to privacy leakage in these scenarios; Section 5.2 outlines Mohito's defense.

5.1 Intersection Attacks in IoT Systems

Recall that in Mohito we consider two types of adversaries: a semi-honest integrator and a semi-honest vendor. In the first case, the adversary shares the view of the integrator and can learn two things[2]:

(1) the set of users who participate in each round, and
(2) the number of commands received by each vendor in each round.

By accumulating such information over multiple rounds, it can deduce which user is more likely to communicate with which vendor, violating our metadata privacy security goal.

We give one simple example to illustrate how the attack may work. Suppose that in round i, there are three users, U_A, U_B, and U_C, that send a message to the integrator, while vendors V_A, V_B, and V_C receive 1, 2, and 0 messages respectively; then in round j, the participating users become U_A and U_D, while vendors V_A, V_B, and V_C receive 1, 0, and 1 messages respectively. By intersecting the information from these two rounds, we can observe that only U_A is in both rounds and only V_A receives a message in both rounds. Hence, we can infer U_A is communicating with V_A (for simplicity, assuming each user only uses a single vendor). Conversely, we can also compute the difference of the information, and observe that U_D appears only in round j and causes V_C to receive an additional message. We note that this latter type of scenario is common in IoT systems, as it corresponds to the event when a new user joins the system, making the Mohito protocol that we have discussed so far vulnerable to this attack.

We note that the second type of adversary (where it shares the view of a vendor) does not benefit from intersection attacks. The Mohito protocol forces the vendor to send a message to every device it owns in each round but does not allow it to learn which of these messages represent real commands. Therefore, this adversary cannot learn the set of devices that actually participate in each round by eavesdropping on the vendor, so it cannot perform an intersection to narrow down the set of devices, similar to how the first type of adversary narrows down the set of users.

5.2 Defending against Intersection Attacks

To prevent intersection attacks, we must prevent the integrator from learning either 1) the set of participating users or 2) the number of commands each vendor receives. Hiding the first information is a well-studied problem in the literature, and the two common approaches are anonymous credentials [19, 20, 48] and client-side cover traffic [15, 18, 42, 54, 56]. We note that these approaches either have inherent downsides (e.g. anonymous credentials still leak IP addresses) or do not align with our system model (e.g. client-side cover traffic often requires users to be always online). Nonetheless, should the situation fit, they can be plugged directly on top of Mohito and we briefly discuss them in Section 8.

Therefore, we focus on hiding the second information, namely the number of commands each vendor receives. To achieve this, Mohito injects cover traffic from the shuffler.

Server-side cover traffic. At a high level, we instruct the shuffler to add fake commands such that, in each round, the integrator finds that the number of commands delivered to each vendor is always equal to some pre-determined number. Let, $\mathbf{C}$ be a vector of size $|\mathcal{V}|$, such that $\mathbf{C}_v$ denotes the number of messages expected by the v^{th} vendor. If v^{th} vendor actually receives $\mathbf{A}_v$ commands, then the shuffler should inject $\mathbf{B}_v = \mathbf{C}_v - \mathbf{A}_v$ fake commands for the v^{th} vendor. In this way, the integrator always receives $\mathbf{C}_v$ commands for the v^{th} vendor and never learns the number of "real" commands of that vendor.

Specifically, assume that the integrator and users agree on an ordering of vendors. Suppose a user wants to send a command to a device belonging to the v^{th} vendor. This user prepares a vector $\mathbf{e}_v$ of length $|\mathcal{V}|$, where $\mathbf{e}_v$ denotes a standard-basis vector (all-zeros vector with a one at the index v), and where $|\mathcal{V}|$ denotes the total number of vendors. The user then splits this vector into two additive shares $\mathbf{x_1}$ and $\mathbf{x_2}$ such that $\mathbf{x_1} + \mathbf{x_2} = \mathbf{e}_v$. The user encrypts the two shares in such a way that $\mathbf{x_1}$ will be delivered to the integrator while $\mathbf{x_2}$ will be delivered to the shuffler (using the two-layer encryption approach we discussed in Section 4.2).

Next, the integrator accumulates and sums up all shares it receives from users in this round. Let the accumulated share is $\mathbf{X_1}$. It then computes $\mathbf{Y} \leftarrow \mathbf{C} - \mathbf{X_1}$, The integrator sends $\mathbf{Y}$ to the shuffler, which computes

$$\mathbf{B} \leftarrow \mathbf{Y} - \mathbf{X_2}$$

where $\mathbf{X_2}$ is the shuffler's accumulated share[3]. It follows that, $\mathbf{B} = \mathbf{C} - \mathbf{X_1} - \mathbf{X_2} - = \mathbf{C} - \mathbf{A}$. Therefore, the shuffler can simply place $\mathbf{B}_v$ fake commands for the v^{th} vendor into the shuffled list of messages before returning them to the integrator. As discussed in Section 4, a real message is a ciphertext generated by encrypting another ciphertext (i.e., encrypted commands) with the integrator's pubic key, so we can simply generate an indistinguishable fake message by encrypting a random string.

In this way, the integrator now always sees that v^{th} vendor receives $\mathbf{C}_v$ commands. However, it does not know how many of these commands are from the users and how many of them are fake commands from the shuffler, namely, $\mathbf{A}_v$ and $\mathbf{B}_v$. As a result, the integrator can no longer perform an intersection attack, as it only knows the senders of the messages but learns nothing about the receivers. Additionally, this approach also eliminates the need for Mohito to wait for enough user participation before advancing to the next round.

Traffic bursts. One downside of setting a pre-determined $\mathbf{C}_i$ for each vendor is that there might be a burst of traffic in some rounds where $\mathbf{A}_j > \mathbf{C}_j$ for some vendor j. In these rounds, the shuffler will observe that $\mathbf{B}_j < 0$ for vendor j. To account for this, we do not add any fake commands for vendor j, but for every other vendor, we add $|\mathbf{B}_j|$ fake commands to each of them. In this way, the integrator does not know which vendor is responsible for the traffic burst, as it just sees every vendor uniformly receives $|\mathbf{B}_j|$ additional commands. We note that this does *not* impact the computational cost of each vendor. Recall that the cost of executing n decoding queries is $O(n\lambda)$, where λ is the security parameter, and does not depend on the size

[2]While the adversary also learns the ephemeral ids and encrypted commands, they appear random to the adversary.

[3]We do not want to reveal the number of actual commands received by each vendor ($\mathbf{A}$) to the shuffler. Vendors may be commercially competing with each other and therefore such information should not be leaked to a potential competitor.

of the OKVS. Therefore, since the number of decoding queries that each vendor needs to make remains the same, a traffic burst does not lead to performance overhead for the vendors.

Choosing $\mathbf{C}$. Recall that our threat model permits the integrator to collude with a small number of users and may generate up to δ commands in each round. To account for this, we must ensure each vendor v has a $\mathbf{C}_v$ that is greater than δ, such that no matter how many commands these colluding users generate, they cannot cause a traffic burst and manipulate the number of commands returned to the integrator. We discuss more details on finding a reasonable choice for $\mathbf{C}$ in Appendix B.

