Securing User Data in Online Integration Platforms: From Risk Assessment to System Design

by

Yunang Chen

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN–MADISON

2023

Date of final oral examination: 10/02/2023

The dissertation is approved by the following members of the Final Oral Committee:

Rahul Chatterjee, Assistant Professor, Computer Sciences
Earlence Fernandes, Assistant Professor, Computer Science and Engineering, University of California San Diego
Kassem Fawaz, Associate Professor, Electrical and Computer Engineering
Somesh Jha, Professor, Computer Sciences

To the memory of my dear grandfather, Dianhuang Xu, whose wisdom, kindness, and love have illuminated my path since my childhood.

ACKNOWLEDGMENTS

I am profoundly grateful for the invaluable support and guidance I received throughout my doctoral journey. What once felt like an endless journey six years ago has finally reached its destination, all thanks to the incredible help I received along the way.

First and foremost, I would like to express my gratitude and appreciation to my advisors, Prof. Earlence Fernandes and Prof. Rahul Chatterjee, whose guidance, patience, and encouragement have supported me throughout my PhD. Earlence first introduced me to the intricate world of security and privacy research at a time when I possessed limited experience of the field. Since then, he has not only guided me through the initial challenges and struggles but also been constantly encouraging and empowering me to forge my own unique research path. Rahul has been instrumental in assisting me with his technical expertise. There were instances during my research journey when I found myself stuck with complex problems without appropriate tools. Rahul always stepped in, suggesting a range of promising directions that ultimately steered me towards overcoming these challenges. I am indebted to their collective expertise and unwavering support, without which my academic achievement would not have been possible.

I extend my sincere thanks to other members of my dissertation committee and the faculty members who generously supported me: Prof. Kassem Fawaz, Prof. Somesh Jha, Prof. Andrei Sabelfeld, Prof. Danny Huang, and Prof. David Heath. Their insightful contributions and constructive feedback were instrumental in refining the quality of this dissertation and my research. I also want to give a huge shout-out to my awesome collaborators and labmates in MadS&P — Amrita Roy Chowdhury, Ruizhe Wang, Mohannad Alhanahnah, Yue Gao, Rose Ceccio, Mazharul Islam, Ashish Hooda, and the rest of the crew. We teamed up on loads of stuffs and they

have made my research journey a whole lot more enjoyable.

Finally, my family deserves my utmost gratitude. From the earliest days of my childhood to the culmination of my doctoral journey, their encouragement during moments of doubt, their unwavering belief in my abilities, and their endless patience have been my constant pillars. My parents, even though we could not physically meet up for four years, maintained an unbroken weekly tradition of connecting through Facetime. Their pep talks are confidence boosters, comfort bringers, and problem-solving sessions for all kinds of my nuanced troubles. Their understanding and continuous support were pivotal in helping me conquer the most rigorous challenges of a PhD. For that, I am eternally grateful.

CONTENTS

LIST OF FIGURES

ABSTRACT

Online integration platforms have become integral components of modern digital ecosystems. These platforms establish connections with numerous third-party services, enabling a seamless exchange of data and interactions among these services. This pervasive connectivity, while enhancing efficiency and convenience, has raised various security and privacy concerns. This dissertation delves into such multifaceted challenges posed by online integration platforms, focusing on two primary threat vectors.

First, we investigate the vulnerabilities that arise from the design of the integration platform's access control. These platforms typically employ permission-based models to regulate the extent of access granted to third-party services. However, many of these models are inadequately designed, which leaves them susceptible to exploitation by malicious third-party services seeking to escalate their privileges and gain unauthorized access to user data. To illustrate these vulnerabilities, we conduct a systemic analysis of the app integration platforms in team-based business collaboration platforms (Slack and Microsoft Teams). Our study reveals that an adversarial-controlled app not only poses a direct threat to the data within the platforms but may also jeopardize the security of other connected third parties.

Second, even when the access control of an integration platform is appropriately designed and implemented, the platform itself still poses a privacy concern by design. Integration platforms inherently possess vast repositories of user data, as they act as centralized hubs through which data from various third-party services pass. Consequently, the platforms have the capability to accumulate extensive information about users and their activities, creating a potential risk to user privacy. Given the intrinsic nature of this issue, it calls for the development of novel system designs. In particular, we focus on two popular types of integration platforms,

namely trigger-action platforms and smart-home platforms. Through a comprehensive examination of their specific security requirements and data communication flow, we propose secure and efficient protocols designed to safeguard user data from the prying eyes of these platforms.

Due to the connections with various third-party services, a compromise to the security of integration platforms has cascading effects. Securing integration platforms is therefore a vital part of the protection of user data. By focusing on several distinct but representative types of integration platforms, this dissertation aims to contribute valuable insights and practical solutions to enhance the security and privacy of user data in the evolving landscape of digital interactions among online services.

1 INTRODUCTION

In an era marked by the proliferation of digital services and the ever-expanding scope of the Internet, users have increasingly embraced a multitude of services to manage various aspects of their digital lives. Online integration platforms have thus emerged as indispensable cornerstones of modern digital ecosystems, since they play a vital role in bridging the gap between a diverse array of digital services that do not directly communicate with each other. These platforms facilitate seamless interaction and data exchange between third-party services. For instance, trigger-action platforms, one popular type of integration platforms, allow users to connect an email service to a smart speaker and set up an automation rule that rings the speaker each time a new email is received [26]. Such pervasive connectivity of the integration platforms has enabled a more efficient and convenient digital landscape [146].

However, integration platforms also introduce a series of security and privacy concerns. These platforms inherently serve as central hubs through which data flows from various third-party services, and hence a compromise to the security of an integration platform can have far-reaching consequences. Any vulnerability or breach can be exploited to gain unauthorized access to a wealth of user data, with repercussions that extend not only to the user data within the platform but also to the connected third-party services. The diverse range of these third-party services may thus lead to a spectrum of potential risks, ranging from phishing attempts and loss of sensitive information to life-threatening attacks, such as malicious manipulation of critical components in energy distribution and medical systems.

In light of these risks, the task of securing integration platforms takes on paramount importance and motivates the overarching question to answer in this dissertation:

What are the prevalent security and privacy risks to user data in online integration platforms and how can we develop robust security mechanisms to mitigate these risks?

To solve this question, we take a comprehensive approach by focusing on the threat vectors that originate from the two primary players in the system: the connected third-party services and the integration platforms themselves.

The first threat arises from third-party services under adversarial control. Integration platforms commonly rely on permission-based models to govern the scope of access granted to third-party services [76, 101]. Nonetheless, when these access control schemes are not adequately designed, they introduce vulnerabilities that can be exploited by malicious third-party services. These adversaries can manipulate the system to escalate their privileges, ultimately obtaining illicit access to user data to which they should not have any entitlement.

The second threat stems from the platforms themselves, constituting a more direct and inherent challenge. Even when the access control of an integration platform is appropriately designed and implemented, the platform still naturally accumulates a vast amount of user data, enabling it to infer a wealth of information pertaining to users and their daily activities [102]. This aggregation of data, while essential for the platform's intended function, simultaneously poses a risk to user privacy.

This dissertation examines both of these threats and, through the case study of selected integration platforms, aims to understand the scope of the associated risks as well as their possible mitigations, thereby advancing our efforts toward secure and privacy-preserving integration platforms.

1.1 Contributions

Our contribution towards securing user data in integration platforms comes in twofold. Firstly, we assess the vulnerabilities stemming from the inadequate design of their access control models for third-party services. We use the app ecosystem in business collaboration platforms as a case study to illustrate such vulnerabilities, given the complex array of functionalities and permissions offered in these systems. Secondly, we address the inherent security and privacy issues within these integration platforms by introducing new system designs. Our focus in this endeavor is targeted at two types of integration platforms, namely trigger-action platforms and smart-home platforms, chosen due to their extensive integration with a vast number of third-party services.

Through the study of these distinct yet representative types of integration platforms, we have contributed valuable insights and practical solutions to enhance the security and privacy of user data, which are summarized in the thesis statement below and then expanded upon in the rest of this section.

Thesis statement. Securing user data in online integration platforms necessitates a dual defense strategy against threats originating from third parties and from the platforms themselves. Addressing the former requires dynamic and fine-grained access control mechanisms, while the latter demands the development of specialized protocols tailored to each platform's distinct security requirements and communication flows.

Contributions to security analysis of third-party access control in integration platforms [81]

Business collaboration platforms. Business collaboration platforms like Microsoft Teams and Slack enable teamwork by supporting text chatting

and third-party resource integration. A user can access online file storage, make video calls, and manage a code repository, all from within the platform, thus making them a hub for sensitive communication and resources. The key enabler for these productivity features is a third-party app ecosystem. We contribute an experimental security analysis of the access control scheme in Microsoft Teams and Slack. To guide the analysis, we derive a common permission model for these two BCPs and then experimentally examine each interaction method between apps and users. Specifically, we introduce three new attack classes that leverage fundamental shortcomings of the access control model: app-to-app delegation attacks, user-to-app interaction hijacking, and app-to-user confidentiality violations. We constructed proof-of-concept attacks for these classes to achieve effects such as sending arbitrary emails on behalf of victims, merging code requests, launching fake video calls with loose security settings, and stealing private messages without having the appropriate permission. Finally, we provide an analysis of countermeasures that these business collaboration platforms can adopt today.

Contributions to system designs towards secure and privacy-preserving integration platforms [78, 79]

Trigger-action platforms. Trigger-action platforms, such as IFTTT, Zapier, and Microsoft Power Automate, enable millions of end-users to automate interactions between a wide variety of third-party services ranging from cloud services to IoT device vendors and social networks. End-users create simple automation rules using the trigger-action paradigm. For example, one can connect Outlook email with a smart speaker so that whenever an email that contains the keyword "Important" arrives at the inbox will trigger a notification from the speaker. Unfortunately, the current design of TAPs is flawed from a security and privacy perspective, allowing un-

fettered access to sensitive user data from connected third-party services. To address this issue, we present two different systems, each occupying a different point in the design spectrum:

- First, we have eTAP, a privacy-enhancing protocol that allows trigger-action platforms to execute automation rules without accessing users' private data in plaintext. We use garbled circuits as a primitive to support a commonly used set of computations in user-created automation rules, and leverage the unique structure of trigger-action platform to make the protocol practical. We formally state and prove the security guarantees of our protocols. We implement and evaluate eTAP. It can support 93.4% of computational rules in Zapier and 100% of the 500 most-used rules in IFTTT. We show that most functions can be evaluated with a modest performance impact: on average rule execution latency increases by 70 ms, or 55% when compared to an insecure baseline system.

- Next, we have minTAP, a more lightweight approach that provides data access minimization for trigger-action platforms. The goal of minTAP is to mitigate the attribute-level overprivilege in automation rules. Instead of preventing all plaintext access, minTAP releases only the necessary attributes of user data to trigger-action platforms and fends off unrelated API access, by leveraging language-based data minimization to apply the principle of least-privilege. Using real user-created rules on IFTTT, we demonstrate that minTAP sanitizes a median of 4 sensitive data attributes per rule with less performance overhead (5 ms) and does not require any modifications to IFTTT.

Smart-home systems. Modern IoT services feature an integrator service and several device vendors. The integrator acts as an intermediary by providing a centralized interface that allows users to remotely control devices from various vendors. To achieve this, the integrator forwards

communications between users and vendors. While such IoT services provide benefits, they also observe interactions between users and devices, which can be used to infer sensitive personal information, leading to privacy concerns. 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 metadata, such that no one can learn 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 practical protocol that is capable of 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 metadata-hiding systems that provide similar security guarantees.

Other technical contributions

In the process of designing secure protocols for integration platforms, we have also developed a generalized technique for oblivious evaluation of regular expression (Section 3.5). This technique holds a broader application scope beyond the context of integration platforms — any systems that need to evaluate regular expressions without learning input strings can utilize this technique. Specifically, we devise a way to convert the transition function of a deterministic finite automaton into a Boolean circuit that can be efficiently encoded into a garbled circuit.

1.2 Related Work

In this section, we provide an overview of related research work. The aim of this section is twofold: to contextualize our work within the current security and privacy landscape of integration platforms, and to establish links with existing work in the broader domains of authorization, access

control, and data privacy. Subsequent chapters delve into more detailed discussions of related research tailored to the specific focus of each chapter.

Access control in integration platforms

OAuth [158], an open standard for access delegation, is commonly used by integration platforms to define the set of resources that a connected party can access. Despite the wide adoption, OAuth-based access control systems are usually poorly implemented. Studies [76,179,189] have shown that developers tend to make many mistakes when implementing OAuth, such as exposing application secrets, redirecting secret tokens arbitrarily, or even inventing home-brewed and insecure OAuth protocol flows. These mistakes ultimately undermine the security properties of OAuth and leave the systems vulnerable to attacks.

Moreover, the OAuth protocol does not specify how permissions should be defined. Therefore, overprivileged access is a common issue in OAuth-based systems, even when there is no implementation flaw. This issue is particularly pronounced on IoT platforms, where a line of work [71,73,101, 115,126] has demonstrated that third-party applications are frequently granted access to a significantly larger volume of resources than they actually utilize. Similar findings are corroborated by studies conducted on mobile platforms [96,125], voice assistant platforms [178], and trigger-action platforms [102].

A remark by Chen et al. [76] aptly pinpoints the root cause of these problems — the initial objective of OAuth was to simply serve the authorization needs for traditional websites, but the protocol has been significantly repurposed and re-targeted over the years. Consequently, the specification of OAuth may not satisfy the requirements of these emerging platforms. Our work chooses to examine the OAuth-based access control system in business collaboration platform, a recently established integration platform tailored to meet the surging demands of remote work. Compared to

the integration platforms studied in the prior research endeavors, business collaboration platforms offer a broader spectrum of functionalities and enable more direct communication channels between users and third-party applications. This higher level of intricacy results in more substantial challenges when it comes to designing a secure access control system, potentially leading to new attack vectors and vulnerabilities.

Designing secure integration platforms

The research community has recently shown interest in designing new privacy-preserving protocols for trigger-action platforms, due to the generalizability of these platforms' trigger-action style computational paradigm.

Least-privilege. The problem of overprivileged access isn't confined to third-party applications alone; it also extends its influence to the platforms themselves, turning them into a privacy threat. To solve this problem, several solutions, such as SPKI/SDSI [88] and Macaroon [66], have been proposed to facilitate finer-grained authorization by attaching predicate conditions to access tokens. However, while they offer a generic approach, they do not seamlessly integrate into the data flow of trigger-action platforms, nor are they readily adaptable to capture the expressive nature of the trigger-action style computational paradigm. DTAP [102] extends upon Macaroon's idea to build a protocol tailored for trigger-action platforms to ensure all tokens acquired by the platforms are not overpriviledged. Our work in Chapter 4 goes further to craft a more fine-grained access control system by delving into the data attribute level and allowing for a more flexible way to express the predicate condition.

Encryption. OTAP [84] takes a different approach to secure trigger-action platforms. It uses end-to-end encryption to fully protect the integrity and confidentiality of data while it transits through an untrusted platform. Its main drawback is that no computation is allowed — a primary feature for

trigger-action platforms. While there are some works that utilize hardware-based trusted execution environments (TEEs) to enable computations [172, 206], our work in Chapter 3 proposes a purely cryptographic solution that avoids the underlying security design issues in TEEs [77, 154, 186].

Metadata-hiding. Filter-and-Fuzz [198] achieves metadata protection by instructing smart home devices to generate cover traffics, successfully hiding the timestamp of each message. In comparison, our work in Chapter 5 additionally hides the metadata information of which sender is communicating with which receiver and employs a more efficient server-side cover traffic scheme.

1.3 Organization of Dissertation

This dissertation is organized as follows. In Chapter 2, we contribute to the understanding of security vulnerabilities resulting from the access control model in integration platforms with a systematic analysis of the third-party app ecosystems in two widely-used business collaboration platforms, Microsoft Teams and Slack. In Chapters 3 to 5, we describe three different designs to address the privacy concerns stemming from the design of today's integration platforms. Specifically, we propose eTAP and minTAP for trigger-action platforms and Mohito for smart-home platforms. Each of these designs provides a different trade-off among security, performance, and functionality. Finally in Chapter 6, we conclude with remarks summarizing the contributions of this dissertation and examine potential future work.

2 Security Analysis of Access Control in Business Collaboration Platforms' App Ecosystem

In this chapter, we delve into the first category of threats to user data in online integration platforms – that is, the attempts by malicious third parties seeking to circumvent the access control enforced by the platforms. We use the third-party app ecosystem in business collaboration platforms as a case study to illustrate to scope of this threat and propose a set of possible countermeasures. These platforms offer a broader spectrum of functionalities and enable more direct communication channels between users and third parties, thus resulting in more substantial challenges when it comes to designing a secure access control system and leading to new attack vectors and vulnerabilities.

2.1 Introduction

Business Collaboration Platforms (BCPs) like Slack and Microsoft Teams are indispensable collaboration and productivity tools. Beyond multi-user chat features, BCPs enhance productivity by allowing users to integrate third-party resources. For example, users can make video calls with Zoom, store files on DropBox, chat with customers, and manage code repositories, all from within the BCP. A vibrant third-party app ecosystem allows many such integrations. Thus, BCPs not only host private communications between users but also serve as a hub for all their sensitive resources from third-party systems. As such, it is vital to understand the security and privacy properties of this emerging class of distributed multi-user collaboration platforms.