6 FULL Mohito PROTOCOL

In this section, we formalize the full Mohito protocol. Each device in Mohito must be first properly set up (Section 6.1) before it is ready to receive commands. Users send commands to their devices in synchronized rounds through the Mohito servers, consisting of one integrator and a number of vendors (Section 6.2). Devices respond to the commands with status update messages (Section 6.3).

For simplicity, we omit the user setup phase and assume each user has already registered an account with the IoT servers and obtained an access token for sending commands to the integrator through standard authorization protocols like OAuth.

Notation. We denote the integrator as I and the set of vendors, users, and devices as $\mathcal{V}, \mathcal{U}$, and $\mathcal{D}$ respectively. Each device $D \in \mathcal{D}$ has three attributes: (User, Vendor, ID), denoting the user that owns the device, the vendor that manufactures the device, and a string that globally identifies the device, respectively. We assume the ID of a device can be used to derive a globally unique key for some PRF F. For simplicity, when a message m is encrypted under someone's key, such as pk_A that is owned by party A, we simply denote the resulting ciphertext as $\langle m \rangle_A$. In addition, when calling Enc or Dec on a vector, say $\mathbf{m}$, we are encrypting or decrypting each individual element in the vector.

6.1 Device Setup Phase

In Mohito, a device D may come online after it has been successfully set up by a user U. This setup phase is mandatory in almost every modern smart home system. A common device setup phase incorporates two steps: first D establishes a private communication channel with U (usually through a local Wi-Fi hosted by D), and then U tells D how to connect to the Internet (usually by sharing their home Wi-Fi's SSID and password) so that D can directly communicate with its vendor V. In Mohito, we extend the setup phase by additionally instructing D to exchange the following information with U and V:

(1) Share the device's ID D.ID with both U and V. This ID will later be used to generate a PRF key.
(2) Generate a symmetric encryption key k_D and share it with U. We will use k_D to ensure end-to-end encryption between D and U, and we assume k_D is used with an authenticated encryption scheme.

We note that such private communication channels between device and user are often short-lived and require active human inputs to establish. Therefore, similar to today's smart home systems, Mohito only performs the device setup phase once at the beginning of each device's lifecycle.

6.2 Command Sending Phase

Once a user has successfully set up one or more devices, they may start sending commands to their devices through Mohito servers, which consist of an integrator and a number of vendors. We list the pseudocode of our protocol in Fig. 2.

6.2.1 Round initialization. At the beginning of each round, the integrator I chooses a unique round identifier t for this round and shares t with each user participating in this round as well as with each vendor. In addition, it selects a vendor V^* that is in charge of shuffling the commands for this round and shares V^* with the users participating in this round. We refer to V^* as the shuffler.

6.2.2 Users issue requests. When a user U wants to issue command cmd to device D, they invoke the function IssueCommand and send the output to the integrator I. This includes the following operations:

(1) Generate an ephemeral id z for this command using a PRF F that is keyed by the device's ID and takes as input the current round identifier t. We additionally append a command counter j to the input to F to allow U to send multiple commands in the same round. A command counter of j means that the current cmd is the j-th command that U sends to D in this round.
(2) Construct the message blob m as $(z, v, \langle \mathsf{cmd} \rangle_D)$, where v is the ID of D's vendor and $\langle \mathsf{cmd} \rangle_D$ is the ciphertext by encrypting the command c with the symmetric encryption key k_D (obtained during the device setup phase of D).
(3) Compute the additive shares $\mathbf{x_1}$ and $\mathbf{x_2}$ as described in Section 5.2.
(4) Encrypt with the public keys of the integrator I and the shuffler V^* to compute the final message m' as $(\langle \langle m \rangle_I, \mathbf{x_2} \rangle_{V^*}, \mathbf{x_1})$.

To prevent replay attacks, we require each command cmd to include a unique command identifier and the current timestamp as its attributes. In addition, we restrict that each user can send at most q commands to their devices in each round, so that $1 \le j \le q$. This restriction is reasonable as many real-world servers already enforce similar rate-limiting techniques in their APIs [3, 7].

6.2.3 Servers shuffle and encode requests. Once the integrator I has gathered all commands in a round, it processes them with the help of the shuffler V^*. Assume that, in a given round, I receives k messages $\mathbf{m'} = (m'_1, \ldots, m'_k)$ (ranked chronologically) from users that participated in this round. Integrator invokes I.EncodeCommands to compute the OKVS for each vendor using the following steps:

(1) I collects its shares $\mathbf{x_1}_1, \ldots, \mathbf{x_1}_k$ from $\mathbf{m'}$ to compute $\mathbf{Y}$, and then sends the remaining part of user messages to V^*.
(2) V^* adds the fake messages based on the scheme described in Section 5.2 and returns the messages to I after shuffling. It should also store the shuffle permutation π as well as the indices in the message list that correspond to fake messages for later use.
(3) I groups the resulting messages by the vendor id v attached in each message, encodes each group into an OKVS S using

U.IssueCommand(cmd, D, j, t, V^*)

$\mathsf{k}_D \leftarrow \mathsf{KGen}(D.\mathsf{ID})$
$z \leftarrow F_{\mathsf{k}_D}(j \| t)$
$v \leftarrow D.\mathsf{Vendor.ID}$
$\mathbf{x_1}, \mathbf{x_2} \leftarrow\!\!\$\ [\![\mathbf{e_v}]\!]$
$m \leftarrow (z, v, \langle c \rangle_D)$
$m' \leftarrow (\langle\langle m \rangle_I, \mathbf{x_2}\rangle_{V^*}, \mathbf{x_1})$
send m' **to** I

I.EncodeCommands $\left(\mathbf{m'}, \mathbf{C}^1, \mathbf{C}^2, \ldots, \mathbf{C}^{|\mathcal{V}|}\right)$

$\mathbf{m''}, (\mathbf{x}_{11}, \ldots, \mathbf{x}_{1k}) \leftarrow \mathbf{m'}$
$\mathbf{X_1} \leftarrow \sum_i \mathbf{x}_{1i}$
$\mathbf{Y} \leftarrow (\mathbf{C}^1, \ldots, \mathbf{C}^{|\mathcal{V}|}) - \mathbf{X_1}$
$\mathbf{m} \leftarrow\!\!\$\ V^*.\mathsf{ShuffleCommands}\left(\mathbf{m''}, \mathbf{Y}\right)$
$\mathbf{z}, \mathbf{v}, \langle \mathsf{cmd} \rangle_{\mathbf{D}} \leftarrow \mathsf{Dec}(\mathbf{m})$
for $V \in \mathcal{V}$ **do**
  $kv \leftarrow \{(\mathbf{z}_i, \langle \mathsf{cmd} \rangle_{\mathbf{D}i}) \mid \mathbf{v}_i = V.\mathsf{ID}\}$
  $S \leftarrow \mathsf{Encode}(kv)$
  send S **to** V