We contribute to understanding the security of BCPs by performing an experimental analysis of the third-party app model. We focus on the app model because it allows BCPs to access sensitive data from third-party

systems. Although there is work on understanding the operational security issues of BCPs (e.g., web security flaws [38, 39]), to our knowledge, no work has examined the third-party app model. We focus our work on Slack and Microsoft Teams — two of the most widely-used BCPs with mature app ecosystems [19]. Furthermore, these two systems share design-level commonalities and potentially with other BCPs. Thus, any security findings are potentially broadly applicable to BCP design.

Performing the security analysis of Slack and Microsoft Teams is challenging because these systems, including their apps, are closed-source. Specifically, apps themselves are remotely-hosted web services whose endpoints are only known to the BCP. This precludes classical analysis techniques such as source code and binary analysis or API endpoint testing. As an external party, we can only interact with apps the way a human user would — through the BCP itself. Therefore, we focus our analysis efforts on the *interactions* between apps and users, such as sending messages and reacting to them. To conduct the analysis methodically, we first systematize an access control model that describes the approaches taken by Slack and Teams using a uniform vocabulary. We then explore how an attacker can violate the access control model by experimentally studying each interaction method.

We find that the BCP app model uses a two-level access control system consisting of the OAuth protocol and a runtime policy enforcer. Abstractly, a BCP app requests OAuth tokens to interact with categories of resources. For example, an app might request an OAuth token to read chat messages. However, this token does not entirely dictate what specific messages the app can read. Thus, the user has to specify the fine-grained access control policy at runtime. Once the user installs an app and permits it to read chat messages, the user can additionally specify that the app may read messages from specific channels (e.g., the "usenix-security-submission" channel). Whenever an app issues an API request to the BCP server to

read a chat message from a specific channel, the access control system first verifies the OAuth token and then executes a runtime policy check to verify that the app is authorized to read from that specific channel.

By examining each interaction method between BCP apps and users, we establish that this two-level access control system does not adequately confine third-party application behavior. Concretely, we have discovered that the BCP access control system violates two standard security principles: (1) *least privilege* and (2) *complete mediation* [169]. This allows malicious apps to escalate their privilege and violate the confidentiality and integrity of private chat messages and third-party resources connected to BCPs. To demonstrate the concrete harms posed to end-users, we introduce three attack classes for BCPs along with attack prototypes:

(***1***) ***App-to-App Delegation Attacks*** (***Section 2.4***)***:*** BCPs support apps that can interact with each other for productivity reasons, independently of human involvement. To support such meaningful interactions, the BCP access control model allows apps to act on behalf of a user. We show how malicious apps can exploit this to violate the confidentiality and integrity of resources that victim apps manage. Our proof-of-concept attacks include sending arbitrary emails on a victim's behalf, merging code pull requests, and retweeting any links using the victim's account.

(***2***) ***User-to-App Interaction Hijacking*** (***Section 2.5***)***:*** BCP apps can customize how users interact with them and with workspace features. For example, an app can introduce new 'slash commands' into a workspace or manipulate how URLs get unfurled. For example, one can start a Zoom video call by entering `/zoom` on the Slack UI. We show how a second malicious app can interfere when a user attempts to interact with a benign app, a problem similar to DNS domain squatting and voice assistant skill squatting [134, 207].

(***3***) ***App-to-User Confidentiality Violations*** (***Section 2.6***)***:*** BCP apps interact with users by participating in any approved channels or conversations,

where a human user explicitly 'adds' the app as a member. BCPs implement runtime policy checks to enforce security policies in these situations. We show how a malicious app can exploit gaps between OAuth and these runtime mechanisms to leak private messages it does not have permission to view.

Finally, we propose a set of countermeasures that BCPs like Microsoft Teams and Slack can adopt today as a temporary solution to mitigate the attacks (Section 2.7). For example, enforcing user confirmation before every app-to-app interaction and command name collision can fix most issues, but this is undoubtedly a user-hostile solution. As a result, solutions with acceptable security and usability trade-offs necessitate rethinking the app and access control model in multi-user communication platforms.

Ethics and Disclosure. We conducted all experiments inside private workspaces with the authors as the only members. We did not exercise cross-workspace features; thus, our investigations did not influence other workspaces. We did not distribute or submit our test malicious apps to any BCP app directory, so our attack did not affect BCP users other than the authors' testing accounts. We ethically disclosed all attacks we found to Slack and Microsoft, both of which have confirmed their existence. Due to their view of the workspace as a trusted environment, the assumptions that social engineering is a prerequisite for the attacks, and that the workspace administrator will correctly manage app installations, these attacks do not meet their definitions of a security vulnerability.

2.2 Business Collaboration Platforms

BCPs provide chatrooms that facilitate online collaboration among a group of people, who usually belong to the same workspace, such as a project team or a research group. In BCPs, one can create a virtual *workspace* to host all conversations for a group. It supports discussions among the users

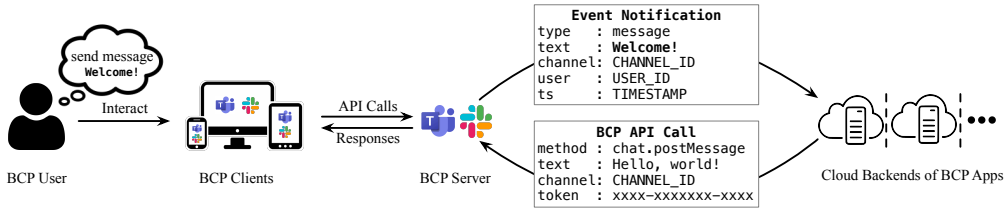


Figure 2.1: Overview of BCP's ecosystem: A *BCP user* interacts with their *BCP clients* to communicate with the *BCP server*. BCP apps, which are maintained as separate web services by different third-party developers, communicate with BCP server via API calls and event notifications. A user has to install and authorize an app before accessing its functionalities.

who joined the workspace through various conversation *channels*. Users can open a new channel which can be *public* — any user can join — or *private* — only those who are invited can join. Users can also send *direct messages* to any other user or group of users in the workspace. To use a BCP, a human user interacts with their BCP client on their computer or mobile device, which then communicates with the backend servers of the BCP through various APIs. The backend server then responds to the client, updating what the user sees. We illustrate this communications framework in Figure 2.1.

In this work, we focus on **Microsoft Teams** and **Slack**, due to their popularity and mature third-party app ecosystem. A recent survey of 900 businesses [19] has shown that they are the two most popular BCPs[1]and are the only ones that provide a list of officially supported third-party apps.

BCP App

Beyond basic chatting features, modern BCPs usually offer many third-party integrations, commonly known as *apps*, which are cloud services pro-

[1]The original survey listed Skype for Business as the top spot, but it has since been discontinued and replaced by Microsoft Teams.

viding additional productivity-enhancing functionalities in the workspace, often connecting user's data from other services (such as email or online storage) to the workspace. These BCP apps exist on cloud servers not maintained by the BCP. These app backends communicate with the BCP servers by subscribing to event notification APIs and reacting when information about a new event is received, as depicted in Figure 2.1. Generally, a BCP app can simultaneously act in three roles: workspace feature provider, interactive bot, and user delegate.

Workspace feature provider. The app may enhance a workspace's existing features. For example, an app made by Twitter can customize the default *link unfurling* feature to preview tweets linked in messages automatically. The app may also provide user-invokable actions through *slash commands*. As another example, Google's Slack app [10] shows a user's recent schedule when the user types `/gcal`.

Interactive bot. The app can present itself in the workplace as a bot user and interact with other users the same way as a typical human user. The user can, for example, chat with the app's bot user directly, invite it to a channel, or share files with it. Due to these convenient features, this role has become the app's primary communication interface with its users.

User delegate. If permitted, the app may also perform actions on behalf of users. This role is particularly beneficial for enhancing productivity. For example, when users visit Dropbox's web page and wish to share files with others in their Slack workspace, they must divert their attention back and forth between Dropbox and Slack. In contrast, with the delegation ability, Dropbox enables the user to click a button without leaving the webpage and let Dropbox's Slack app [8] share files on their behalf. As a result, the shared files appear to have been sent directly from the user.

Life Cycle of BCP Apps

Microsoft Teams and Slack allow any BCP user to create and distribute BCP apps without requirements, such as applying for a developer account. BCP apps generally go through the following stages in their life cycle: registration, publication, installation, per-user authorization, in-use, and removal.

Registration. To enable the various functionalities in Section 2.2, an app needs to query different web APIs or subscribe to different event notification APIs on the BCP's backend server, which in turn usually require different permissions. The app developer must register the app in the corresponding BCP's developer portal by submitting a manifest, which specifies the app's backend URL, required permissions, and subscribed events. We note that, in both Microsoft Teams and Slack, the developer does not need to submit any of the app's codebase, as all their apps are hosted purely inside the developer's server. No client-side code is accessible by Slack, Microsoft, or the end-users.

Publication. After the app has been successfully registered, the developer can choose to either distribute the app's public installation URL through its own advertising channels or submit the app to the official app directory [33, 36]. For the second option, the app must follow submission guidelines and go through the platform's vetting procedure, which primarily involves checking if the app's requested permissions match its claimed functionality (e.g., through a provided test account). However, as BCP apps are closed-source and their codes are not submitted for examination, it is difficult to enforce these guidelines strictly.

Installation. In Microsoft Teams and Slack, any user[2] can install an app to the workspace. During installation, a permission request page will be presented to the user, detailing what the app can do, as illustrated in Figure 2.2. The user then either accepts all permissions or rejects all

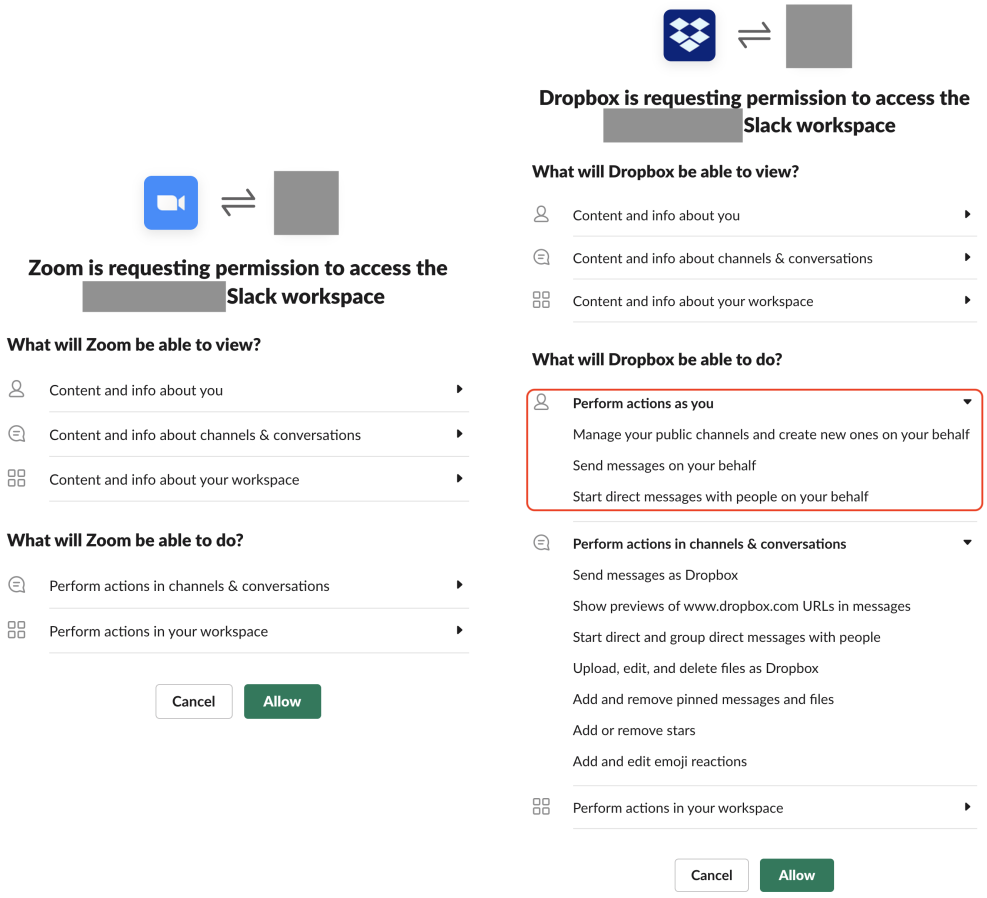


Figure 2.2: Installing apps with bot scopes (left) and user scopes (right) in Slack.

permissions. This installation is relatively invisible to other users; they are not notified when a new app is installed, and the list of installed apps is often hidden in secondary menus in the UI.

Per-User Authorization. If an app wants to act as the delegate of some users in the workspace, it may initiate a separate permission request to

[2] Although Microsoft Teams and Slack provide a setting for the administrators of a workspace to limit which users are allowed to install apps and which apps can be installed, the default for both BCPs is that any user can install any apps from any source.

each user, usually by sending the request link via the app's bot user. Once the user authorizes it, the app gains permission to act on behalf of that user.

In-use and Removal. After the app is installed and authorized, it may additionally ask for integration with the user's account on third-party services. For example, Google's Slack app requests the user to authorize access to their Google account. BCPs do not manage the communications between BCP apps and third-party services. If the app developer updates an app to request a different set of permissions, the user has to reinstall the app and go through the permission prompts as before. Finally, when a user uninstalls an app, it is deauthorized by the BCP. However, there is no guarantee that the app properly disconnects itself from third-party services.

Security and Privacy Concerns

The widespread usage of BCPs in remote work environments implies that a lot of sensitive information passes through it. With the potential ability to access such information, BCP apps lead to security and privacy concerns. Moreover, some of the design choices that we described earlier exacerbate such concerns: (1) *all-or-nothing permissions* that disallow selective toggling of permissions; (2) *imperceptible installation* that reduces the chances for users to notice what kinds of apps are installed and also prevents any workspace-wide consent mechanisms; (3) *pure server-side implementation* that prevents BCPs or other entities from inspecting the app's behavior through traditional tools like static or dynamic analysis. This also allows the app to change its behavior at will.

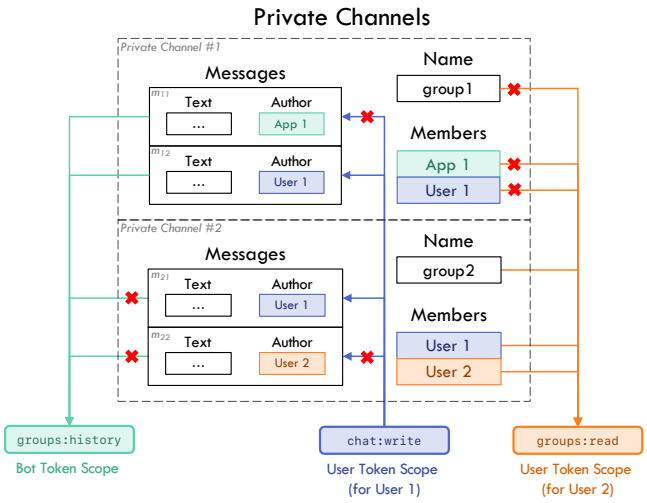


Figure 2.3: An example of Slack permission system. We show three example scopes that App 1 may acquire. The arrow lines indicate that a token can be used to query all resource instances of the types allowed by the token's scope. However, Slack performs additional runtime policy checks (indicated by the red crosses) to determine which of these instances can actually be accessed.

2.3 Analysis of App Permission Model in BCP

We study the permission systems in Microsoft Teams and Slack to identify their similarities and differences to understand the potential security design issues and systematically perform experimental security analysis. We focus on these two BCPs since they are the top two most popular ones [19] and have mature app ecosystems. We also introduce a practical threat model and the methodology we will use to analyze the third-party apps in these two BCPs.

App Permission System

At a high level, Microsoft Teams and Slack have designed their access control model based on a similar permission-based system. This permission

system controls whether or not an app has access to various resources in a workspace. An app must first declare a set of *permission scopes* it requires, with each scope representing the permission to read or write a type of resource. However, such scopes are statically defined by the BCPs and thus do not allow more dynamic and fine-grained access control over the specific instances under a single type of resource. To solve this problem, the BCP permission system includes *runtime policies* that are usually user-configurable. For example, to read a message in a private channel, a Slack app not only needs the `groups:history` scope but also has to be added to the channel's member list by some user, as shown in Fig. 2.3. We now examine this two-level permission system in detail and show that it has security design issues that can violate the least privilege principle and cause privilege escalation.

Level 1: static permission scopes. An app needs to acquire several different permission scopes to perform all of its functionality. Each scope represents the permission to read or write a type of resource in a workspace, such as channel messages or shared files.