V^*.ShuffleCommands $(\mathbf{m''}, \mathbf{Y})$

$\mathbf{m}, (\mathbf{x}_{21}, \ldots, \mathbf{x}_{2k}) \leftarrow \mathsf{Dec}\left(\mathbf{m''}\right)$
$\mathbf{X_2} \leftarrow \sum_{i=1}^{k} \mathbf{x}_{2i}$
$\mathbf{B} \leftarrow \mathbf{Y} - \mathbf{X_2}$
for $V \in \mathcal{V}$ **do**
  $i \leftarrow V.\mathsf{ID}$
  if $\mathbf{B}_i < 0$ **then**
    for $V' \in \mathcal{V} \setminus \{V\}$ **do**
      add $|\mathbf{B}_i|$ fake messages for V' to $\mathbf{m}$
  else
    add $\mathbf{B}_i$ fake messages for V to $\mathbf{m}$
$\langle \mathbf{z} \rangle_{\mathbf{I}}, \langle \mathbf{v} \rangle_{\mathbf{I}}, \langle \langle \mathsf{cmd} \rangle_{\mathbf{D}} \rangle_{\mathbf{I}} \leftarrow \mathbf{m}$
$\pi \leftarrow\!\!\$\ \Pi$
send $\pi\left(\langle \mathbf{z} \rangle_{\mathbf{I}}\right), \pi\left(\langle \mathbf{v} \rangle_{\mathbf{I}}\right), \pi\left(\langle\langle \mathsf{cmd} \rangle_{\mathbf{D}} \rangle_{\mathbf{I}}\right)$ **to** I

V.DecodeCommands $(\mathcal{D}_a, t, S)$

for $D \in \mathcal{D}_a$ **do**
  $\mathsf{k}_D \leftarrow \mathsf{KGen}(D.\mathsf{ID})$
  for $j = 1 \ldots q$ **do**
    $z \leftarrow F_{\mathsf{k}_D}(j \| t)$
    $m_j \leftarrow \mathsf{Decode}(S, z)$
  send $m_1 \| \ldots \| m_q$ **to** D

Figure 2: Mohito protocol for command sending phase. Each procedure is executed by different entities: user (U), integrator (I), shuffler vendor (V^*), and device vendor (V).

the ephemeral id z as key and the encrypted commands $\langle \mathsf{cmd} \rangle_D$ as value, and sends S to the corresponding vendor.

6.2.4 Vendors decode commands. Finally, each vendor V invokes the function DecodeCommands, which takes as input the set of currently active devices $\mathcal{D}_a$ (i.e. devices that have an ongoing connection with the vendor), the current round's identifier t, and the OKVS S received from I, and computes the message to be delivered to each device $D \in \mathcal{D}_a$.

Since V does not know which devices actually receive a command from their user, it will compute all possible ephemeral ids for each active device $D \in \mathcal{D}_a$ in this round, try to decode each of these ephemeral ids from S, and send the decoded values to D. In this way, Mohito forces each active device to communicate with their vendor in each round. As existing smart home devices in the wild are already constantly chatting with their vendors even when they are idle [18, 59], we believe that this extra communication is still practical in real-world settings.

Finally, each active device $D \in \mathcal{D}_a$ receives a message from V and tries to decrypt the message using its own symmetric encryption key k_D. Each device that actually receives a command from its user successfully recovers the command; each device that does not observe a decryption error.

6.3 Device Response Phase

After a device D receives a message from its vendor V, it must reply with a status update r, which represents the result or the new device status after executing the user command. Each active device in $\mathcal{D}_a$ sends its own r to V regardless of whether it successfully decrypted a command; otherwise V will identify which devices actually received commands by observing which devices reply. In the case that D does not actually receive a command, r can be a random string.

At a high level, we can view the protocol in this phase as the reverse of the protocol in command sending phase. That is, the roles of encoder and decoder are switched and the shuffler now reversely shuffles the integrator's message list. We list the pseudocode of this phase in Fig. 3.

6.3.1 Devices issue responses. The device D performs an onion encryption on its response r by using first D's own symmetric encryption key k_D, then the public key of the shuffler V^*, and finally the public key of the integrator I. The ephemeral id associated with the original command is also recomputed and attached to the encrypted response.

The outer encryption layer with pk_I is necessary, because otherwise the vendor that acts as the shuffler in this round would learn which of the devices actually receive commands by checking

D.IssueResponse(r, j, t, V^*)

$\mathsf{k}_D \leftarrow \mathsf{KGen}(D.\mathsf{ID})$
$z \leftarrow F_{\mathsf{k}_D}(j \| t)$
send $(z, \langle\langle\langle r\rangle_D\rangle_{V^*}\rangle_I)$ **to** D.Vendor

V^*.ShuffleResponses $(\mathbf{r}')$

/ π and $\mathbf{m}$ is the internal variables from V^*.ShuffleCommands
$\mathbf{r} \leftarrow \pi^{-1}(\mathsf{Dec}(\mathbf{r}'))$
for $r_i \in \mathbf{r}$ **do**
  if i-th element in $\mathbf{m}$ is a fake message **then**
    remove r_i from $\mathbf{r}$
send $\mathbf{r}$ **to** I

V.EncodeResponses $(\mathbf{r}'')$

send Encode $(\mathbf{r}'')$ **to** I

I.DecodeResponses (S, V)

/ $\mathbf{z}$ and $\mathbf{v}$ are the internal variables from I.EncodeCommands
for $i = 1 \ldots \|\mathbf{z}\|$ **do**
  if $V.\mathsf{ID} = \mathbf{v}_i$ **do**
    $r''_i \leftarrow \mathsf{Decode}(S, \mathbf{z}_i)$
  else
    $r''_i \leftarrow \perp$
$\mathbf{r}'' \leftarrow (r''_1, \ldots, r''_{\|\mathbf{z}\|})$
$\mathbf{r}' \leftarrow \mathsf{Dec}(\mathbf{r}'')$
$\mathbf{r} \leftarrow\!\!\!\$\ V^*.\mathsf{ShuffleResponses}(\mathbf{r}')$
return $\mathbf{r}$

Figure 3: Mohito protocol for device response phase.

whether the device's response is in the response list it received from I during the steps in Section 6.3.3.

We note that sometimes the devices may not be able to generate their responses in the same round that they receive commands. We discuss in Appendix A how to extend Mohito's protocol to support asynchronous responses.

6.3.2 Vendors encode responses. The vendor V, after receiving responses from each of its active devices, encodes them an OKVS S and sends S to I. Note that we cannot send responses directly to I without the OKVS encoding; otherwise I would learn which commands it receives from V^*.ShuffleCommands are fake by comparing the ephemeral ids attached to the responses with the ones attached to the commands.

6.3.3 Servers decode and reverse-shuffle responses. Once the integrator I receives an OKVS S from a vendor V, it iterates through the list of ephemeral ids $\mathbf{z}$ that it used to encode the OKVS for V during I.EncodeCommands (Section 6.2.3) and tries to decode values from S. Next, I sends the list of decoded values to the shuffler V^*, which then shuffles the list using the inverse of the order that V^* used during V^*.ShuffleCommands. In this way, each entry in the final resulting list $\mathbf{r}'$ represents the response to the command that has the same index in the list $\mathbf{m}'$ that I received at the beginning of I.EncodeCommands. That is, $\mathbf{r}'_i$ is the response to $\mathbf{m}'_i$. Next, I can identify the user U who initially sent $\mathbf{m}'_i$ (for example, U may still have the TCP connection with I open and waiting for a response) and send the response back.

7 IMPLEMENTATION AND EVALUATION

We build a proof-of-concept implementation of Mohito. We benchmark our implementation to discuss how to appropriately set round duration and show that, when compared to prior general-purpose metadata-hiding system not tailored for IoT setting, we can achieve 600× more throughput. Our source code is available at https://github.com/earlence-security/mohito.

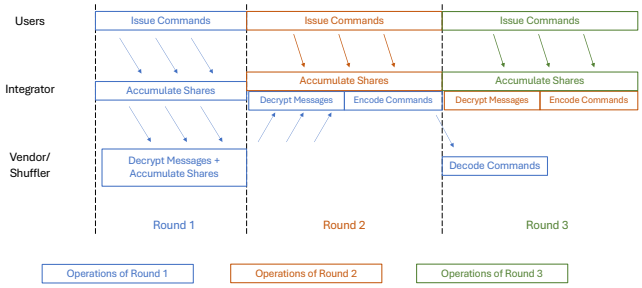

Figure 4: Execution graph of our Mohito implementation. Message streaming is used to reduce system idle time.

7.1 Implementation

We prototype the Mohito protocol in Go and choose PaXoS [45] as the underlying implementation for OKVS. In addition, we use HMAC-SHA256 for PRF and Curve25519, XSalsa20 and Poly1305 for authenticated encryption. We use grpc to handle communication between IoT servers and apply its message streaming functionality when possible. For example, at the beginning of each round, the integrator forwards each user message to the shuffler as soon as it comes in; hence, the shuffler can start its decryption process immediately, instead of waiting for the integrator to collect all user messages in this round. The execution graph of our system during the command sending phase is shown in Fig. 4. This strategy allows the servers to execute the protocol concurrently and greatly reduces their idle time.

In addition, the shuffler does not shuffle messages in memory, as this would be costly. Instead, it computes the permutation only and uses it to determine the order in which the messages are streamed back to the integrator. However, we do ensure the shuffler has received all messages in the current round before starting to send

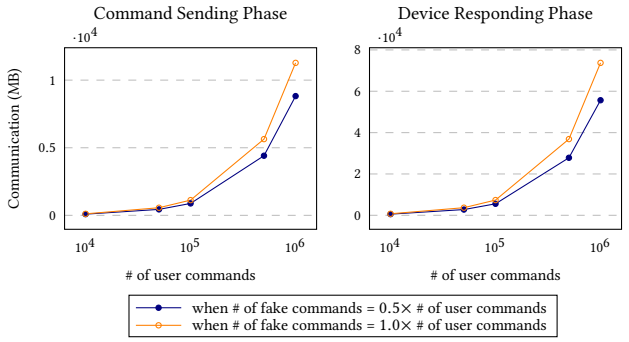

Figure 5: The communication cost of Mohito, given a command size of 1 KB. The device responding phase costs more bandwidth than the command sending phase, since the size of OKVSs in the former scales with the number of active devices, while the size in the latter scales with the number of commands.

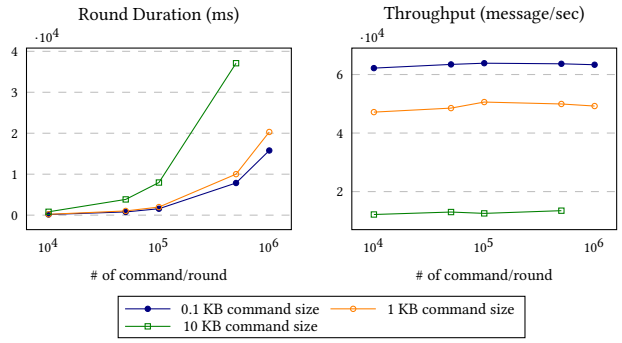

Figure 6: Performance of Mohito. Based on the duration of each round (left), we can compute the system throughput (right), which remains unaffected by the number of commands we put into each round.

them back; otherwise, the integrator would learn that some messages rank lower than others in the permutation. We also pregenerate enough fake messages for the shuffler (as these fake messages only require the public keys of the integrator as input) so that the shuffler does not need to compute them on the fly.

7.2 Experiments and Evaluation

For all experiments, we deployed Mohito on three AWS c5d.2xlarge servers, two of which are running the integrator and the vendor/shuffler while the remaining one simulates the users and the devices collectively. Each server is configured with 8 vCPUs and 16 GB of memory. They are connected with 10 Gbps network.

While we primarily focus on evaluating the IoT servers in this section, we note that the protocol we run on each end user and each device is relatively simple: 1) it takes only 0.03 seconds on our machine and should not be a bottleneck for modern embedded microprocessors; 2) it executes the same set of operations in every round, so it can be baked directly into the device firmware (similarly to SSL/TLS libraries).

7.2.1 Communication Cost. For the command sending phase, there are three parameters that determine the communication cost between the integrator and vendors in each round[4]: the number of commands sent by users, the size of the command, and the number of fake commands injected by the shuffler. We only focus on how the communication cost varies as the number of real and fake commands increases, because 1) it reflects the impact of adding more users/devices on the system, 2) cost scales linearly with command size. Fig. 5 (left) shows the results when the command size is set to 1 KB.

For the device response phase, the communication cost also depends on the number of active devices, as Mohito requires that each device generate a response. In practice, there will be more active devices in each round than the number of commands, as in each round not all devices will receive a command. Fig. 5 (right) shows how communication changes when we assume that in each round only 10% of active devices receive commands.

7.2.2 Performance. This section focuses on the performance of IoT servers, representing the overhead that Mohito protocol added to a standard non-metadata-hiding IoT system, where each server simply forwards the message and no computations are required. Our servers are deployed in the same data center to minimize communication latency, allowing us to focus on the performance impact of the Mohito protocol.

Round duration. As shown in Fig. 4, the operations of different rounds overlap due to message streaming. The integrator begins the second round as soon as it finishes forwarding all user messages from the first round to the shuffler. As a result, for all rounds except for the first, the integrator needs to process the shuffler's return messages from the previous round while it is forwarding the user messages from the current round. This also means that the integrator can only proceed to the next round after it has completed processing the shuffler's return messages from the previous round, which consists of two operations: the decryption of the shuffler's return messages and the OKVS encoding. Therefore, the execution time of these two operations determines the round duration.