To install the app, the user must accept all of its requested permissions; neither BCPs provide options to selectively toggle them. Slack's permission scopes are implemented as standard OAuth permission scopes. Slack provides two types of scopes for its apps: *bot token scope*, which allows an app to provide workspace features or act as a bot user, and *user token scope*, which allows an app to perform actions on behalf of an authorized user. For example, the `chat:write` bot token scope permits the app to send messages with its bot user as the author, while the `chat:write` user token scope allows sending messages as the user. Microsoft Teams follows a similar design: a set of core app capabilities that must be declared in an app's manifest is the equivalent of Slack's bot token scope, while Microsoft Graph API's OAuth permission scopes are equivalent to Slack's user token scope. The difference is that only the first type of scope is shown during

the app installation; the second type can only be acquired by initiating a separate permission request to the user after installation.

These scopes are *static*, in the sense that they are predefined based on how BCPs categorize the workspace resources, and therefore might not align with the user's desired security policies, which can vary by workspaces and evolve. To compensate for the static nature of scopes, both BCPs impose a second level of permission checking.

Level 2: runtime policy checks. Microsoft Teams and Slack implement runtime policies to determine which instances in a resource type an app can access based on various conditions. Users can usually control these conditions to express their desired security policies. For example, users can have more fine-grained control of which messages in private channels an app (that has the prerequisite permission scope) can view: in Slack, they can invite the app to a specific channel, indicating that the app can view all messages inside this channel; in Microsoft Teams, they can `@`-mention the app in the messages that they wish the app to read. In this way, runtime monitors grant users some flexibility to dynamically adjust the set of resources of an app can access.

Security design issues. Despite the two-level checking, we uncover two design issues in the BCP permission system that violate basic security principles.

1. The runtime policies are ad-hoc and incomplete. As a result, not all user security policies can be correctly expressed. We find that not only do they differ in each BCP, but even in the same BCP there are often inconsistencies between the runtime policies of similar types of resources. For example, Slack treats public channel messages and direct messages as two separate types of resources; however, it only imposes a policy on the former by checking whether the app is invited to the channel, but provides no mechanism to limit which

user the app can send direct messages to. The incompleteness of runtime policies leads to coarse-grained access control, violating the principle of *least privilege*.

2. The ownership or provenance of some resources is not properly tracked or enforced. This frequently happens when a user delegates an app to create resources. For example, Microsoft Teams does not differentiate between messages sent by a real user and a delegated app. In addition, due to the multi-user multi-app nature of BCP workspace, the ownership of a resource can sometimes be hard to define correctly. When the ownership or provenance is absent, or the system assumes the wrong one, the principle of *complete mediation* can be violated and potentially lead to privilege escalation.

Although it is possible to build a BCP permission system to fix the above problems by allowing the user to specify the security policy for every instance of resources and tracking every resource's provenance, we will see in Sections 2.4 to 2.5 that such an ideal system is hard to design and often requires sacrificing usability.

Threat Model

Based on our analysis of the permission model above, we derive a threat model for BCP apps. We assume that the attacker has targeted a BCP workspace containing a number of users and already-installed apps. The attacker has also tricked one of the users (referred to as the victim) into installing the attacker-controlled malicious app, i.e., the victim has granted all the permission scopes requested by the malicious app. We believe this is a reasonable assumption, because (1) the malicious app can easily mimic a legitimate app by copying its publicly available manifest, making the two indistinguishable for the victim during installation, and (2) by default, any user in the workspace is allowed to install any app from any source. In our

threat model, the attacker can be either an outsider or a curious user inside the workspace who wants to gain the information they cannot access. For example, an admin can recommend everyone in the organization to install a malicious app (disguised as an innocent management app), hoping to steal chat logs from private channels they are not invited.

In addition, we assume that the BCP's clients and its backend server are secure and do not collude with the attacker — attacking such infrastructure is an orthogonal research direction. Therefore, the capacity of the malicious app is limited to the functionality defined by the BCP's API. We also assume that the other apps installed in the workspace are benign and secure, which means they follow the security guidelines [35,40] and do not contain any implementation-level flaws such as exposing their tokens directly.

Security Analysis Methodology

We perform experimental security analysis on Microsoft Teams and Slack to study how a malicious app (defined by our threat model) can exploit the two security design issues in these two BCPs' permission systems. Specifically, for each potential exploit, we evaluate its *practicality* and *prevalence*.

To explore potential exploits, we examine every type of interaction the malicious app can have with other entities in the workspace and check whether such interaction involves resources that have incomplete runtime policy or suffer from improper ownership tracking. If so, we explore attacks causing security-critical consequences. For each attack, we analyze how it stems from the security design issues in the permission system, how it violates the security principles, and how it jeopardizes the workspace's integrity or confidentiality guarantees expected by the user. We detail our findings in Sections 2.4 to 2.6, and summarize the prerequisites and effect surface for each attack in Fig. 2.4.

For practicality, we build proof-of-concept malicious apps and, if applicable, target the attack on selected apps. Since most apps require a valid third-party account to function properly, running large-scale analysis is infeasible. Thus, we only select a few targeted apps that connect to sensitive resources and test them manually. We only install one targeted app at a time in our test workspace to avoid undesired interference.

For prevalence, we analyze the app's *potential ability* to launch attacks. We collect the requested permissions of all published apps from the two BCPs' official app category[3], and count how many apps have sufficient permissions or resources to launch each attack. It is important to note that our goal is *not* to prove that some specific apps are malicious; we only examine the capabilities granted by various permission scopes and how they can be abused to perform malicious actions. This strategy allows for a sound analysis despite apps being closed-source, as the apps we find indeed have prerequisite permissions to *potentially* launch attacks.

[3]We collected 2,460 apps from the Slack [33] on April 7, 2021 and 1,304 apps from Microsoft Teams [36] on November 17, 2021.

Attack	Slack	Teams	Prerequisites	Attack Effect Surface
Delegation	✓	✓	Permission to perform actions (primarily read & write direct messages) on victim's behalf.	Invoke actions in victim's other apps to manipulate data in victim's connected third-party accounts.
– post app removal	*	✓	App has acquired the above permission before removal.	Incur delegation attack after the app is removed. *In Slack, this can only be achieved via pre-scheduled messages.
Interaction hijacking				
– slash command	✓		Permission to add slash commands.	Hijack any slash command in the workspace stealthily, affecting everyone using the command.
– link unfurl		✓	Permission to provide customized unfurling.	Replace any other app's unfurled content stealthily, affecting the links sent by victim.
Message extraction				
– via link unfurl	✓		Permission to read & write direct messages on victim's behalf.	Read messages in any private channel where victim is a member of.
– via pin/star/reaction	✓		Permission to pin, star, or react to messages on victim's behalf.	Read victim's direct messages and messages in any private channel where victim is a member of.

Figure 2.4: Summary of proof-of-concept attacks and their requirements and threats. Per our threat model, the victim is a user who has authorized all the app's requested permissions.

2.4 App-to-App Delegation Attacks

One of the core functionalities provided by BCP apps is to chat with users through their bot users interactively. However, a BCP app can also send and receive messages on the user's behalf and, therefore, chat with other app bot users. In this section, we present the *delegation attack*, where one malicious app abuses such *app-app interactions* and causes security-critical consequences. We then show that the source of this vulnerability roots in the fundamental design issues of current BCP permission systems — a violation of least privilege.

App-to-App Interactions

Both Microsoft Teams and Slack allow their apps to present themselves in a workspace as bot users so that human users can send direct messages to these bot users to instruct them to perform certain tasks. This functionality is commonly used to let users manage their data in other online services, such as emails and file storage, without leaving the BCP.

At the same time, these two BCPs also allow apps to perform certain actions in the workspace on behalf of the user. If an app sends a message in this way, this message will appear as if the user sent it. Such delegation can be useful to enhance productivity. For example, Dropbox's BCP app [8] utilizes it to share files in channels on behalf of the user. In Slack, this can be achieved if the app has acquired the `chat:write` user token scope in its OAuth permission request with the user; in Microsoft Teams, although none of its standard app capabilities grants permissions to delegate, one can still employ the advanced Microsoft Graph API and ask for the `Chat.ReadWrite` scope.

By combining the above two functionalities, we can enable app-to-app interactions in BCPs: one app that has the delegated permission to send user's messages can interact with another app's bot user. Such interaction

can be beneficial; for example, Dokkio's Slack app [7] can organize files sent by Dropbox's app into a coherent page for the workspace and tag them as shared by different users. Slack regards app-app interaction as an important feature with growing demand [111]. However, allowing one app to communicate with other app's bot users has severe security implications. When the former app turns malicious, it can potentially invoke actions from the latter app, and such actions might affect data in the user's connected third-party account. We refer to attacks exploiting this vulnerability as *delegation attacks*.

We note app-app interactions can happen in other ways. Although receiving a message from the user is the most intuitive trigger event to indicate when the app should perform its actions, an app may subscribe to other triggers as well, like when a file is shared or an emoji reaction is added. As such, apps with delegated permissions to produce these triggers can also launch potential delegation attacks.

Post-removal interactions. Even after an app's removal from the workspace, it can have residual effects that cause delegation attacks. Slack provides its apps the ability to schedule a message to be sent at a future time (using the same `chat:write` user token scope). We find that if the app is removed before the message's scheduled time, its message will still be sent, potentially invoking actions from other apps. In Microsoft Teams, although there is no scheduling feature, this issue is more severe due to its two separate permission schemes. Upon uninstallation, only the app's standard capabilities declared in the manifest will be removed, while its delegation permissions acquired through the Graph API remain entirely intact. Therefore, a user *cannot*, by simply removing a Teams app from the workspace, prevent the app from continuing to send messages on the user's behalf and interact with other apps, allowing the channel for delegation attacks to remain open.

Current defenses. We note that Microsoft Teams and Slack do have

workarounds that can prevent app-to-app interactions. They allow apps to interact with users through alternative ways, such as slash commands and interactive UI windows. This prevents other apps from interfering since neither BCPs allow an app to send slash commands or click buttons in a UI. Slack in particular also tracks which messages are sent by a real user through the Slack client and which are sent by a delegated app, so that the app receiving the messages can choose whether to respond or not. However, both of these mechanisms require the receiving app's developer to decide which actions can be triggered by other apps, but the current design of BCP permission system does not provide any ways for it to learn whether the delegated messages align with the user's actual intent, making it impossible to arrive at the correct decision. As we will discuss in Section 2.7, a principled fix would trade-off functionality or usability.

Delegation Attack

We now focus on the delegation attack targeting both Microsoft Teams apps and Slack apps. We have built a tool that crawls the information of a targeted app from the two BCPs' official app directories and analyzes which trigger events the app is subscribing to. In the case of Microsoft Teams, we can also extract all message keywords that trigger the targeted app's actions. We set up a workspace as defined per our threat model. The attacker app has acquired the appropriate delegated permission from a victim user who has also installed the targeted apps with connection to third-party services. The attacker app produces the trigger events, and we observe whether the targeted app will be tricked into performing the actions (see Section A.1 for more implementation details). Since most apps require a valid third-party account to function properly, performing large-scale automated analysis is infeasible. Thus, in this section, we select a few apps connecting to sensitive third-party resources and manually target them, demonstrating that delegation attacks can indeed trigger

security-critical or privacy-violating actions.

① **Send emails on victim's behalf.** MailClark's Slack app [13] allows sending emails directly from Slack to include non-Slack users in a Slack conversation. MailClark provides a unique email address for a list of non-Slack guests in a channel configured by the user. The email account and the recipients are only accessible to MailClark and the user. The attacker app induces MailClark to send any emails of the attacker's choice to recipients configured by the user. Specifically, the malicious app launches this attack by sending messages to the channel as the user. During this procedure, MailClark will automatically send the attacker's message as an email to all recipients and indicate the author as the user.

② **Chat with victim's website visitors.** Chatlio [5] is a service that lets developers add live chat functionality to their websites. It also provides an accompanying Slack app that automatically forwards any messages of the website visitors to a Slack channel and vice versa. Therefore, website owners can chat with any visitors in real-time through Slack. Unfortunately, this convenient feature makes Chatlio's app a victim of delegation attacks. Our attacker app can post messages directly into the channels used by Chatlio to chat with website visitors and thus launch further phishing attacks or harvest sensitive user info, as it now appears like a trustworthy entity to the visitors.

③ **Merge pull requests in victim's code repository.** BitBucket's Microsoft Teams app [4] will merge a given pull request if it receives a message starting with the keyword `merge`. It will then ask for confirmation, at which point the attacker app can reply with the text `yes` to approve the merge. The attacker app may additionally use the `list` keyword to ask BitBucket's app to display all pull requests in the victim user's connected repos or the `find` keyword to locate a specific pull request. If the repo is public, the attacker can even submit and merge its own pull request,

leading to code poisoning or backdoor injection.

④ **Execute victim's automation flows.** Microsoft Power Automate has a Teams app [41] that, upon receiving the message `Run flow [id]`, will execute the specified automation flow in the user's account. These flows can perform various actions in a wide range of services connected to Power Automate. The app also accepts messages like `List flows` and `Describe flow [id]` that can be utilized by the attacker to learn more about the user's flows and conduct more targeted attacks.

⑤ **Retweet on victim's behalf.** Ziri [15] is a Slack app that helps users interact with tweets in a non-disruptive way. It connects to the user's Twitter account and requests permission to retweet. After that, whenever a Twitter link is shared in Slack, and the user adds a Twitter emoji reaction to that message, Ziri will automatically retweet the shared Twitter on the user's behalf. The attacker app can thus send a message containing a link to a chosen tweet (that includes harmful information) and add an emoji to the message on behalf of the user. After that, Ziri will successfully detect the tweet link and retweet it using the victim user's account. Such uncontrolled tweets can have detrimental effects, especially when the connected account is high profile, such as the organization's official twitter.

Summary. The first four attacks rely on message events to trigger the actions in the targeted app, while the last one relies on a reaction event. We note that once the attacker and targeted apps are installed and properly authorized, the attacks do not require additional user inputs and can happen anytime, even when the user is not logged into its BCP client. In addition, the attacker app can delete the traces of trigger events once the attack is finished, making it even sneakier (since in both BCPs, the permission to send messages or add emoji reactions also grants for free the permission to delete them).

Analysis of Root Cause and Potentially Prevalence

The delegation attack is possible because both BCPs' permission systems violate the principle of *least privilege*. Currently, the permission to send delegated messages is governed by Slack's `chat:write` or Microsoft Teams's `Chat:ReadWrite` scope; however, these two scopes allow the app to send messages to any place that the user has access to, be it a public channel, direct message with other users, or direct message with other app's bot user. In addition, neither BCPs provide additional runtime policies that allows the user to limit the destinations. Therefore, even if the user wants to install a simple app that only sends delegated messages to a small subset of other users for sharing or notification purposes, it must grant this app such overprivileged scopes that inevitable comes with the ability to launch delegation attacks.

App's residual permissions after removal. The reason why a removed app can still keep some residual permission differs in two BCPs. Slack's permission system violates the *principle of complete mediation* by failing to check that the proper provenance of the scheduled message, which is the removed app, should have no permissions at the time when the message is sent. Whereas in Microsoft Teams, it is the result of two separate permission systems: only the app's core capabilities are associated with Teams, while the Graph API's permissions are tied to the user's Microsoft Account (outside the permission system of Teams). Therefore, when the app is uninstalled in Teams, only the former is revoked while the latter is not affected. We note this issue is not Teams-specific, but also exists in other systems when permissions are managed by different trust domains [203].

Potential Prevalence. We report the number of apps capable of executing the delegation attack and that are vulnerable to the attack. For Microsoft Teams, we find vulnerable apps by counting apps that use bot commands capability, as these apps will accept text input from the user (or a delegated

app) to perform various actions. We observe that 427 (33%) of Teams apps use bot commands, implying that they are vulnerable to a delegation attack. However, Teams apps do not list whether they will request any delegated permission since it is acquired through a separate system. For Slack, we find 563 Slack apps (23%) request at least one 'write' user scope, allowing them to interact with other apps adversarially, while 1,493 Slack apps (61%) request at least one 'read' scope, implying that they are subscribing to events in the workspace and thus can be potentially affected by the attack. We note that the measurements for Slack's vulnerable apps are the worst-case estimation. Since these apps are third-party web services with hidden endpoints, it is impossible to learn the app's behavior directly. Furthermore, most apps only perform actions after a third-party account is connected, preventing us from fully automating the evaluation of apps on a large scale. Thus we may miscount apps that (1) have already employed a countermeasure by blindly rejecting delegated messages, (2) subscribe the certain events but never trigger their security-critical actions based on these events.

2.5 User-to-App Interaction Hijacking

BCPs provide various features that serve as entry points for users to interact with apps. Examples of these features includes '@'-mention, slash command, and link unfurling (see Section 2.2). In this section, we discuss how a malicious app exploits such interactions between the user and other apps in the workspace. Specifically, we find two different ways that this can happen: the malicious app can hijack other app's registered slash commands, and replace another app's unfurled link content. In particular, we note that both Microsoft Teams and Slack allow apps to customize their appearance (e.g., name, icon, and description) without restriction. A malicious app can thus completely mimic the appearance of another

app[4]to exploit the above interactions more stealthily. Finally, we analyze the root cause and potential prevalence of these attacks.