In Fig. 6, we show how the round duration varies due to the number of commands returned by the shuffler (including both real and fake commands) and the size of command. While a shorter round leads to smaller latencies, we would also want to keep the number of commands the system can handle large enough so that the small number of devices controlled by the adversary cannot cause a traffic burst, as we discussed in Section 5.2. For example, if we assume the adversary can send up to $\delta = 100$ commands in each round and there are 1,000 vendors in the system, we need to ensure each round can handle at least 100,000 commands, which, given a command size of 1 KB, translates to 1.9 seconds per round.

We note that the performance of the device response phase is almost identical to the performance of the command sending phase.

[4]We note that while the number of vendors in the systems also impacts the communication cost, it only controls the number of additive shares attached to the command; therefore, adding a new vendor is equivalent to increasing the command size by two integers. In addition, the number of users or devices

While each vendor may need to encode more responses in the device response phase compared to the number of commands they decode during the command sending phase, the bottleneck of the system only depends on the integrator's operations as we have shown above. Therefore, the only difference between these two phases is that, instead of encoding commands into an OKVS, the integrator now decodes responses from OKVS. However, the encoding/decoding cost is relatively small compared to the decryption cost. For example, given 100,000 commands and a command size of 1 KB, the time to decrypt is 1.3 seconds, while it takes 0.6 seconds to encode and 0.2 seconds to decode. As such, for the remaining evaluations, we only focus on the command sending phase.

Throughput. Based on Fig. 6, the throughput of the system primarily depends on the command size. Given a command size of 1 KB, Mohito is capable of handling approximately 48,000 commands per second, no matter how many commands we put into each round. Even if we assume half of these commands are fake, the effective throughput of the systems is 24,000 commands per second. For comparison, Express [32], a state-of-the-art general-purpose metadata-hiding messaging system that uses a two-server PIR technique, can fit into the IoT server structure and provides similar security guarantees, but it only supports around 40 messages per second while running on machines with twice the number of CPU cores as ours, due to the need to support other functionalities such as message persistence that are not required in the IoT setting.

Latency. We define end-to-end latency as the time needed to deliver a command from a user to its device. Hence, if we ignore the communication latency between different parties, the end-to-end latency consists of three parts: the remaining duration of the current round, the entire duration of the next round, and the time it takes for the vendor to decode commands. Since each vendor only needs to decode its own OKVS, the decoding time is much shorter than the duration of each round and therefore the latency is primarily determined by our choice of the round duration. If we assume the uniform distribution of the command arriving time, then the average latency is approximately equal to 1.5 times of the round duration. That is, with 100,000 commands per round and a command size of 1 KB, the end-to-end latency is 2.7 seconds. We again use Express [32] as a comparison. Express provides roughly half of Mohito's latency for a single message, but its latency scales with the number of concurrent messages, while Mohito ensures all messages have similar latency.

System Cost. We measure the costs of running the Mohito servers (the vendors and the integrator) on AWS. Our current setup supports a throughput of 24,000 commands per second and costs 9.2 USD per day for the shuffler. However, since each vendor takes turns to become the shuffler, the cost of running the shuffler is split among all vendors. In addition, Mohito's performance can be scaled up by adding more machines. As shown by our evaluation of the round duration, the system bottleneck is the integrator's operation to decrypt a large list of messages, a highly parallelizable task. If the integrator has access to more servers, we can run Mohito's integrator protocol in a distributed manner: 1) each server decrypts a portion of the messages, 2) the servers send the decrypted messages to each other in a way such that the messages with the same vendor id end up in the same server, 3) each server computes the OKVSs locally and send them to the corresponding vendor. In this way, Mohito's computational cost is split evenly across all servers, effectively scaling the performance by the number of available servers. Therefore, if higher throughput is desired, it can be achieved by adding more machines.

8 DISCUSSION

Practicality and Adoptability. To ensure the practicality of our system and its adoption by existing IoT services, we have designed it to align with the integrator-vendor communication structures and batch API designs of current platforms. The only addition in our protocol is the introduction of the shuffler. Each vendor must agree to periodically act as the shuffler before it can engage in Mohito's privacy-preserving protocol. While this may seem to contradict our goal of not overburdening smaller vendors with the workload of larger ones, the integrator can select the shuffler in a way that ensures the average workload of each vendor is balanced. That is, the total number of messages a vendor must shuffle should in the long run be roughly proportional to the number of devices it operates. Therefore, smaller vendors should not be discouraged from participating in Mohito. For example, assuming the device distribution follows the IFTTT data in [44], smaller vendors tend to own around 50 active devices and hence they only need to become a shuffler once in every 22,500 rounds; as such, they may just run the shuffler server (as a lambda function) for a few seconds each day, so the extra overhead is insignificant. The compatibility and small overhead of Mohito minimize the need for substantial modifications on the part of vendors. Additionally, the adoption of a metadata-hiding privacy-preserving protocol is highly incentivized for vendors due to increasing legal requirements [9, 10] and growing user concerns regarding IoT data privacy [25, 60].

Malicious Security. While we model the IoT servers as honest-but-curious parties, Mohito's protocol can be upgraded to the malicious setting using techniques like secret-shared non-interactive proofs (SNIP) [26]. First, Mohito already defends against malicious servers if we exclude the defense against intersection attack, as we show in Appendix C. Second, for our defense against intersection attack, a malicious integrator can lie about its aggregated sum of secret shares to learn how many real commands a vendor receives. In this case, the integrator is aggregating private values from a number of users and therefore, if we allow inter-vendor communication (either directly or indirectly by using integrator as a proxy), then we can use a subset of vendors to run SNIP as a blackbox to prevent the integrator from tampering the secret shares, as long as one of the vendors remains honest.

Preventing intersection attacks via hiding user credentials. As we note in Section 5.2, another way to prevent intersection attacks is to remove the identity of the users via an anonymous credential system. Anonymous credentials allow each user to authorize and communicate with a server without revealing the user's identity, preventing the adversary from deducing a mapping between users and vendors. There are many works on anonymous credentials [19, 20, 48] and they can be plugged into Mohito directly, as Mohito does not place additional requirements on how users should authorize

with IoT servers. However, even with anonymous credentials, each time a user communicates with a server, the user's IP address will unavoidably leak. Thus, the adversary can still recover a mapping between IP addresses and vendors. This leakage is weaker, since in practice many IP addresses are dynamic [55], so the IP address alone may not always identify a user. Alternatively, OHTTP [8] can also be used to remove user identity; however, it would require adding an additional third party to our system model to serve as a relay between users and the integrator. The purpose of the relay is similar to Mohito's shuffler, as both ensure user messages cannot be attributed back to the users.

Preventing intersection attacks via client-side cover traffic. Users can also hide their traffic patterns by generating cover traffic from their side. In this way, a message from the user to the integrator may represent either a real command or a fake one, preventing the integrator from learning the set of users that are participating in each round. However, user-side cover traffic may not be practical in the IoT setting, because it would require the user to be always online (or at least online at specific times), but the user's client, usually a smartphone app, should not be expected to constantly run in the background and may disconnect arbitrarily. Nonetheless, approaches to client-side traffic for IoT devices have been proposed [15, 18, 56] and, if a user can satisfy these conditions, it may use client-side cover traffic alongside Mohito to prevent the integrator from learning when the user is participating.

9 RELATED WORKS

Mohito applies an anonymous communication system to the problem of IoT metadata protection. Thus, we review related works in these fields separately.

Metadata in IoT. IoT metadata and its associated privacy risks have been extensively studied by prior works. There is a line of works that analyzes how passive network observers , such as Internet service providers and WiFi eavesdroppers, can infer device activities based on encrypted IoT traffic [12, 17, 18, 31, 43, 50]. These rely on metadata information, including traffic rates and the domains of servers that IoT devices contact. While the adversaries (IoT vendors and integrator) in our threat model are different than theirs, both types of adversaries have access to such metadata.

Several defenses against passive network observers have been proposed. Most of them build solutions from the client/device side and utilize a technique called traffic shaping [15, 16, 18, 29, 57]. Traffic shaping modifies traffic generated by IoT devices by padding messages and injecting cover traffic. In Mohito, we assume a simple padding strategy by setting all messages to the same size; in practice, we can incorporate more efficient padding algorithms in these works to reduce the bandwidth overhead. Also, if a user can ensure its client is always online, their client-side cover traffic techniques can be used to complement Mohito's server-side cover traffic. EPIC [41] proposes a different type of solution by designing a differentially-private routing protocol at the network layer. To our knowledge, Mohito is the first system that hides IoT metadata from the server side.

Privacy-preserving IoT integrators. Recently, a number of works proposed privacy-preserving trigger-action platforms [13, 21, 22, 24, 33, 56, 58]. Trigger-action platforms are a special type of IoT integrator that allows device automation. These works focus on data privacy, but they have the additional benefit of supporting computation on IoT data. For example, eTAP [22] uses secure multi-party computation, and PatrIoT [58] uses hardware-based trusted execution environments. Combining metadata protection mechanisms with secure computation remains a challenging problem.

Anonymous communication systems. Mohito belongs to the class of communication systems that achieves cryptographic guarantees regarding anonymity and metadata-hiding properties. Many of these systems are based on mix-nets, which perform message shuffling in a peer-to-peer system. Examples of such systems include Dissent [53], Atom [37], and XRD [38]. They suffer from high latency due to the lack of a centralized party and therefore need to run multiple shuffles in each round. In contrast, the communication structure in IoT systems allows Mohito to perform a single shuffle, greatly reducing latency. Riposte [27], Pung [14], Talek [23], and Express [32] instead achieve anonymous communication by reading/writing user messages from/to a private database via private information retrieval. These approaches store the messages in the database persistently, thereby allowing reading and writing to happen in different rounds; however, they prioritize the performance of a single message and do not scale well when high throughput is required.

There is another class of communication systems that provides differential privacy guarantees [39, 40, 51, 52]. These systems, while generally having better performance, allow quantifiable leakage of metadata. Therefore, an attacker may eventually learn who is communicating after observing a large number of rounds in a differentially private system, whereas the security of Mohito and other cryptographic-based systems does not degrade over time.

10 CONCLUSION

We have presented Mohito, a privacy-preserving IoT system with metadata protection. It prevents both the integrator service and device vendors from learning which user communicates with which device. We tailor the protocol of Mohito to support the efficient handling of large concurrent traffic and achieve load-balancing among vendors with different computational resources, which are two key requirements for IoT systems. We evaluate the performance of Mohito and demonstrate that our implementation, although doubling the single-message latency, increases the throughput by 600× when compared to general-purpose metadata-hiding systems that provide similar security guarantees.

ACKNOWLEDGMENTS

We thank the anonymous reviewers and our shepherd for valuable feedback. This work is partly supported by NSF grants CNS-2312119 and CNS-2246353 and by gifts from Amazon and Google.

REFERENCES

[1] 2018. Technology preview: Sealed sender for Signal. https://www.signal.org/blog/sealed-sender/.
[2] 2024. Alexa Developer Documentation. https://developer.amazon.com/en-US/docs/alexa/documentation-home.html.
[3] 2024. AWS IoT Core endpoints and quotas. https://docs.aws.amazon.com/general/latest/gr/iot-core.html.

[4] 2024. Careful Connections: Keeping the Internet of Things Secure. https://www.ftc.gov/business-guidance/resources/careful-connections-keeping-internet-things-secure.

[5] 2024. Google Home Developers. https://developers.home.google.com/docs.

[6] 2024. IFTTT. https://ifttt.com.

[7] 2024. IFTTT Service Rate Limits. https://help.ifttt.com/hc/en-us/articles/1260803229749-IFTTT-Service-Rate-Limits.

[8] 2024. Oblivious HTTP. https://www.rfc-editor.org/rfc/rfc9458.

[9] 2024. Privacy guidance for manufacturers of Internet of Things devices. https://www.priv.gc.ca/en/privacy-topics/technology/gd_iot_man.

[10] 2024. Protecting Consumer Privacy in the Digital Age: Reaffirming the Role of Consumer Control. https://www.ftc.gov/system/files/documents/public_statements/980623/ramirez_-_protecting_consumer_privacy_in_digital_age_aspen_8-22-16.pdf.

[11] 2024. Service API Requirements - IFTTT. https://ifttt.com/docs/api_reference.

[12] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. 2020. Peek-a-boo: I see your smart home activities, even encrypted!. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. 207–218.

[13] Mohammad M Ahmadpanah, Daniel Hedin, and Andrei Sabelfeld. 2023. LazyTAP: On-Demand Data Minimization for Trigger-Action Applications. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 3079–3097.

[14] Sebastian Angel and Srinath Setty. 2016. Unobservable communication over fully untrusted infrastructure. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 551–569.

[15] Noah Apthorpe, Danny Yuxing Huang, Dillon Reisman, Arvind Narayanan, and Nick Feamster. 2018. Keeping the smart home private with smart (er) iot traffic shaping. *arXiv preprint arXiv:1812.00955* (2018).

[16] Noah Apthorpe, Dillon Reisman, and Nick Feamster. 2017. Closing the blinds: Four strategies for protecting smart home privacy from network observers. *arXiv preprint arXiv:1705.06809* (2017).

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

[18] Noah Apthorpe, Dillon Reisman, Srikanth Sundaresan, Arvind Narayanan, and Nick Feamster. 2017. Spying on the smart home: Privacy attacks and defenses on encrypted iot traffic. *arXiv preprint arXiv:1708.05044* (2017).

[19] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. 2009. Randomizable proofs and delegatable anonymous credentials. In *Advances in Cryptology-CRYPTO 2009: 29th Annual International Cryptology Conference, Santa Barbara, CA, USA, August 16-20, 2009. Proceedings*. Springer, 108–125.

[20] Jan Camenisch and Els Van Herreweghen. 2002. Design and implementation of the idemix anonymous credential system. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*. 21–30.