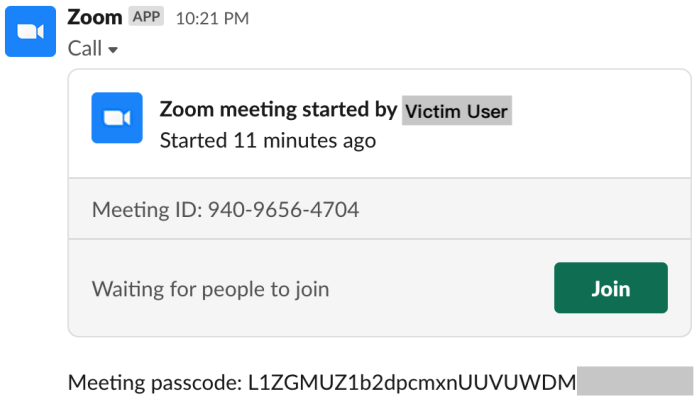
Slash Command Hijacking

In Slack's user-to-app interactions, all apps' slash commands share a single namespace, creating the potential for name collisions. A malicious app can hijack another app's commands, responding to any user that tries to launch the hijacked command in the victim app's stead. Two specific design flaws enable this attack. First, Slack only invokes the most recently installed app when multiple apps in a workspace have registered the same command. Second, both creating and renaming commands are silent and do not trigger a notification or permission prompt in Slack. As a result, one can hijack a targeted command in two ways: (1) create a new command with the same name as the targeted one; (2) rename an existing command to the targeted one. In other words, the `commands` scope becomes over-privileged as it implicitly allows an app to take over any command within a workspace (by exploiting the name collision). However, Slack does not recognize this design issue as a security-critical problem[5]; we find no runtime policy checks of an app's permission to create or rename commands with a specific name.

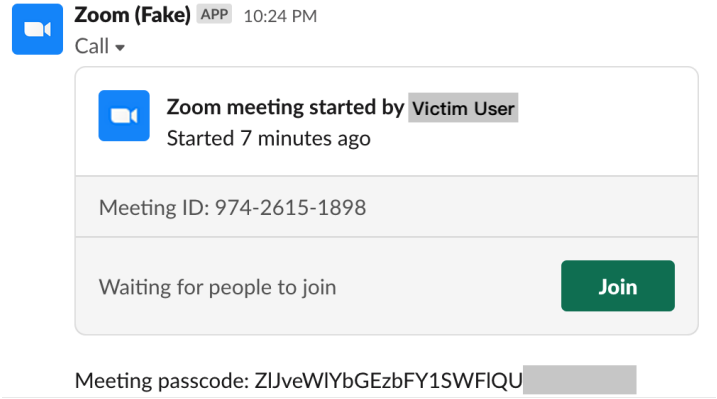
We demonstrate the command hijacking attack on Zoom's Slack app [16]. From Zoom's app, users can invoke the command `/zoom` to start private Zoom meetings and display a Zoom call in Slack, as shown in Figure 2.5a. If the command is invoked in a private channel, only users in this private channel will receive this private call. We create a malicious app that

[4]This may not be the case for apps published in the BCP official catalog, as per their security guidelines. Although a Slack app can still requests `chat:write.customize` to send messages with customized appearance.

[5]Slack acknowledged this problem in its document, but only *suggests* developers to "avoid terms that are ... likely to be duplicated," and not to make the command "too complicated for users to easily remember."



(a) The official Zoom app.



(b) The spoofed Zoom meeting.

Figure 2.5: Zoom meetings created by official and spoofed `/zoom` commands in Slack. The spoofed Zoom meeting is secretly created by the attacker but publicly shown as started by the victim. The word "Fake" is added clear demonstration, it can be removed in practical attacks.

masquerades as the official Zoom app. At the time of installation, our malicious app requests the `commands` scope to implement a benign command called `/foo`. Once installed, we *rename* this command as `/zoom` to hijack the previous official `/zoom` command. After that, the malicious app will use the attacker's Zoom account to start meetings every time a user invokes the `/zoom` command, as shown in Figure 2.5b. Attackers can also treat

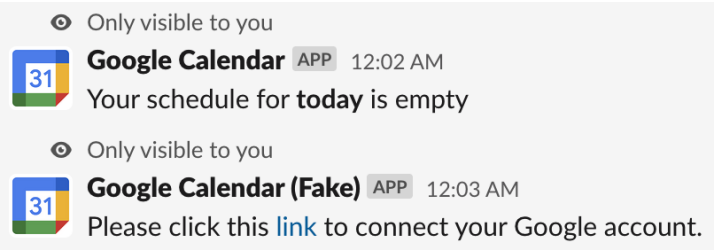


Figure 2.6: Demonstration of phishing attacks using the Command Hijacking attack in Slack. The two messages are sent to the user after invoking the official and hijacked `/gcal` command, respectively. The attacker can start a valid OAuth authorization process to acquire access to the user's account.

this vulnerability as a novel entry point for phishing attacks, as shown in Figure 2.6.

Since Microsoft Teams does not allow apps to register their own commands, it does not suffer from this vulnerability.

Link Unfurling Hijacking

Microsoft Teams allows an app to provide customized link unfurling for an authorized user. The app can register a domain in its manifest. Whenever the user posts a URL under this domain, the app can append a rich message card containing texts, images, or even interactive buttons. For example, Lucidchart's Teams app [12] unfurls a document sharing URL to preview the document as well as a button to accept the sharing invitation. Such unfurled content can be hijacked similarly to Slack's slash command: a malicious app can register the same domain as the victim app and, if the malicious app is installed after the victim app, its unfurled content will be displayed instead of the victim app's one. Moreover, the malicious app can masquerade as the victim app to further deceive the user, as its name and icon will also be part of the unfurled content.

While Slack also allows multiple apps to register the same domain, it

chooses to display all app's unfurled contents in parallel, avoiding the issue of link unfurling hijacking.

Analysis of Root Cause and Potential Prevalence

The command and unfurling hijacking attacks work by violating *least privilege* and *complete mediation*, which results from an overprivileged scope and the improper tracking of resource ownership. First, the corresponding scope that allows an app to use slash commands or unfurl a domain should not spontaneously grant the ability to modify the app's currently registered command names or domains; an app that performs such operation should need to be re-installed. Second, whenever an app registers a command or a domain, it should gain ownership of this command or domain, however, given the namespace collision, both BCPs fail to enforce such ownership, which thus can be easily taken over by another newly-installed app.

Potential Prevalence. In Slack, this slash command attack only exploits the `commands` scope, which is requested by 1,266 apps (51.5%). These apps can immediately overwrite each other's commands to hijack their standard workflows. Recall, once installed, these apps can change their slash commands at any time, without requiring re-installation or notifying the users (or admins) of the workspace. We also find that many apps in the Slack App Directory already have conflicting commands: 270 apps register commands used by other apps. This implies the wide reuse of conflicting commands, and thus Slack is likely to preserve this design choice. In Microsoft Teams, the link unfurl attack relies on the `messageHandlers` capability, which is requested by 77 apps (5.9%). We find that 13 of them register a domain that is also registered by other apps.

2.6 App-to-User Confidentiality Violations

We analyze the different ways in which BCP apps interact with user messages. Our main discovery is that an attacker can leak messages from private channels without having permission to read from those channels. Concretely, we can exploit two features in Slack: (1) Link unfurling of message URLs (Section 2.6); (2) Pinning, starring, or emoji-reacting to messages (Section 2.6). We additionally find that the root cause behind this privilege escalation is incomplete mediation coupled with a lack of ownership tracking of resources (Section 2.6). We note that in Microsoft Teams these features are either absent or inaccessible to apps, so it does not suffer from this vulnerability.

Message Extraction Attack via Link Unfurls

BCPs have a built-in link unfurling feature that previews the website content for any URLs contained in a chat message. We first describe how link unfurling works with message URLs and then show an attack where a malicious app *without* Slack's `groups:history`, the permission scope that controls the read access to messages in private channel, abuses this feature to effectively monitor all chats in any private channel joined by an authorized user.

Unfurling of Message URLs

Slack provides a public URL to every message in a workspace. This URL, if accessed, will only show the message if the login credential of a user who has access to the message is provided. We find that when the user sends a message m_1 in their own *personal channel* (i.e., where users can message themselves) and m_1 contains a URL that links to m_2, where m_2 can be any message in any of the channels that the user is a member of, Slack will automatically unfurl m_2, adding its text content (up to 8001

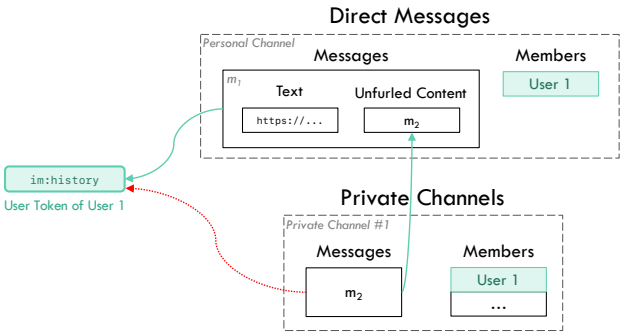


Figure 2.7: Privilege escalation exploiting link unfurling.

characters) and author as an additional attribute to the original message m_1.

While this is a reasonable and useful functionality because the user's personal channel is intended for drafting messages and keeping links and files handy (as described by Slack), it leads to unwarranted access, as illustrated in Fig. 2.7. Slack allows an app with `im:history` user token scope to read the user's personal channel. This grants the app the ability to read m_1 with all its attachments. In this case, the attachments include the unfurled content, which is m_2, a message from a private channel. Therefore, the app is implicitly permitted to read m_2, which is protected under the `groups:history` scope, and the app with only `im:history` does not have access to originally.

Attack Workflow

Now, we present a powerful attack based on the issue identified above. Through this attack, a malicious app can achieve privilege escalation — it gains the ability to monitor all chat messages in any private channel where the victim user is a member of, effectively gaining the permissions provided by the `groups:history` user token scope but without explicitly requesting it.

The key insight enabling this attack is that if the attacker can learn

the message URL of a private channel message, it can then instruct the malicious app to post a generated URL to the victim user's personal channel (using the `chat:write` scope as we described in Section 2.4), actively leaking messages from that private channel. We additionally find that Slack's message URL always follows the format:

"`https://[workspace].slack.com/archives/`
`[channel-ID]/p[message-ID]`"

Therefore, the attacker's job becomes learning valid combinations of channel ID and message ID.

We have discovered several ways to obtain such combinations without resorting to `groups:history` and detailed them in Section A.1. Here we describe one method that utilizes `groups:read`. This user token scope provides the read access to the metadata of the user's private channels, including the channel ID and the ID of the latest message in the channel. By constantly querying a channel's metadata, the attacker can pull every message from any private channel the victim user has joined. We note that even if multiple messages occur between two queries, the attacker can still guess their IDs since Slack's message ID is a counter that increments for consecutive messages (see Section A.1 for details).

Extracting other types of messages and files. This attack also works for other types of messages. An app's bot user can use this to view any public channel messages without the corresponding bot token scope or invitation to join that channel. Additionally, it can even be applied to read files shared with the user. Unlike message URL, there is no easy way to obtain a valid file URL through alternative approaches; yet, whenever a file is uploaded in a chat message, the file's public URL will also be included in that message. The attacker can then instruct Slack to unfurl the public URL to obtain a direct-downloadable link. Therefore, the attacker can access files by reading all the messages in the user's joined channels.

Message Extraction Attack via Pins, Stars, or Reactions

We demonstrate another message extraction attack exploiting the incompleteness of resource ownership tracking in Slack. This time we leverage the productivity feature of pinning and starring messages (that add them to a user's saved message list) and the convenience feature of adding emoji reactions to messages. The attack builds upon the same message ID guessing technique from the prior attack.

To pin, star, or react to a message, the app needs to present the message ID and the ID of the message's channel to the corresponding Slack API, with the `pins:`, `stars:`, or `reactions:write` user token scope respectively. However, the read counterpart of these scopes (`pins:`, `stars:`, or `reactions:read`) does more than permit the app to view the IDs of the pinned, starred, and reacted messages; they also allow the app to view the *contents* of these messages. Therefore, after a valid channel ID and message ID is obtained, the app with both read and write scopes can either pin, star, or react to the message, effectively allowing itself to read the given message. As we have seen in the prior attack, an app without permission to read a user's private channel message is still able to acquire the channel ID and message IDs of that channel's messages. Hence, a malicious app can repeatedly pin, star, or react to these messages and read through all messages in the channel. We note that the app can also undo these operations using the corresponding write scope again to prevent the user from spotting any suspicious activity. With this attack, the malicious app can read all the messages that the user has access to, using only these seemingly harmless operations.

Analysis of Root Cause and Potential Prevalence

In both message extraction attacks, the malicious app obtains the ability to read any messages that the user has access to, with only some irrelevant

permission scopes. We consider this behavior as a violation of the user's privacy expectations. When a user grants the `im:history` scope to an app, there is no description in the authorization prompt that suggests the app can read private channels[6]. In addition, it puts the privacy of other users in these channels at risk — the messages they posted may suddenly become accessible to an app that they never authorized. Even worse, they have no way of knowing the leakage, since all it takes is for one user to install the app, an action that is hardly perceptible to them (Section 2.2), while the app itself is never a member of the channel.

An adversarial admin can use these attacks to monitor chats in private channels they are not invited to by forcing everyone to install their malicious app that disguises itself as an innocent management app.

Such privacy violation in the first attack is a failure of not enforcing *complete mediation*, which results from the improper tracking of resource provenance in Slack. Take Fig. 2.7 for example: when Slack finds a link to m_2 in m_1, it blindly appends the content of m_2 as m_1's attachments, without tracking where m_2 originates from. As such, any entity that can read m_1 can also read m_2, whereas these two messages have different provenances and should be checked against two separate permissions. The second attack can also be mitigated if Slack tracks and checks who performed the operation. While Slack needs to allow apps to read the content of pinned, starred, or emoji-reacted messages for functionality purposes, this rule should not apply if the app trying to read the message is the one who performed the operation (since it does not make sense for an app to pin a message it does not already know).

Potential Prevalence. Out of all 1,640 apps (66.7%) that do not request explicit scopes to read private channels (i.e., `groups:history`), we only counted 11 apps with the necessary permissions to extract messages via

[6] Accessing private channel messages with only `im:history` will cause Slack API to return an `missing_scope` error and a message saying that `groups:history` is needed.

pins, stars, reactions, or link unfurls.

2.7 Potential Countermeasures

We discuss countermeasures for the attacks we previously discussed. We note that these countermeasures are point fixes for the BCP permission model as it currently exists. The attack classes we've identified exist because the BCP permission model violates classic security principles. As such, even with these countermeasures, we cannot guarantee that all future issues will be prevented. We characterize each countermeasure from three perspectives: which design issues it attempts to solve, how much it helps mitigate the attacks, and what the cost or trade-off is.

Finer-grained Scopes

The BCPs we examined define several coarse-grained scopes that manage multiple resources of different types. For example, Slack's `chat:write` user scope allows an app to send messages to *any* target with the identity of the authorizing user. The Microsoft Teams Graph API `Chat.ReadWrite` scope grants a Microsoft Teams app similar permissions. Therefore, even if the app's functionality only requires sending messages to human users, it needs to acquire one of these broad scopes, which inevitably comes with the permission to send messages to apps and thus the ability to perform impersonation attacks on other apps. These scopes are coarse-grained as they allow an app to send messages to *separate* targets (app and non-app). BCPs can break down these scopes into two separate scopes: one that allows sending messages to non-app targets, and another that allows messages to app targets. However, this countermeasure cannot handle the attacks exploiting scopes that do not have finer-grained concepts (such as command hijacking).

Stricter Runtime Policy Checks

Stricter runtime checks can help address the message extraction attacks found in Slack. Specifically, Slack first needs to fix its coarse-grained modeling of the message resources by decoupling the unfurled content from the message and treating it as a separate type of resource. Slack also needs to track the origin of the unfurled content, for example, whether it is a message from another channel or a file shared with the user. Then, whenever an app requests to read a message, Slack should enforce an additional dynamic condition check to examine whether the provided token has the correct privilege to access the origin of the unfurled content. If not, only the message should be returned to the app, but not the appended unfurled content.

For the attack via pins, stars, or reactions, we present two options. The first is that when an app wants to read the pinned or starred messages, Slack should send the message content only if the app has the privilege to read the original message; otherwise, only the message ID is returned. However, this may inversely encourage malicious apps to request more privileges to maintain their original functionality. The second is for the BCP to consider the entity that issued the pin, star, or react operation. For example, an app can only read the content of a pinned/starred/reacted message if the pinning/starring/reacting is done by a human user or a different app; if it is done by the requesting app itself, then the BCP only returns the message ID. The tracking should occur even when a user has delegated control of their account to an app. When an app performs actions on behalf of a user, those actions should still be tracked as having been taken by an app. This should not hurt any benign app's functionality because if a message is pinned, starred, or reacted on by a benign app, it is reasonable to assume that the app should already know the message's content.

However, this countermeasure does not apply to situations where it is

difficult for an app or Slack to determine whether an action is malicious or user-intended. In Section 2.4, we demonstrated various legitimate scenarios in which users indeed want apps to perform actions on their behalf.

Indicate Identity of Action Issuer

To counter delegation attacks, the victim app should be able to determine if a received event comes from a human or an impersonated user and thus choose whether to respond or not. Thus, BCPs should indicate the identity of the action issuer (i.e., whether a real or delegated user performed the action) and therefore allow for identity checks on the victim app's side. Slack has provided this information for a few actions, such as posting messages but ignored it for other actions such as reacting to a message, which might also lead to exploits. However, as mentioned earlier, in some cases, even if the app knows the action is coming from another app, it is hard to tell whether the intent of the action is malicious or not.