[21] Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2022. Practical Data Access Minimization in {Trigger-Action} Platforms. In *31st USENIX Security Symposium (USENIX Security 22)*. 2929–2945.

[22] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. 2021. Data privacy in trigger-action systems. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 501–518.

[23] Raymond Cheng, William Scott, Elisaweta Masserova, Irene Zhang, Vipul Goyal, Thomas Anderson, Arvind Krishnamurthy, and Bryan Parno. 2020. Talek: Private group messaging with hidden access patterns. In *Annual Computer Security Applications Conference*. 84–99.

[24] Yu-Hsi Chiang, Hsu-Chun Hsiao, Chia-Mu Yu, and Tiffany Hyun-Jin Kim. 2020. On the Privacy Risks of Compromised Trigger-Action Platforms. In *Computer Security – ESORICS 2020*, Liqun Chen, Ninghui Li, Kaitai Liang, and Steve Schneider (Eds.).

[25] Eun Kyoung Choe, Sunny Consolvo, Jaeyeon Jung, Beverly Harrison, and Julie A Kientz. 2011. Living in a glass house: a survey of private moments in the home. In *Proceedings of the 13th international conference on Ubiquitous computing*. 41–44.

[26] Henry Corrigan-Gibbs and Dan Boneh. 2017. Prio: Private, robust, and scalable computation of aggregate statistics. In *14th USENIX symposium on networked systems design and implementation (NSDI 17)*. 259–282.

[27] Henry Corrigan-Gibbs, Dan Boneh, and David Mazières. 2015. Riposte: An anonymous messaging system handling millions of users. In *2015 IEEE Symposium on Security and Privacy*. IEEE, 321–338.

[28] George Danezis and Andrei Serjantov. 2005. Statistical disclosure or intersection attacks on anonymity systems. In *Information Hiding: 6th International Workshop, IH 2004, Toronto, Canada, May 23-25, 2004, Revised Selected Papers 6*. Springer, 293–308.

[29] Trisha Datta, Noah Apthorpe, and Nick Feamster. 2018. A developer-friendly library for smart home iot privacy-preserving traffic obfuscation. In *Proceedings of the 2018 workshop on IoT security and privacy*. 43–48.

[30] Emma Dauterman, Eric Feng, Ellen Luo, Raluca Ada Popa, and Ion Stoica. 2020. {DORY}: An encrypted search system with distributed trust. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 1101–1119.

[31] Shuaike Dong, Zhou Li, Di Tang, Jiongyi Chen, Menghan Sun, and Kehuan Zhang. 2020. Your smart home can't keep a secret: Towards automated fingerprinting of iot traffic. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security*. 47–59.

[32] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, Dan Boneh, et al. 2021. Express: Lowering the Cost of Metadata-hiding Communication with Cryptographic Privacy.. In *USENIX Security Symposium*. 1775–1792.

[33] Earlence Fernandes, Amir Rahmati, Jaeyeon Jung, and Atul Prakash. 2018. Decentralized action integrity for trigger-action IoT platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*.

[34] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2021. Oblivious key-value stores and amplification for private set intersection. In *Advances in Cryptology–CRYPTO 2021: 41st Annual International Cryptology Conference, CRYPTO 2021, Virtual Event, August 16–20, 2021, Proceedings, Part II 41*. Springer, 395–425.

[35] Niv Gilboa and Yuval Ishai. 2014. Distributed point functions and their applications. In *Advances in Cryptology–EUROCRYPT 2014: 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques, Copenhagen, Denmark, May 11-15, 2014. Proceedings 33*. Springer, 640–658.

[36] Dogan Kedogan, Dakshi Agrawal, and Stefan Penz. 2003. Limits of anonymity in open environments. In *Information Hiding: 5th International Workshop, IH 2002 Noordwijkerhout, The Netherlands, October 7-9, 2002 Revised Papers 5*. Springer, 53–69.

[37] Albert Kwon, Henry Corrigan-Gibbs, Srinivas Devadas, and Bryan Ford. 2017. Atom: Horizontally scaling strong anonymity. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 406–422.

[38] Albert Kwon, David Lu, and Srinivas Devadas. 2020. {XRD}: Scalable messaging system with cryptographic privacy. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 759–776.

[39] David Lazar, Yossi Gilad, and Nickolai Zeldovich. 2018. Karaoke: Distributed private messaging immune to passive traffic analysis. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 711–725.

[40] David Lazar and Nickolai Zeldovich. 2016. Alpenhorn: Bootstrapping secure communication without leaking metadata. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. 571–586.

[41] Jianqing Liu, Chi Zhang, and Yuguang Fang. 2018. Epic: A differential privacy framework to defend smart homes against internet traffic analysis. *IEEE Internet of Things Journal* 5, 2 (2018), 1206–1217.

[42] Nick Mathewson and Roger Dingledine. 2004. Practical traffic analysis: Extending and resisting statistical disclosure. In *Privacy Enhancing Technologies*, Vol. 3424. Springer, 17–34.

[43] M Hammad Mazhar and Zubair Shafiq. 2020. Characterizing smart home iot traffic in the wild. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)*. IEEE, 203–215.

[44] Xianghang Mi, Feng Qian, Ying Zhang, and XiaoFeng Wang. 2017. An empirical characterization of IFTTT: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*. 398–404.

[45] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 2020. PSI from PaXoS: fast, malicious private set intersection. In *Advances in Cryptology–EUROCRYPT 2020: 39th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, May 10–14, 2020, Proceedings, Part II*. Springer, 739–767.

[46] Nidhi Rastogi and James Hendler. 2017. WhatsApp security and role of metadata in preserving privacy. In *12th International Conference on Cyber Warfare and Security*, Vol. 6817. 269–275.

[47] B. Schneier. 2014. Metadata = Surveillance. *IEEE Security & Privacy* 12, 02 (2014), 84–84. https://doi.org/10.1109/MSP.2014.28

[48] Siamak F Shahandashti and Reihaneh Safavi-Naini. 2009. Threshold attribute-based signatures and their application to anonymous credential systems. In *Progress in Cryptology–AFRICACRYPT 2009: Second International Conference on Cryptology in Africa, Gammarth, Tunisia, June 21-25, 2009. Proceedings 2*. Springer, 198–216.

[49] Vijay Srinivasan, John Stankovic, and Kamin Whitehouse. 2008. Protecting your daily in-home activity information from a wireless snooping attack. In *Proceedings of the 10th international conference on Ubiquitous computing*. 202–211.

[50] Rahmadi Trimananda, Janus Varmarken, Athina Markopoulou, and Brian Demsky. 2020. Packet-level signatures for smart home devices. In *Network and Distributed Systems Security (NDSS) Symposium*, Vol. 2020.

[51] Nirvan Tyagi, Yossi Gilad, Derek Leung, Matei Zaharia, and Nickolai Zeldovich. 2017. Stadium: A distributed metadata-private messaging system. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 423–440.

[52] Jelle Van Den Hooff, David Lazar, Matei Zaharia, and Nickolai Zeldovich. 2015. Vuvuzela: Scalable private messaging resistant to traffic analysis. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 137–152.

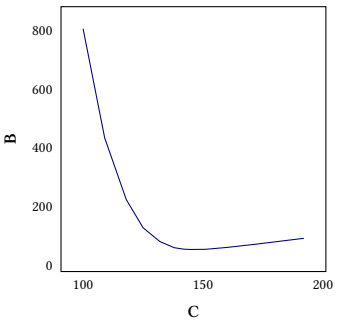

Figure 7: Number of fake messages B, assuming a total of 100 vendors and A follows $\mathcal{N}(100, 20)$.

[53] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. 2012. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*. 179–182.
[54] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. 2013. Hang with your buddies to resist intersection attacks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*. 1153–1166.
[55] Yinglian Xie, Fang Yu, Kannan Achan, Eliot Gillum, Moises Goldszmidt, and Ted Wobber. 2007. How dynamic are IP addresses?. In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*. 301–312.
[56] Rixin Xu, Qiang Zeng, Liehuang Zhu, Haotian Chi, Xiaojiang Du, and Mohsen Guizani. 2019. Privacy Leakage in Smart Homes and Its Mitigation: IFTTT as a Case Study. *IEEE Access* 7 (2019), 63457–63471.
[57] Keyang Yu, Qi Li, Dong Chen, Mohammad Rahman, and Shiqiang Wang. 2021. Privacyguard: Enhancing smart home user privacy. In *Proceedings of the 20th International Conference on Information Processing in Sensor Networks (co-located with CPS-IoT Week 2021)*. 62–76.
[58] 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*. 273–282.
[59] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. 2018. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. 1074–1088.
[60] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. 2018. User perceptions of smart home IoT privacy. *Proceedings of the ACM on human-computer interaction* 2, CSCW (2018), 1–20.

A ASYNCHRONOUS RESPONSES

In a real-world setting, not all devices will have responses ready by the end of the round. Some devices may need a longer period of time to execute the user's command before it can respond. In cases where the response for a command in round t is received in round $t + i$, the vendor V may simply encode another OKVS S' using this response and send it to the integrator I, who then re-executes DecodeCommands using the internal state it stored from round t. This asynchronous approach would incur some small communication overhead between I and the users who participated in round t, as I will send a message to each of these users whenever it runs DecodeCommands.

B CHOOSING THE EXPECTED NUMBER OF COMMANDS

The optimal choice of $\mathbf{C}_v$ for a vendor v depends on how many real commands this vendor usually receives in each round, or more precisely, the distribution of this vendor's $\mathbf{A}_v$. For example, if $\mathbf{A}_v$ follows the normal distribution $\mathcal{N}(100, 20)$ and there are 100 vendors in the system, then Fig. 7 shows how the number of fake commands we need to inject changes as $\mathbf{C}_v$ changes and the number reaches a minimum when $\mathbf{C}_v$ is around 147. Setting $\mathbf{C}_v$ too small will cause traffic bursts to happen more frequently, while setting it too large will lead to more fake messages generated when there is no traffic burst. We note that in some scenarios where we do not even want the shuffler to learn the distribution of $\mathbf{A}_v$, we can draw a new $\mathbf{C}_v$ from some pre-determined distribution in every round, instead of setting $\mathbf{C}_v$ to a constant value.

In addition, recall that our threat model permits an IoT service to collude with a small number of users and may generate up to δ commands in each round. To account for this, we should add δ to the optimal choice of $\mathbf{C}_v$ discussed above to ensure that, no matter how many commands these colluding users generate, they cannot cause a traffic burst and manipulate the number of commands returned to the integrator.

C SECURITY OF Mohito

Data privacy. The privacy of the Mohito protocol is ensured via end-to-end encryption. Specifically, each user command and each device response is encrypted using a symmetric encryption key k_D, which is generated during the device setup phase and is shared between only the user and its device.

Data integrity. Since we assume IoT servers are semi-honest, the only party that can tamper with the integrity of data is a malicious user or device. To do so, the malicious user or device must guess the ephemeral id generated by an honest user or device. However, since the ephemeral id is the output of a PRF and the key of the PRF is not known to the malicious user or device, the probability of correctly guessing the ephemeral id is negligible.

Metadata privacy (integrator). We model the adversary $\mathcal{A}_I$ as a party that passively corrupts the integrator and actively corrupts a small subset of users and devices, as discussed in Section 3.2. In each round, it receives the following information during the command sending phase: 1) a list of users and their messages to the integrator, and 2) a list of messages from the shuffler. We formalize the definition of metadata privacy by specifying a simulator algorithm that, given the list of honest users in this round as well as the list of messages generated by malicious users controlled by $\mathcal{A}_I$, produces an output that is computationally indistinguishable from the information listed above.

Intuitively, this means that $\mathcal{A}_I$ learns nothing in each round, as everything (apart from the messages generated by $\mathcal{A}_I$ itself) it observes can be simulated by an algorithm that has no knowledge of the honest users' commands.

Claim. There exists an algorithm Sim_I *that takes as input the list of honest users and a list of messages generated by the adversary-controlled users in a round and simulates the view of the adversary* $\mathcal{A}_I$.

Proof Sketch. The algorithm Sim_I simulates messages from honest users by encrypting random values. These simulated messages are indistinguishable from real messages due to the security of the encryption scheme. Next, Sim_I plays the role of the shuffler. When the integrator requests to shuffle the messages, Sim_I decrypts the

adversary-generated messages and places them in random slots of the return list. The rest of the return list is filled with random values encrypted by the integrator's public key. This list is indistinguishable from the list returned by a real shuffler, as the adversary does not know the random permutation used for shuffling. Note that the adversary does learn a *partial* permutation, corresponding to the adversarially chosen messages, but this does not reveal any other parts of the permutation.

Metadata privacy (vendor). Similar to the metadata privacy of integrator, we model the adversary $\mathcal{A}_V$ as a party that passively corrupts a vendor and actively corrupts a small subset of users and devices and show that a simulator algorithm exists to simulate the view of $\mathcal{A}_V$. In particular, we assume that this vendor is also acting as the shuffler.

Claim. There exists an algorithm Sim_V *that takes as input the list of honest users and a list of messages generated by the adversary-controlled users in a round and simulates the view of the adversary* $\mathcal{A}_V$.

Proof Sketch. The algorithm Sim_V first simulates messages that the shuffler receives by encrypting random values with the shuffler's public key and attaching them to the list of messages generated by the adversary-controlled users. Then Sim_V generates a list of random values and encodes them (along with the commands generated by the adversary-controlled users) into an OKVS with random keys. The resulting OKVS is sent to $\mathcal{A}_V$. Due to the obliviousness of OKVS and the fact that all encoded values appear random, $\mathcal{A}_V$ cannot distinguish this OKVS from an OKVS generated from real users' commands.