Explicit User Confirmation

The final countermeasure is to request confirmation from users. From the perspective of victim users, all attacks stem from the fact that either victim apps or the BCPs automatically reacted to malicious events (in an unwanted way). Therefore, before accessing sensitive data, both the apps and the BCP should prompt the user for confirmation. For example, they can create a consent popup UI that involves clicking a button. Based on the current design of Microsoft Teams and Slack, only human users can perform such actions, making it hard to forge UI actions. This will prevent both delegation and message extraction attacks.

To resolve namespace collision attacks, BCPs should actively check for namespace collisions when apps are being installed. For example,

Slack should detect when an app attempts to register a command with the same name as a command already registered in the workspace, and Microsoft Teams should detect when an app has the same name as another app already installed in the workspace. We outline three solutions that BCPs may adopt. First, they can refuse to install the new app whose command would conflict with an existing one. However, this robs BCPs of functionality and unfairly penalizes apps installed later. Second, they can permit installation but require the user to make a selection whenever a namespace collision arises during use, but this requires the user to pay attention at all times. Third, after detecting a collision, they can provide an alias mechanism where users can change the conflicting names. In conclusion, runtime user confirmation can mitigate namespace collision attacks, but at the expense of productivity and user convenience.

2.8 Related Work

To the best of our knowledge, this is the first work to analyze the security and privacy of third-party apps in business communication platforms. However, considerable work has been done in other types of app platforms that share varying degrees of similarities with BCPs.

Social networks. Facebook and other social network platforms allow third-party applications that offer users additional functionality and services but generally at the cost of user privacy [74, 167]. These apps are similar to BCP apps in terms of pure server-side implementations and all-or-nothing permission, but they are installed in a single-user home space, whereas BCP apps are in a multi-user workspace. Symeonidis et al. show Facebook apps lead to collateral information collection [181], where they can collect not only data of the users who install them but also of their friends. This is akin to our findings of BCP apps; however, BCP apps can also actively affect other users' actions, such as through

interaction hijacking. On the other hand, several studies propose different access control schemes for apps in social networks [53,83,174,175,182,188]. While these solutions aim to solve the problem of coarse-grained permissions, they usually require the social network provider to host some part of the application codes, which does not suit the current communication framework of BCP apps.

Voice assistants. Amazon Alexa, a voice assistant often built into smart home devices, allows users to install third-party apps called skills. Similar to BCP apps, Alexa skills often appear in the form of chatbots; however the primary way of interacting with Alexa skills is through voice commands. Studies have shown that Alexa skills can be easily squatted to enable phishing attacks [134,207], similar to how Slack's commands can be hijacked. However, skill squatting relies on the inherent ambiguity of voices, whereas we exploit the namespace collisions of commands. In an orthogonal direction, many works try to measure the privacy practices of current Alexa skills and find that many skills do not honor their privacy policy and request overprivileged access [44,113,141,178].

Android. Many studies have analyzed the security and privacy of Android apps. The closest related attacks to this work are the confused deputy and collusion attacks [68,96,145,147,171]. Just as in BCPs, the app-to-app communications in Android can be used with malicious intent; however, they usually aim to achieve privilege escalation to access more user data instead of attacking users' accounts in other services. In addition, the problem of coarse-grained permission scopes is also found in Android, granting apps powerful capabilities that can be used to exploit various vulnerabilities [125]. Meanwhile, defenses proposed for Android apps usually require static or dynamic analysis [99,104,112,192,196], making them incompatible with BCP apps, which have no client-side codes.

Other OAuth-based systems. Studies have shown that overprivileged

attacks are a common issue in OAuth-based systems [71, 73, 101, 115, 126]. In addition, despite its wide adoption, OAuth is usually poorly designed and implemented by developers [76, 179, 189]. BCPs use coarse-grained scopes for certain operations and couple them with separate runtime policy checks that we have shown to be incomplete.

2.9 Limitations

For ethical reasons, we did not publish our attack apps to the Slack app directory or Microsoft Teams app store, and thus cannot comment on their vetting processes. However, we did analyze their security guidelines [35, 40] for publishing apps and found no obvious restrictions that would fundamentally prevent the attacks described in this work. These attacks rely on abusing permissions acquired for benign purposes, causing the information-limited vetting to be ineffective. BCPs do, however, prohibit two apps from sharing the same name, making it harder for a published app to mimic the appearance of another app; but as we noted in Section 2.5, a Slack app can circumvent this restriction by requesting the `chat:write.customize` permission scope, which allows the app the send messages using customized name and icon, avoiding the need to modify the app's own name and icon declared in the manifest.

2.10 Summary

We performed an experimental security analysis of the app model of two popular BCPs: Slack and Microsoft Teams. Our methodology was to study each BCP-facilitated interaction method between apps and users. We found that these BCPs violate two standard security principles: least access and complete mediation. We created proof-of-concept attacks that exploit these violations to (1) impersonate users and trick victim apps into

performing unwanted actions; (2) hijack commands; (3) steal messages from private channels without appropriate permissions. Our discussion of countermeasures indicates that while point fixes for these attacks can be deployed at the cost of BCP usability, preventing further issues requires redesigning the BCP app access control model.

3 eTAP: Protecting Data Privacy and Integrity in Trigger-Action Platforms

From this chapter, we begin to study the second category of threats in online integration platforms, which is their extensive access to user data from connected third-party services. Here we first focus on trigger-action platforms, which is a popular type of integration platforms with plethora of third-party connections, due to the generalizability of these platforms' trigger-action style computational paradigm. We introduce eTAP, a privacy-enhancing protocol that provides no plaintext access to the platforms.

3.1 Introduction

Trigger-action platforms (TAPs), such as IFTTT [26], Zapier [32], and Microsoft Power Automate [20] are web-based systems that enable users to stitch together their cyber-physical and digital resources (e.g., IoT devices, GMail, Instagram, Slack) to achieve useful automation. TAPs provide a simple *trigger-compute-action* paradigm and an easy-to-use interface to program automation rules.

For example, using their smartphone, a user can setup a rule that checks if an email contains the word "confidential" and, if so, sends an SMS with the subject line and the sender's address to a pre-specified number (Fig. 3.1). Instead of an SMS, the rule could also blink a smart light whenever a matching email arrives. To execute this rule on a TAP, when an email arrives (*trigger*), the mail service (*trigger service*) sends the email to the TAP that runs the string search (*computation*), which then contacts an SMS gateway or a smart bulb service (*action service*) with required information to perform the *action*. We refer to the combination of trigger/action services and the TAP as a trigger-action system — a key ingredient for fulfilling the promise of the IoT [146]. They provide a layer

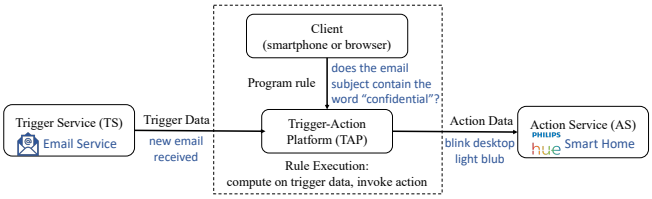


Figure 3.1: Overview of current trigger-action systems. The dataflow for the example rule is illustrated in blue color: "IF I receive an email containing the word 'confidential', THEN blink my desktop smart light."

of abstraction that enables trigger and action services to develop APIs independently without worrying about compatibility with each other.

These benefits unfortunately come at the high price of private data disclosure to the TAPs. Even the simple rule discussed above reveals the user's private emails to the TAP. As the TAP is the center of communication between triggers and actions, it can launch *person-in-the-middle* attacks by invisibly collecting private information on all of its users, similar to what has already been happening on centralized ride-hailing platforms [114, 140]. Due to the highly compatible nature of TAPs, this data includes location, voice commands, fitness data, pictures, files, etc. [120] and is limited only by the variety of online services of users (e.g., IFTTT supports 600 services [118]). Commercial TAPs do not provide any technical protections for user data. For example, IFTTT's terms of use explicitly state that they collect personal data from third parties, and may pass it to other third parties, partners, or any company that might acquire IFTTT [120].

Furthermore, because TAPs are widely-used centralized web services (e.g., IFTTT has more than 20 million users [122]), they are attractive targets for attackers. Breaches of cloud services are commonplace [18, 160, 165, 193]. Attackers sometimes even have continued access to the compromised service for days, and even weeks before getting detected [86, 87, 159]. A similar breach will have disastrous consequences for TAP users. Such privacy risks might discourage users as well as trigger/action

services from using TAPs. Indeed GMail, due to security and privacy concerns, pulled back some of its APIs from IFTTT [163].

In this chapter, we introduce eTAP, an encrypted trigger-action platform that executes user rules without accessing the underlying user data in plaintext. Thus, eTAP provides confidentiality even when the attacker fully controls the TAP. Although this problem fits in the general framework of secure function evaluation (SFE) [199,200], building a functional and secure trigger-action platform with good performance requires overcoming several challenges.

First, we desire confidentiality of user's data and authenticity of computation when the TAP is compromised and acts maliciously. While there are protocols for SFE that provide security even if some parties act maliciously [105, 142], these constructions are not yet practical [142, 166]. Second, using off-the-shelf protocols for SFE will require invasive changes to the architecture of trigger-action systems that break the independence between trigger and action services, making them less useful. Third, running arbitrary computations on the TAP using SFE will be inefficient.

We leverage the unique structure and threat model of trigger-action systems to overcome these challenges. At a high-level, we create a trusted generator of garbled circuits (GCs). This allows eTAP to use semi-honest implementations of SFE coupled with a few efficient extensions, which we contribute with security proofs, to achieve security against a fully malicious circuit evaluator.

In our setting, the user's smartphone, a standard component in TAP design, plays the role of a trusted circuit generator that periodically generates and transmits garbled circuits to the untrusted TAP. The trigger service garbles sensitive data when it is available and calls the TAP, which then executes the circuit and contacts the action service with the (garbled) results. The action service performs security checks and then executes the action. We assume that the user's phone is fully trusted, while TAP

is malicious. An attacker interested in compromising a large number of users is more likely to try compromising the TAP than the user's phone. To maintain the same level of trust as current TAPs provide, we treat the trigger and action services as semi-honest — they follow the protocol but can be inquisitive — and they should not learn any new private information that they do not learn in the current setting.

To overcome the challenge concerning the efficiency of arbitrary computations, we perform an analysis of the types of computations in popular commercial trigger-action platforms. We show that the computations supported by TAPs are stateless and use Boolean, arithmetic, or string operations. Most GC libraries support Boolean and arithmetic operations natively, but none support string operations out of the box. Existing work contributes oblivious deterministic finite automata that can match regular expressions [151]. However, it does not support substring extraction and replacements — a common operation in trigger-action systems. We therefore introduce a novel approach to efficiently encode a subset of fixed-length string operations as Boolean circuits. We then use the standard GC approach to evaluate them securely on the TAP. Our approach also has the advantage of unifying all the formal security properties of eTAP rather than having a separate set of proofs for string operations. eTAP can compute 93.4% of all rules published on Zapier that require computation and 100% of the 500 most-used rules on IFTTT. (Of course, eTAP supports all rules that do not require any computation.)

We formally prove the security of eTAP in the presence of a malicious TAP (Section 3.6). We show that the malicious TAP can execute user rules without learning the private data or tampering with the result of computation. eTAP also provides mutual secrecy between the trigger and action services.

eTAP is a clean-slate approach to building trigger-action systems and lays a foundation for securing the data they handle. However, it does

require some changes to current systems. First, the trigger/action services need to understand our protocols. We provide simple shims that they can use to upgrade their functionality while maintaining their independence and RESTful nature. Second, the user's client device takes on a more prominent role because it generates garbled circuits. As efficient circuits cannot be reused in general, the client has to periodically generate and transmit these circuits to the TAP. We estimate that this process has a modest impact: the trusted client is expected to transfer 61.7 MB of data per day for an average user. This is equivalent to the data consumed by uploading a one-minute of Full-HD video.

3.2 Background

We discuss background information on trigger-action systems and the cryptographic primitives that we use.

Trigger-Action Systems

Trigger-action systems allow stitching together disparate online services using a trigger-compute-action paradigm to automate different tasks. There are three main components of the system: trigger services (TSs), action services (ASs), and a trigger-action platform (TAP). We also explicitly mention another computing component: the user's client device that they use to interface with the trigger-action system. Fig. 3.1 shows the interactions between different components.

Trigger and action services are online services for IoT or web apps. There are a plethora of such services such as Instagram, Slack, GMail, Amazon Alexa, Samsung SmartThings, and many others. These services rely on REST APIs to send and receive data, and each service may support several APIs to provide different functionalities. They typically support the OAuth protocol [158], which is used to delegate authorization. With

OAuth tokens, a third party, such as a TAP, can access APIs and execute trigger-compute-action rules.

Commercial TAPs are compatible with hundreds of trigger and action services, allowing each trigger or action service to focus on building their own REST APIs without worrying about compatibility with each other. Third-parties own a large majority of these services that integrate with IFTTT (e.g., LG, Samsung, Google).[1]

Additionally, modern TAPs also allow performing non-trivial computation over the trigger data. The ability to modify the trigger data provides great flexibility for TAPs to achieve compatibility between trigger and action services (e.g., two calendar apps that use different date formats). The TAP also uses operations to decide whether or not it should send a message to the action service (e.g., does the email contain the word "confidential"). TAPs serve as a computation and communication hub. Zapier has explicitly supported computation on trigger data from the very beginning [24, 25]. IFTTT has recently started to expose its computing interface to end-users [121]. Thus, trigger-action systems are evolving to be *trigger-compute-action* systems. We use these two terms interchangeably throughout the chapter.

Users interface with trigger-action system through a client device, typically a smartphone. The user programs rules by selecting a trigger service, then specifying a computation on that data using a library of functions, and finally selecting an action to be run on the action service. As noted before, the user also authorizes the TAP to access their online services using the client device.

Privacy and authenticity risks in current TAPs. Commercial TAPs operate on sensitive trigger data of millions of users, making them an attractive target for attackers. If the TAP is compromised, the attacker gains the

[1] As of Aug 2020, 417 out of 522 services on IFTTT are third-party that require a user to login and authorize access to IFTTT.

privilege of the TAP — unfettered access to user data and resources. The types of data are limited only by the set of rules that users create and the end-point services that the TAP supports. Commercial systems like IFTTT support approximately 600 services currently [118]. The sensitive information from these services can be emails (our earlier example), data files, health information, voice commands, images, etc.

Fernandes et al. [102] first noted this problem with TAPs, and discussed a more appropriate threat model where TAP can act maliciously. Under this model, they addressed a sub-problem: preventing a compromised TAP from misusing overprivileged OAuth tokens. Their work adds integrity to the rules, but it does not allow any computation over the trigger data.

By contrast, we target modern TAPs that allow computation over the trigger data. Beyond integrity, we also aim to protect the *privacy* of that data. Our work provides a way for TAPs to compute on sensitive data without seeing the plaintext, despite arbitrarily deviating from the protocol. We believe such privacy risks might be preventing trigger-action systems from achieving their true potential. Furthermore, we provide computational integrity as well, thus subsuming prior work [102].

Cryptographic Primitives

Symmetric-key encryption scheme. Let $\mathcal{E} = (\mathsf{K}, \mathsf{E}, \mathsf{D})$ be a semantically secure encryption scheme. The key generation function $\mathsf{K}(1^\kappa)$ generates a κ-bit uniformly random key k; the randomized encryption scheme E takes a message $x \in \mathcal{X}$ and the generated key k as input and outputs a cipher text $ct \leftarrow\!\!\$\ \mathsf{E}(k, x)$; and the deterministic decryption function takes a cipher text and the key k as input and outputs a message, $x \leftarrow \mathsf{D}(k, ct)$, or $\perp$ (if decryption fails).

We use an authenticated encryption scheme [63] that achieves the IND-

CCA security guarantee. This ensures both the privacy and authenticity of plaintext.

Garbled circuits (GCs). This is a cryptographic technique for secure function evaluation (SFE) [63, 201]. Following Bellare et al.'s [64] notations, a garbling scheme $\mathcal{G}$ is a tuple of four functions $\mathcal{G} = (\mathsf{Gb}, \mathsf{En}, \mathsf{De}, \mathsf{Ev})$. Let $f : \{0,1\}^n \to \{0,1\}^m$ denote the function to be evaluated securely. Here, Gb is a randomized *garbling function* that converts the function f (represented as a Boolean circuit) into a *garbled* circuit F. It also outputs encoding and decoding information e and d needed for encoding inputs and decoding the outputs. As such, $(F, e, d) \leftarrow_{\$} \mathsf{Gb}(1^\kappa, f)$, where κ is the security parameter. The *encoding function* (En) encodes an input $x \in \{0,1\}^n$ using the encoding information e, which is the set of labels corresponding to the value of each bit in x; $X \leftarrow \mathsf{En}(e, x)$. The *evaluation function* (Ev) enables evaluation of the garbled circuit F over the garbled input X to generate the garbled output $Y \leftarrow \mathsf{Ev}(F, X)$, which is the set of labels corresponding to the output wires. Finally, the *decoding function* (De) decodes the output of the evaluation $y \leftarrow \mathsf{De}(d, Y)$.

Garbling involves generating two random labels L_1^w and L_0^w for each of its wires, representing the true and false value for the wire w. A number of optimizations have been proposed to reduce the size of a garbled circuit. One of them is the free XOR technique [130], which requires all wire labels to follow the form $L_1^w = L_0^w \oplus e_r$, where e_r is a string randomly chosen by Gb. This allows XOR gates in the circuit to be computed with only the input wire labels.

Typically, GCs are used for 2-party secure function computations where two parties with their respective private inputs x_1 and x_2 run the protocol such that, no party learns more than $f(x_1, x_2)$ for a public function f. The protocol works as follows. First, one of the parties, called the *generator*, uses the garbling function to generate $(F, e, d) \leftarrow_{\$} \mathsf{Gb}(1^\kappa, f)$. Next, it encodes its input as $X_1 \leftarrow \mathsf{En}(x_1, e)$. The other party, called the *evaluator*,

receives F and X_1 and also retrieves $X_2 \leftarrow \mathsf{En}(e, x_2)$ — encoding of its private input x_2 — using an oblivious transfer (OT) [164] protocol with the generator. Following this, the evaluator runs the garbled circuit to obtain $Y \leftarrow \mathsf{Ev}(F, (X_1, X_2))$. Finally, either party can decode Y to obtain the final output $y \leftarrow \mathsf{De}(d, Y)$.

A secure garbling scheme provides the following security properties [64]: (a) *Message obliviousness.* Given (F, X), an adversary learns nothing about x or y (beyond what is known from f). (b) *Input privacy.* Given (F, X, d), an adversary learns nothing about x beyond what is known from y and f. (c) *Execution authenticity.* Given a garbled input X, it is hard to find Y' such that $Y' \neq \mathsf{Ev}(F, X)$ and $\mathsf{De}(d, Y') \neq \perp$.

We use these cryptographic primitives to design eTAP. In Section 3.3, we analyze existing TAPs to understand what functions eTAP must support. We give the detailed protocol in Section 3.5, with its security proven in Section 3.6.

3.3 Analysis of Current Trigger-Action Systems

We analyze two popular commercial TAPs, IFTTT [26] and Zapier [32] with the following goals in mind: (1) understand the sensitive data that TAPs compute on; (2) establish that although TAPs offer a variety of operations on data, they are not arbitrary and will fit well in a garbled circuit framework; and (3) derive an abstract TAP computational model that will help ensure our system supports realistic functionality.

Types of sensitive information. The current trigger-action system design gives the cloud-based TAP complete access to trigger data. To better characterize the types of sensitive trigger data accessible to TAPs, we analyzed the IFTTT dataset mentioned in [150], by mapping each of its 320,000 IFTTT rules to one of the three trigger sensitivity levels defined by Bastys et al. [60] — public, private, and time-sensitive. Private triggers contain

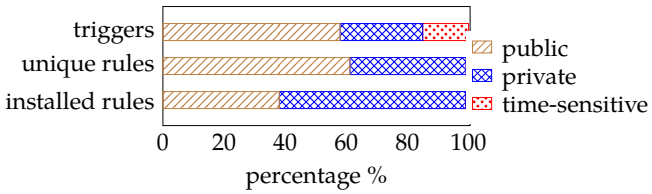


Figure 3.2: Breakdown of triggers, rules, and installed rules in IFTTT based on their sensitivity levels.

information like emails and calendar events, whereas public triggers contain information like news and weather reports. The time-sensitivity level means that private information exists in the availability of the trigger message. For example, considering the rule "IF I leave home, THEN turn off the WiFi," the TAP will learn whether the user leaves home depending on whether it receives a message from the trigger service. Fig. 3.2 shows a breakdown of sensitive trigger data according to how frequently they are used.

We observe that although a significant percentage (15%) of triggers and action APIs supported by IFTTT are time-sensitive, in reality, they are rarely used — only 0.8% of all available rules in IFTTT (or 0.9% of all installed rules) use a time-sensitive trigger. We also observe that, although there are fewer private triggers than public ones, private triggers are most frequently used — 61% of all installed rules contain a private trigger API. These APIs return private information like emails, messages, location traces, photos, sensitive files, medication lists, health information, etc. Thus, we design eTAP to protect the vast majority of private trigger information that people actually use in real-world rules. We do not currently provide confidentiality for time-sensitive information, but we outline possible approaches using standard techniques like cover traffic in Section 3.9.

Operations on trigger data. IFTTT allows users to express computation on trigger data using *filter code* — small snippets of TypeScript with some

restrictions (e.g., no I/O operations) [23]. Zapier rules contain two components: *filters* that compute a predicate on the trigger data, and *formatters* that modify the trigger data. Multiple filters and formatters can be chained together.

To understand the common operations in IFTTT, we again used the dataset of Mi et al. [150]. We selected the 500 most popular rules (based on user installation count) that are connected to private trigger APIs. Unfortunately, a challenge is that filter codes for IFTTT rules are not public. We therefore manually approximated the filter code for these rules by (1) estimating the functionality of each rule based on their title and description, (2) examining the corresponding trigger/action APIs, and (3) deducing the operations that are required to convert trigger fields to action fields.

We also crawled the Zapier website for one day in October 2019 and collected all the publicly available rules that require computations on trigger data [24, 25]. We collected a total of 378 rules and extracted the operations used in those rules.

The operations we found in IFTTT and Zapier are shown in Fig. 3.3. Current garbled circuit libraries support a majority of these operations natively. The main challenge is string operations, for which we contribute a novel technique to convert deterministic finite automata into Boolean circuits (Section 3.5).

Execution model of trigger-action systems. Based on our survey of IFTTT and Zapier, we derive an abstract model of these trigger-compute-action rules. During rule setup on the client, the user typically specifies two functions — a *predicate* f_1, and a *transformation* f_2. These functions take the trigger data and some additional user-provided constants as input. The predicate function f_1 tests the trigger data for a condition to determine whether TAP should contact AS. The output of f_1 is either true or false. The transformation function f_2 modifies the trigger data before sending the result to AS. Both f_1 and f_2 run inside the cloud-based TAP.

Type	Operation	Description
Bool	`x | a`	`x` OR `y`
	`x & a`	`x` AND `y`
	`! x`	NOT `x`
Num	`x < n`	Is `x` *less than* `n`?
	`x > n`	Is `x` *greater than* `n`?
	`x.mathop(n)`	Basic math ops. $(+, -, \times, \div)$
	`x.format()`	Format `x` into a string
Str	`x == s`	Does `x` *exactly match* the string `s`
	`x.contain(s)`	Does `x` *contain* the string `s`
	`x.startwith(s)`	Does `x` *start with* the string `s`
	`x.endwith(s)`	Does `x` *end with* the string `s`
	`x.split(d, i)`	Split `x` using delimiter string `d` and select the `i`-th substring
	`x.replace(s, t)`	Replace all occurrences of `s` in `x` with `t`
	`x.to_lowercase()`	Convert all characters in `x` to lowercase
	`x.truncate(n)`	Truncate `x` to size `n`
	`x.extract_phone()`	Extract the first phone number found in `x`
	`x.extract_email()`	Extract the first email address found in `x`
	`x.strip_html()`	Remove all HTML tags in `x`
	`x.html2markdown()`	Convert all HTML tags in `x` to Markdown
	`m.lookup(x)`	Look up the value for the key `x` in a user-provided map `m`
Any	`x == null`	Does `x` *exist*?
	`x.default(y)`	Set value of `x` to `y` if it does not exist

Figure 3.3: Operations used in top 500 IFTTT rules with private triggers and all Zapier's function-dependent rules.

Let $x \in \mathcal{X}$ be the part of the trigger data on which TAP performs some computation, and $y \in \mathcal{Y}$ be the action data TAP sends to AS, where $\mathcal{X}$ and $\mathcal{Y}$ are the domains of the trigger and action data, respectively. Both x and y can be data structures that contain multiple fields. We find that TAPs do not modify some fields of trigger data such as large media files, but only forward them to AS. We denote such trigger data as payload v. Let $c_1, c_2 \in \mathcal{C}$ be the two user-provided constants for the functions f_1 and f_2, where $\mathcal{C}$ is the domain of the constants. On receiving (x, v) from TS, TAP

executes

$$\text{“if } f_1(x, c_1) = \mathsf{true}\text{, then send } (f_2(x, c_2), v) \text{ to AS”}$$

For simplicity, we assume the domains of f_1 and f_2 to be the same. So, $f_1 : \mathcal{X} \times \mathcal{C} \to \{\mathsf{true}, \mathsf{false}\}$, and $f_2 : \mathcal{X} \times \mathcal{C} \to \mathcal{Y}$.

TAPs operate in two modes: (1) *polling mode*, where TAP contacts TS at a predefined frequency; (2) *push mode*, where TS sends a message to TAP when an event occurs. While our protocol will work with both models, we assume the push model in this work as it is more efficient in general.

Example rule. We show how our abstract model can instantiate our previous example rule: “IF I receive an email containing the word ‘confidential’, then send me an SMS.” The SMS should contain the address of the sender and the email’s subject. Assume that TAP provides an operation to search over strings, called `contain`. The user sets up a rule by choosing its email provider as the trigger service, that sends a copy of every new email to TAP. The action service is an SMS provider that sends SMS to a user-provided number. The user then specifies the `contain` function to check for the string $c_1 =$“confidential” on the email’s subject line, x. The transformation function f_2 creates the required data structure to send the SMS, for example, setting the recipient address as the user-provided phone number c_2 and the message body as the concatenation of the sender’s address and the subject.

3.4 Design Considerations for Providing Data Confidentiality in Trigger-Action Systems

Our goal is to protect the confidentiality of private data involved in trigger-action rules even if they are run on a malicious cloud-based TAP. In this section, we discuss our threat model, define our security and functionality goals, and explore the design space.

Threat Model and Functionality Goals

Fernandes et al. [102] first noted the security and privacy issues of a compromised TAP and the related attacker motivations. We adopt the same attacker model — TAP is *malicious*. Specifically, the attacker: (1) can monitor communications between TAP and the trigger/action services; (2) can arbitrarily deviate from the communication protocol by manipulating, delaying, or dropping the messages; (3) can modify TAP's internal storage and code that includes manipulating and deleting garbled circuits; (4) knows API details of trigger and action services; and (5) knows the functions that are being evaluated on TAP. As we use cryptographic techniques for our security guarantees, we assume that the attacker is computationally bounded.

We assume that the end-point services (trigger and action services) like Samsung SmartThings, Google Calendar, etc. are *semi-honest* — they will follow the protocol as specified, but try to glean more information than what they are entitled to know. This is in line with the trust model used by current TAPs. Also, if they are compromised, then the attacker can achieve its goals of accessing and manipulating user data independently of the trigger-action system. We also assume that TAP is not colluding with TS or AS. As discussed in Section 3.2, third-parties own a large majority of trigger and action services and thus collusion with TAP is unlikely (for example, there is no incentive for LG or Google to collude with IFTTT to reduce the security of their users). Enforcement of the non-collusion condition can also be done via legal affidavits [108, 168] or techniques that involve using a trusted mediator who monitors the communications between the parties [46, 47].

Finally, we assume that the user trusts their client device. We observe that the attacker is motivated to compromise TAP because it will simultaneously be able to attack all users of the platform. An attack on the client

device is not scalable to all users easily, and therefore, is less attractive.

Security goals. Under this threat model, we want two security properties for a trigger-action system:

Privacy: Each party should not learn other parties' data in a trigger-action rule. Specifically, TAP should not learn the trigger data (x, v), user-provided constants (c_1, c_2), and results of the computation (beyond what they already know from the definitions of the functions); the trigger service (TS) should not learn the user-provided constants (c_1, c_2); the action service (AS) should not learn the trigger data x or user-provided constants (c_1, c_2) beyond what is revealed to it after rule execution. Additionally, AS should not learn the output of transformation function f_2 or payload v when the predicate function f_1 evaluates to false.

Integrity: The attacker should not be able to modify any computations on private trigger data without being detected by AS. That is to say, TAP should not be able to trick AS into acting on illegitimate action data, such as delayed, replayed, or tampered messages that are not the result of proper evaluation of the rule. AS only accepts valid messages $y = f_2(x, c_2)$, where x is sent by TS within the last τ seconds (a configurable parameter).

Security non-goals. Denial of service is outside our scope. A compromised TAP can indeed drop all messages it receives from TS and not transmit any message to AS. Metadata and side-channel attacks are also outside our scope. For example, even if messages are encrypted, the compromised TAP can observe the timing of messages that arrive from a trigger service or go to an action service. Coupled with semantic knowledge about the services, this might enable the attacker to determine the sensitive data in the rule even if it is encrypted. As discussed in Section 3.3, this involves time-sensitive rules which are less used frequently in practice. eTAP protects the vast majority of sensitive trigger data for which encryption achieves strong security properties. Section 3.9 outlines standard approaches to

protect metadata that we leave as future work.

Functionality goals. We want to achieve the security goals while respecting the following functionality goals: (*1*) *RESTful API for end-point services*. The end services should be able to design their APIs independently of each other, as they do currently. These APIs should be RESTful, have minimal computational overhead beyond running the API itself, and do not need to store data or state specific to different trigger-action rules. (*2*) *Maintain trigger-compute-action paradigm.* The design should run existing user-created rules without any changes and should maintain the key architectural aspects of current trigger-action systems. Notably, the rules should execute without requiring the client device to be online.

Design Space Exploration

We explore a few potential solutions occupying different points in the design space and discuss why they do not meet our functionality or security requirements.

Computation at the edges. The trigger service can run a user-supplied function over its private data, encrypt the result, and forward that to TAP. However, the trigger service has to support an execution infrastructure similar to AWS Lambda, significantly increasing the complexity and overhead of such services and exposing them to additional security risk due to executing third-party code. Furthermore, sensitive data in user-supplied constants (c_1, c_2) will be exposed in plaintext to the trigger service. For example, consider rule R7 from Fig. 3.8, which converts Slack mentions to Asana tasks (a project management tool). It requires users to provide a lookup table of project names. These are sensitive information that should not be revealed to Slack. Computation can also be moved to the action service, but the same issues exist there as well.

Secure hardware. It is possible to use hardware-based trusted execution

environments (TEEs) or hardware security modules (HSMs) for computing the trigger data on TAP, while preserving confidentiality [172, 206]. Yet besides requiring hardware changes to the TAP servers, current TEEs suffer from fundamental security design issues [77, 154, 186].

Homomorphic encryption of the trigger data. During rule setup, the client can specify a symmetric key between the trigger and action service. The trigger service encrypts its data using this key before sending it to TAP. This will provide trigger data confidentiality and allow the TAP to compute directly on the encrypted data. However, only specialized schemes like linear homomorphic encryption and "somewhat" homomorphic encryption are practical [155], thus limiting expressivity. For reference, TFHE [31], a state-of-the-art library for fully homomorphic encryption, takes 4.45 seconds to compute an addition circuit, which is 3 orders of magnitude slower than our system as evaluated in Section 3.7. Additionally, protection against a malicious TAP would require zero-knowledge proofs [110] of computation that would further reduce efficiency.

Off-the-shelf secure multi-party computation. Secure multi-party computation (SMC) protocols allow multiple distrusting parties to compute a function over their private inputs [199]. However, efficient off-the-shelf SMC protocols do not fit our threat model — TAP is malicious, or architectural requirements — needing TC, TS, AS, and TAP to participate in a multi-round protocol during rule execution. Therefore, we adopt a core primitive of SMCs — garbled circuits — and modify it to our setting.

Secret sharing based SMC. Secret sharing is an alternative to garbled circuits for doing SMC. However, secret sharing-based protocols require intensive multi-round communication (e.g., for evaluating multiplication gates). Additionally, in such protocols every party has to do an equal amount of work, which will require invasive architectural changes to TS and AS. This violates our functionality goal. Finally, the malicious versions

of these protocols are not efficient.

3.5 Design of Encrypted Trigger-Action Platform

In this section, we discuss eTAP's core protocols and analyze how we specialize garbled circuits to trigger-action systems. A high-level overview of eTAP is shown in Fig. 3.4, and the pseudocode is given in Fig. 3.5. Like a typical trigger-action system in Fig. 3.1, eTAP has four components: trusted client's device (TC), trigger service (TS), action service (AS), and a trigger-action platform (TAP). We describe below how our design modifies these four components while maintaining the trigger-compute-action paradigm.

Decentralized trust model. In the current trigger-action system design, users place all trust within a centralized cloud-based TAP. This design leaves open a large-scale security and privacy risk — a single compromise of the TAP will simultaneously compromise all users. To avoid this issue, eTAP borrows a design element from DTAP [102] and designates the user's client device (smartphone) as the root of trust. Each user *only* trusts their own smartphone and uses it to program trigger-compute-action rules. As the eTAP protocols are open-source, we envision a community of developers building client apps, much like we have apps for open protocols like SFTP, Telnet, etc. Thus, the eTAP cloud component and the client app are built and controlled by different entities. Therefore, the client app can still be trusted, even when the TAP is compromised. eTAP bootstraps its guarantees on top of this model. In eTAP, the trusted client (TC) is beyond just an interface — it stores some state (as we describe below) that is key to its operation.

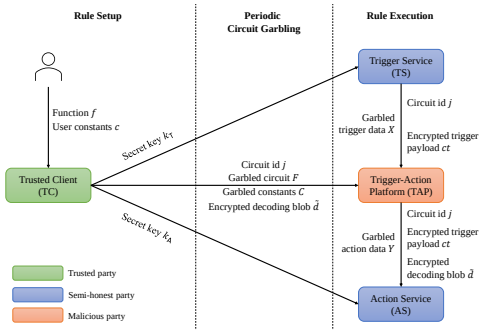


Figure 3.4: Overview of eTAP.

Rule Setup (occurs on trusted client)

Like in existing trigger action systems, the user can configure a trigger-compute-action rule on the trusted client app (TC) using its click-through interface. The user selects a trigger in a trigger service (TS), a predicate f_1, a data transformation over the trigger data f_2, and an action in an action service (AS). The user also specifies any constants c if required.

TC sends the rule descriptions to TAP and helps the TAP negotiate OAuth tokens with TS/AS required for running the rule. In eTAP, unlike existing TAPs, TC shares with TS and AS two uniformly-generated secret keys $\mathbf{k}_T$ and $\mathbf{k}_A$, upon successful authorizations. The key $\mathbf{k}_T$ and $\mathbf{k}_A$ are tied to the specific trigger and action API for this user in TS and AS[2]. If a prior rule has already been set up with the same trigger or action API, then the corresponding OAuth authorization can be skipped and TC will reuse the previously generated $\mathbf{k}_T$ or $\mathbf{k}_A$. Once the rule is setup, TS and AS store the shared key materials; TAP stores the OAuth tokens; and TC stores the rule (f_1, f_2), the keys $(\mathbf{k}_T, \mathbf{k}_A)$, and the constants (c_1, c_2) provided by the user for the rule.

[2]For better usability, current TAPs only acquire one OAuth token per service that can access all APIs in it [102]. eTAP can adapt to this model by exchanging a service-level key $\mathbf{k}_{TS}, \mathbf{k}_{AS}$, and derive the API-level keys $\mathbf{k}_T, \mathbf{k}_A$ from the hash value of $\mathbf{k}_{TS}, \mathbf{k}_{AS}$ and API URL, as required.

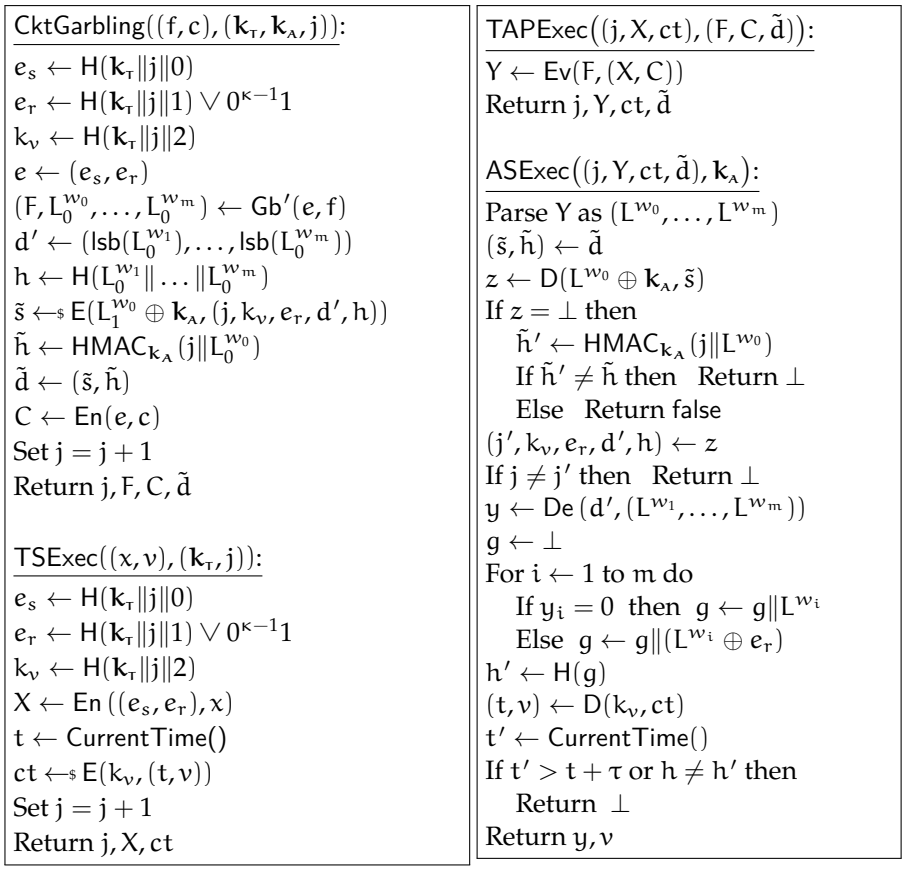


Figure 3.5: Circuit generation and rule execution protocols for *e*TAP. $L_1^{w_0}$ denotes the true label for the first output wire w_0, $L_1^{w_0} = L_0^{w_0} \oplus e_r$; τ is a threshold parameter used to ensure the freshness of a trigger. CktGarbling is run by TC asynchronous to the actual rule execution. The remaining three functions are run by TS, TAP, and AS during rule execution.

Circuit Garbling (periodic, occurs on trusted client)

Once the user creates a new rule, TC has to generate garbled circuit to enable secure evaluation of the functions on the (untrusted) TAP. TC generates garbled circuits corresponding to f_1 and f_2 and the associated encoding/decoding blobs. It uses the encoding blob to obtain the garbled labels for user-supplied constants. The decoding blob allows AS to decode

the garbled outputs and to decrypt the payload. To ensure TAP does not learn or tamper with the decoding blob, TC encrypts it using $\mathbf{k}_A$. TC sends the garbled circuits, encoded constants, and encrypted decoding blob to TAP. TC identifies each instance of the garbled circuit using a monotonically increasing counter j. The circuit id j is initialized to zero if this is the first rule where the user uses the connected trigger API; otherwise, TC queries TAP for the circuit id that the connected trigger API is currently using. As garbled circuits cannot be reused, TC periodically repeats the above process.

Although TC needs to transmit the garbled circuits and related information prior to rule execution (Fig. 3.4), we design eTAP such that TC does not need to be online during execution. TC generates and transmits GCs in batches at times when the smartphone is not being used (e.g., when charging at night). Our evaluation (Section 3.7) demonstrates that transmitting sufficiently many garbled circuits for a day generally takes less bandwidth than backing up a 1-minute Full HD video to a cloud drive. This achieves our design principle of keeping the client device offline during rule execution.

Note that in our setting, the generator of the garbled circuit is the smartphone client — a trusted entity. This is a key insight and design element that is possible due to the nature of our setting. This allows eTAP to use efficient semi-honest implementations of garbled circuits and achieve security in the presence of a malicious TAP.

Cryptographic Details. Without loss of generality, we assume $f_1 : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}$ and $f_2 : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^m$. For notational simplicity, we denote $f : \{0,1\}^n \times \{0,1\}^{2n} \rightarrow \{0,1\}^{m+1}$, such that $f(x, c) = f_1(x, c_1) \| f_2(x, c_2), c = (c_1, c_2) \in \{0,1\}^{2n}$. Additionally, let $H : \{0,1\}^* \rightarrow \{0,1\}^\kappa$ denote a cryptographic hash function. The pseudocode of the circuit garbling is given by the CktGarbling function in Fig. 3.5.

Encoding blob. The encoding blob contains the information required

to encode the trigger data and encrypt the trigger payload. It can be derived from the key $\mathbf{k}_r$ and the garbled circuit id j. TC generates three bitstrings $(e_s, e_r, k_v) \in \{0,1\}^{3\kappa}$, using the hash of $\mathbf{k}_r \| j$. The false labels of the input wires (as described below) are generated using a H with e_s as the random seed, and e_r is used as a global offset for the standard free-XOR optimization [130]. The least significant bit of e_r is set to 1 to enable the standard point-and-permute optimization [61, 204]. Thus $e = (e_s, e_r)$ constitutes the encoding information used for the garbling scheme's encoding function (En). The key k_v is used to protect the payload data v.

Garbled circuit. To generate the garbled circuit F for function f, the labels for every input wire w are computed as $L_0^w = H(e_s \| w)$ and $L_1^w = L_0^w \oplus e_r$ (assuming wire index w is a fixed-length bitstring). The rest of the computation (generating labels of the non-input wires and garbling gates) proceeds as per standard techniques with optimizations, such as row-reduction [156] or half-gate [204].

Encrypted decoding blob. The decoding blob consists of information necessary for AS to decode the labels of output wires (that correspond to the action data y) and to decrypt the payload. Let the output wires be $(w_0, w_1, \ldots, w_m)$, where w_0 corresponds to the output wire of f_1, and the remaining m wires correspond to those of f_2. Following standard practice [61], the decoding information d contains the least significant bits (lsb) of the false label of each output wire $(\mathsf{lsb}(L_0^{w_0}), \ldots, \mathsf{lsb}(L_0^{w_m}))$. In eTAP, decoding information is slightly modified. First, the first bit, $\mathsf{lsb}(L_0^{w_0})$, of d is dropped to create d'. Second, the hash of all the false labels of f_2's output wires $h \leftarrow H(L_0^{w_1} \| \ldots \| L_0^{w_m})$ is computed. Third, a decoding blob is created using d', h, the payload key k_v, the XOR offset e_r, and the current circuit id j. Next, the whole blob is encrypted using a symmetric-key encryption scheme E with a key derived from both $\mathbf{k}_a$ (the secret key shared with AS) and $L_1^{w_0}$ (the true label of f_1's output wire w_0) to obtain

$\tilde{s} \leftarrow\$ \mathsf{E}(L_1^{w_0} \oplus \mathbf{k}_a, (j, k_v, e_r, d', h))$. Additionally, an HMAC [132] of the false label of predicate f_1 is computed using $\mathbf{k}_a$ as $\tilde{h} \leftarrow \mathsf{HMAC}_{\mathbf{k}_a}(j \| L_0^{w_0})$. We use $\tilde{d}$ to denote the tuple $(\tilde{s}, \tilde{h})$. We explain the rationale behind these changes in Section 3.5.

Encoded user constants. Using the encoding information e, TC computes the labels for constants c as $C \leftarrow \mathsf{En}(e, c)$.

To accommodate the above customization, we derandomize the garbling function Gb to Gb' that takes an encoding information e as an input and returns the garbled circuit F, as well as the false labels of every output wire. TC sends $(j, F, C, \tilde{d})$ to TAP and increments the circuit id j by 1.

Rule Execution (occurs on TAP; does not involve TC)

When new trigger data is available for a trigger API, TS will garble the input data and encrypt any payload data, using the encoding blob it computes from $\mathbf{k}_r$ and circuit id j (which is initialized to 0 when the API is first called). It then transmits the ciphertexts to TAP, which will lookup any rules that are connected to the trigger API (and user) and run the associated garbled circuits. TAP finally transmits the output of the evaluation (garbled action data) and the encrypted decoding blob to the corresponding API in AS, which can decode to the plaintext result using $\mathbf{k}_a$ (Fig. 3.4).

TS and AS only perform simple encoding and decoding of data — fixed functionality independent of the trigger-action rule semantics, thus maintaining their RESTful nature. We believe that TS and AS are well-motivated to support these additional operations, in exchange for enhanced security. Indeed, current end-point services are concerned about the privacy of user data. For example, GMail recently removed their IFTTT triggers citing security and privacy concerns [163].

In our setting, the full evaluation of the garbled circuit is split between the untrusted TAP that executes the circuit to produce garbled output labels and the semi-honest AS that decodes the plaintext result from the

labels. This, in combination with the trusted generator, allows eTAP to efficiently achieve the execution authenticity property of GCs using a hash function (Section 3.5), even when TAP itself is malicious. We omit the standard OAuth steps that occur during execution, which the reader can refer to [30] for details.

Cryptographic Details. TS's operations in the rule execution phase is function TSExec in Fig. 3.5. TS recomputes the encoding information $e = (e_s, e_r)$ and the payload key k_v from $\mathbf{k}_r$ and j. It then encodes the trigger data x using the garbling scheme's encoding function, producing $X \leftarrow \mathsf{En}(e, x)$, and encrypts the payload v under a symmetric-key encryption scheme with the key k_v to compute $ct \leftarrow\!\!\$\ \mathsf{E}(k_v, (t, v))$ where t is the current timestamp. Finally, TS forwards the message (j, X, ct) to TAP and increments j by 1.

Upon receiving a trigger message (j, X, ct), TAP retrieves the corresponding garbled circuit F, garbled constants C, and the encrypted decoding blob $\tilde{d}$ using the trigger API and the circuit id j. Next, TAP evaluates F to obtain the garbled action data $Y \leftarrow \mathsf{Ev}(F, (X, C))$ and forwards the tuple $(j, Y, ct, \tilde{d})$ to AS. Function TAPExec in Fig. 3.5 depicts this process.

After receiving a message from TAP, AS decrypts $\tilde{d}$ to obtain the decoding information, which will succeed only when f_1 evaluates to true (i.e. $L^{w_0} = L_1^{w_0}$). If AS is able to decrypt the decoding blob, it uses (d', k_v) to obtain the final output $(f_2(x, c_2), v)$ in plaintext. AS would terminate if the message from TAP is malformed (i.e., hash of labels is inconsistent or decryption fails) or stale (i.e., trigger timestamp is old). The function ASExec in Fig. 3.5 depicts this process.

Rationale for Novel GC Protocol & Security Analysis

eTAP adopts a customized GC-based protocol tailored to the needs of trigger-action platforms. This protocol is novel in the following ways: (1)

By leveraging the structure and threat model of trigger-action systems, we can use efficient semi-honest implementations of GCs to obtain security against a malicious evaluator; (2) eTAP supports fixed-length string operations including matching, extraction, and replacement — common operations in trigger-action programs — using Boolean circuits only; (3) eTAP contributes an efficient technique to ensure authenticity on the evaluator's output (i.e., TAP) that requires only two hashes instead of the existing standard approach that requires hashes for true and false labels for every output wire.

Our setting has four parties: TC generates the garbled circuit via Gb' and then, both TC and TS use $\mathsf{En}(e, \cdot)$ to encode their respective inputs. On the other hand, TAP evaluates the garbled circuit using $\mathsf{Ev}(\mathsf{F}, \cdot)$ while AS decodes the plaintext output using $\mathsf{De}(\mathsf{d}', \cdot)$. Thus, TC and TS jointly act as the "generator", and TAP and AS jointly emulate the role of the "evaluator" of a two-party computation setting. The evaluators (TAP and AS) in our setting do not have any private input, therefore, eTAP does not require any oblivious transfers. Trust assumptions of the constituent parties of the generators and evaluators are asymmetric. Among the generators, TC is fully trusted and TS is semi-honest; among the evaluators, AS is semi-honest and TAP is fully malicious. Recall, TS and AS do not collude with TAP. (See Section 3.4 for the motivations behind these trust assumptions.)

Next, we highlight the changes we introduce in two-party GC protocol and the rationale behind those changes. We formally prove all security properties of our protocol in Section 3.6.

(1) TC generates the encoding information deterministically from the shared secret key $\mathbf{k}_r$ and the circuit id j, so that TS can also generate it without any communication with TC during rule execution. This achieves our design goal of ensuring that TC can be offline during rule execution. We note that this change does not violate the input privacy guarantees of the GC (see Thm. 3.1, 3.3, 3.4, and 3.5).

(**2**) Recall that the decoding blob (which contains information to decode garbled action data and to decrypt payload) is encrypted using the bit-wise XOR of $\mathbf{k}_A$ and $L_1^{w_0}$ as the key. Thus, TAP cannot learn the decoding blob (it does not have $\mathbf{k}_A$, Thm. 3.1). Only AS can successfully decrypt $\tilde{s}$ if it gets the true label of the output wire of f_1, $L_1^{w_0}$, from TAP; which can happen only when the predicate $f_1(x)$ evaluates to true. This meets our privacy requirement that AS should not learn $f_2(x, c_2)$ or v when $f_1(x, c_1) = \mathsf{false}$. We formally prove this in Thm. 3.4 and 3.5.

(**3**) eTAP ensures that the malicious TAP (evaluator) cannot tamper with the results of evaluation. To achieve this we add the following information to the decoding blob: $h = H(L_0^{w_1} \| \ldots \| L_0^{w_m})$, the XOR offset e_r, and $H(L_0^{w_0})$. Standard techniques to achieve this property require the hashes of both true and false labels for every output wire [204]. However, in eTAP, AS does not have access to the circuit F and the garbled inputs (X, C). This makes it safe to disclose e_r to AS (Thm. 3.4, 3.5). Thus, AS can compute $L_0^{w_1}, \ldots, L_0^{w_m}$ from the output labels (see ASExec in Fig. 3.5) and check whether TAP has returned forged labels for the output wires corresponding to f_2. The HMAC $\tilde{h}$ is used to ensure the authenticity of the first output wire corresponding to f_1, when it evaluates to false. Because of this structure, eTAP achieves efficient authenticity verification with two hash values (Thm. 3.2). This modification, combined with trusted generator, allows us to use efficient semi-honest implementations of GCs while achieving security against a malicious evaluator (TAP).

(**4**) We use a circuit id j to synchronize between different parties (TS, TAP, AS) so that they evaluate the correct circuit. Malicious TAP can observe the circuit id (in plaintext) and can tamper with it. eTAP ensures that the AS will always be able to catch a lying TAP, and will never act on an incorrect circuit id j. (See the proof in Section 3.6.) Metadata leaked due to learning j is outside the scope of this work (Security Non-goals in Section 3.4). We discuss a potential solution in Section 3.9.

Supporting TAP-Specific Operations with Garbled Circuits

While in theory any arbitrary function can be converted into Boolean circuits, and therefore can be computed using GCs, in practice they can be expensive. Via an analysis of existing real-world rules (Section 3.3), we found that they involve well-defined and relatively simple Boolean and arithmetic operations — these are well-studied and efficiently supported by existing GC libraries.

However, we also found that many rules use string operations, such as matching regular expressions and extracting or replacing substrings. The corresponding Boolean circuits of these operations, unless properly designed, will be inefficient to execute using GC [152]. eTAP computes these string operations by first translating regular expressions into deterministic finite automatons (DFA) and then applying a novel approach to convert DFA to Boolean circuits that can be efficiently evaluated using GC and can be easily extended for substring extraction and replacement. We next describe how eTAP utilizes this approach to perform regular expression matching. Please refer to [75] for details of how to convert a regular expression into a DFA.

Input and output representations. First, to avoid leaking the length of the string, every string field in the trigger data (and the action data) is padded to a fixed length bitstring. AS is responsible for removing the padding as necessary. The string is encoded into a fixed-length bitstring $\vec{x} = (x_1, \ldots, x_n)$ where $x_i \in \{0, 1\}$ before feeding into the encoding function En. Let the operation of the string be defined using the DFA Γ, which is represented as a five-tuple, $\Gamma = (\mathcal{S}, \Sigma, \delta, s_0, \mathcal{S}_F)$, where $\mathcal{S}$ is the set of states, Σ is the set of alphabets, s_0 is the initial state, and $\mathcal{S}_F$ is the set of final states. The transition function δ takes a state and an alphabet and returns the next state; therefore, $\delta : \mathcal{S} \times \Sigma \to \mathcal{S}$. Since every string is a bitstring, we have $\Sigma = \{0, 1\}$. Let $q = |\mathcal{S}|$ be the total number of states. Without loss of generality, we assume $\mathcal{S} = \mathbb{Z}_q = \{1, \ldots, q\}$.

Let $\vec{\delta}$ be the aggregated transition function that takes the entire string $\vec{x}$ as input and outputs the final state of the DFA,

$$\vec{\delta}(\vec{x}) = \delta(\ldots\delta(\delta(s_0, x_1), x_2), \ldots, x_n).$$

If $\vec{\delta}(\vec{x}) \in \mathcal{S}_F$, then $\vec{x}$ is accepted by the DFA, which means that the string matches the regular expression.

Converting DFAs into circuits. The main goal is to convert the transition function $t = \delta(s, x)$ into a Boolean circuit that uses as few AND and OR gates as possible, to take advantage of the standard free XOR optimization [130].

Since both the states s and t are integers between 1 and q, one can choose to represent each state using $\log_2 q$ bits and find the truth table for δ. However, the resulting circuit would be hard to construct and minimize automatically. Instead, we encode each state as a bit-vector of size q using one-hot encoding. We use S to denote the encoding of a state $s \in \mathcal{S}$, and S^i represents the i-the bit of S, where $S^i = 1$ if $i = s$ and 0 otherwise. We can observe that when $S^i = 1$ and $x = 0$, $T^j = 1$ if and only if $\delta(i, 0) = j$ holds; Similarly, when $S^i = 1$ and $x = 1$, $T^j = 1$ if and only if $\delta(i, 1) = j$. Therefore, the output of the DFA becomes

$$\vec{\Delta}(\vec{x}) = \Delta(\ldots\Delta(\Delta(S_0, x_1), x_2), \ldots, x_n),$$

where Δ is the transition function that operates on the one-hot encoded states.

To represent the transition function Δ as a Boolean circuit, we first define two sets for each state s, P_0^s and P_1^s, where $P_b^s = \{i \mid \delta(i, b) = j\}$ for $b \in \{0, 1\}$. It holds that $T^j = 1$ if and only if either $x = 1$ and $\exists i \in P_1^j, S^i = 1$,

or $x = 0$ and $\exists i \in P_0^j, S^i = 1$. That is to say, for $1 \leqslant j \leqslant q$,

$$\begin{aligned} T^j &= (x \wedge \bigvee_{i \in P_1^j} S^i) \vee (\neg x \wedge \bigvee_{i \in P_0^j} S^i) \\ &= (x \wedge \bigvee_{i \in P_1^j} S^i) \oplus (\neg x \wedge \bigvee_{i \in P_0^j} S^i). \end{aligned}$$

Because only one of the S^i will be 1 at any time, therefore the inner OR gates can also be replaced with XOR:

$$T^j = (x \wedge \bigoplus_{i \in P_1^j} S^i) \oplus (\neg x \wedge \bigoplus_{i \in P_0^j} S^i).$$

Note the above expression can be further simplified using the Boolean algebra property $(x \wedge a) \oplus (\neg x \wedge b) = ((a \oplus b) \wedge x) \oplus a$. Therefore, each bit in T requires at most one AND gate to compute. To run Γ over a string of length n, we need to apply transition function (Δ) n times, and thus the resulting circuit contains at most nq AND gates. Finally, to check if the final state is accepted by Γ, simply computing $\bigoplus_{j \in \mathcal{S}_F} S_n^j$ is sufficient.

We can observe that the size of the entire garbled circuit is $O(nq\kappa)$, on par with the communication cost of the state-of-the-art non-GC based customized approach [152]. However, being purely circuit-based, our approach allows functional conjugation with other operations and retains the same security properties of standard GC.

Extracting and Replacing Substrings with Garbled Circuits

We now discuss how eTAP extends the regular expression matching technique described above to extract and replace substrings.

Finding locations of matching substring. Given a regular expression pattern p, the goal is to find the starting and ending positions of the matching substrings.

Finding the ending positions can be achieved by applying the KMP

algorithm [128] on the pattern p to convert it into a DFA (denoted by Γ), so that Γ will output an accepting state at the end of each matching substring. For example, if the pattern is `ab`, we will rewrite it as `.*ab` and convert the new pattern into DFA. Then we use our matching protocol to run Γ on the input string $\vec{x}$. However, instead of only checking whether the final state S_n is an accepting state, we check every state $S_1, \ldots, S_n$ produced by Γ. We denote the resulting n-bit sequence as $e_1, \ldots, e_n$. If $e_i = 1$, it indicates that the i-th bit is the end of a matching substring.

Since a DFA can only report the end positions of matches end, we need another DFA to find the starting positions. We therefore compute a DFA Γ' on the reversed pattern p. If we run Γ' on the reversed input string, we get the beginning of the matching substring. Then, like the previous step, we run Γ' backward on $\vec{x}$ (by feeding from x_n to x_1) and check the type of every state to generate $b_n, \ldots, b_1$. If $b_i = 1$, it indicates that the i-th bit is the beginning of a matching substring.

Finally, we can find the locations of all matching substrings. That is, we need to compute another n-bit sequence $m_1, \ldots, m_n$ where $m_i = 1$ if and only if the i-th bit is part of a matching substring.

We can observe that $m_1 = b_1$ and for any i such that $2 \leqslant i \leqslant n$, m_i can be calculated as $m_i = b_i \vee (\neg e_{i-1} \wedge m_{i-1})$.

Extracting matching substring. To extract the matching substrings, we want to replace the characters in non-matching parts with the padding character (`0x00`). Therefore, the output string $\vec{y} = \{y_1, \ldots, y_n\}$ is computed by $y_i = m_i \wedge x_i$.

Replacing matching substring. In our dataset, all `replace(s,t)` functions are used with `t` set to empty string, so it is equivalent to removing the matching substring, and thus the output string $\vec{y} = \{y_1, \ldots, y_n\}$ is computed by $y_i = \neg m_i \wedge x_i$.

However, for completeness, we will describe a protocol for the general case scenario where $|t| > 0$, where `t` denotes the size of the string `t`. The

output string size will be $n \times \frac{|s|}{|t|}$ since the TAP should not know which substring is matched and replaced and should assume all substrings can be replaced. When $|s| \gg |t|$ the sizes of the resulting garbled circuits will be unbearably large. Therefore, we purpose an alternative design approach where the actual replacement is processed in the action service: we replace the first character of each matching substring with some placeholder character, say `0xff`, and the rest with the padding character `0x00`, so the action service can invoke the following functions to complete the replacement: `y.replace("0x00", ""); y.replace("0xff", t);` where `y` is the decoded output string. Note the first `replace()` is required regardless of our protocol, since it is needed for removing the padding from the input string.

We argue this approach does not break our security goal, revealing no additional trigger data that is not supposed to be revealed to the action service. If the replacement string t is considered sensitive the client can encrypt the replacement mapping with the 1 label of the output bit corresponding to $\bigvee_{i=1}^{n} m^i$, similar to how we protect d and k in Fig. 3.8.

Assuming an ASCII encoding and `0xff` as the placeholder character, we can compute the output string $\vec{y}$ using $y_i = s_{i-(i-1 \mod 8)} \vee (\neg m_i \wedge x_i)$, where the $i - (i - 1 \mod 8)$-th bit is the first bit of the character that i-th bit belongs.

Supported functions. By incorporating the above techniques, we can use garbled circuit to efficiently compute common arithmetic operations, string operations, and dictionary lookup, which cover all but three functions listed in Fig. 3.3. We sketch the implementation details for each supported function in Section A.2. Based on our analysis in Section 3.3, this set of operations enables eTAP to support 93.4% of the function-dependent rules published on Zapier and *all* of the 500 most popular rules on IFTTT.

It is possible to convert the remaining three unsupported functions (`format`, `strip_html`, and `html2markdown`) to Boolean circuits, as well, but

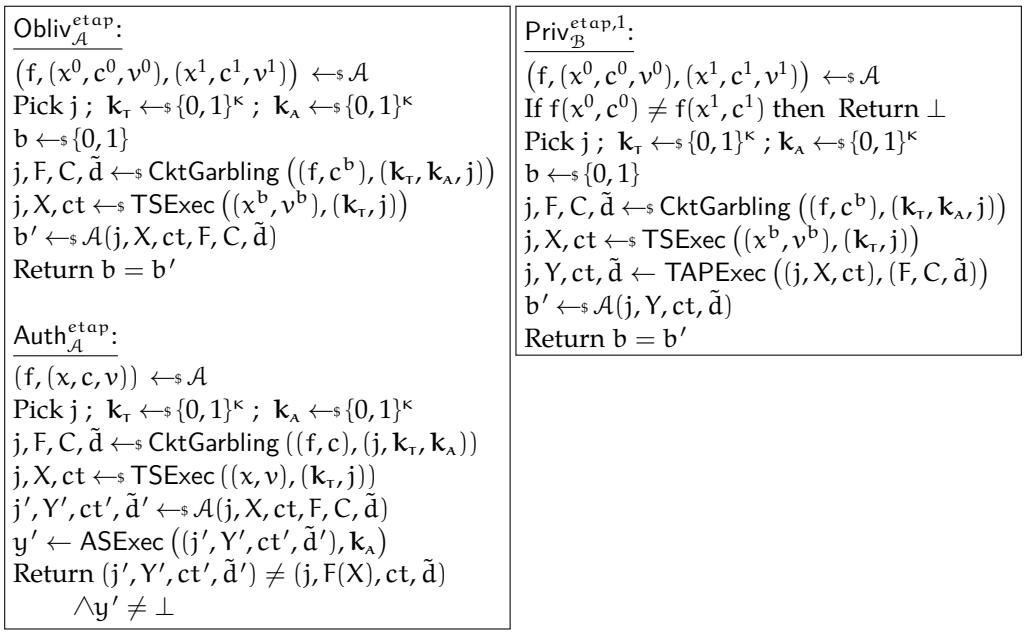


Figure 3.6: Security games for eTAP.

the resulting circuits will be very large (for example, we need to build a full-blown parser to find HTML tags) and inefficient to evaluate. These functions are only used for formatting and do not require any sensitive user input. Thus, it is safe to run them on AS or TS directly with minor modifications to their APIs.

3.6 Security Analysis of eTAP

In this section, we show that eTAP meets the security goals outlined in Section 3.4 by providing concrete security definitions and proofs. We assume the adversaries are probabilistic polynomial time (ppt) — they run in time polynomial in security parameter κ. The garbled circuit protocol $\mathcal{G}$ used in eTAP provides *output privacy*, *message obliviousness*, and *execution authenticity*. The encryption scheme $\mathcal{E}$ is IND-CCA secure. We model the hash function H as a random oracle [65]. Let $\mathsf{negl}(\cdot)$ to be a negligible

function.

We prove the security of each component of eTAP, namely TAP, TS, and AS, separately. The security games are defined in Fig. 3.6.

Security against malicious TAP. Following our threat model, we assume the TAP is compromised and *malicious*. The security definitions we expect from eTAP are as follows.

Obliviousness. We define the obliviousness property of eTAP by the security game $\mathsf{Obliv}_{\mathcal{A}}^{\mathrm{etap}}$ as shown in Fig. 3.6. Informally, $\mathcal{A}$ despite arbitrarily deviating from the protocol should not know anything about the user-provided constants c, the trigger data x, v, and the output of the function $y \leftarrow f(x)$.

Theorem 3.1 (TAP Obliviousness)**.** *For any ppt adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the* $\mathsf{Obliv}_{\mathcal{A}}^{\mathrm{etap}}$ *game is negligible.*

$$\Pr\left[\,\mathsf{Obliv}_{\mathcal{A}}^{\mathrm{etap}} = 1\,\right] \leqslant 1/2 + \mathsf{negl}(\kappa)\,,$$

Proof: The proof of this theorem follows directly from the *message obliviousness* security guarantee of garbled circuits $\mathcal{G}$ [204] and the semantic security of the encryption scheme $\mathcal{E}$. As such, the attacker learns nothing about (x, v, c) from (X, C, ct). First, note that the game $\mathsf{Obliv}_{\mathcal{A}}^{\mathrm{etap}}$ is equivalent to the game $obv.sim_{\mathcal{S}}$ [64] in [204]. Now, consider the simulator $\mathcal{S}$ as presented in Fig. 3 in [204]. In our setting, $\mathcal{S}$ is used by TC and TS to generate $(\hat{F}, \hat{X}, \hat{C})$ which is then used for the rest of the computation. Hence the obliviousness of (x, c) follows directly from the corresponding proof (game $obv.sim_{\mathcal{S}}$) presented in [204] assuming the random oracle model for H [65]. The indistinguishability of ct^b follows trivially from the semantic security guarantee of the encryption scheme, thereby concluding our proof.

We achieve security against a malicious TAP even with a GC implementation for the semi-honest model. Recall that the "generators" — the

trusted client (TC) and the trigger service (TS) — in eTAP are at least semi-honest. Hence, a valid garbled circuit for the correct function f is always generated (as TC is trusted), and all inputs are correctly encoded (since TS is semi-honest and the "evaluators" TAP and AS have no input). Thus, the only way a malicious TAP can compromise the security of eTAP is by forging an inauthentic output label or by replaying, delaying, or dropping a message. We discuss eTAP's resilience to such attacks next.

Authenticity. The security guarantee *authenticity* ensures that no ppt adversary can create a garbled output $Y' \neq Y$ such that AS acts on Y' (that is to say ASExec outputs anything but $\perp$ or false). The formal definition is given by the security game $\mathsf{Auth}_{\mathcal{A}}^{\mathsf{etap}}$ as shown in Fig. 3.6.

Theorem 3.2 (TAP authenticity)**.** *For any ppt adversary $\mathcal{A}$, the probability that $\mathcal{A}$ wins the game $\mathsf{Auth}_{\mathcal{A}}^{\mathsf{etap}}$ is negligible,*

$$\Pr\left[\,\mathsf{Auth}_{\mathcal{A}}^{\mathsf{etap}} = 1\,\right] \leqslant \mathsf{negl}(\kappa)\,.$$

Proof: The proof follows from the non-malleability guarantee (IND-CCA) of the encryption scheme $\mathcal{E}$, execution authenticity of $\mathcal{G}$ [204], and the collision resistance of the hash function H. For the rest of the proof, consider the simulator $\mathcal{S}$ in [204] which additionally returns $h = \mathsf{H}(L_0^{w_1} \| \ldots \| L_0^{w_m}), e_r$ and $L_0^{w_0}$. TC uses this additional information to generate the decoding blob $\tilde{d}$. Similarly, the function in De is changed to that of ASExec.

Case I - Authenticity of $y_1 = f_1(x, c)$.

Note that $\tilde{s}$ is encrypted under a key derived from $\mathbf{k}_A$ and $L_0^{w_0}$. Hence, from the semantic security of the encryption scheme, TAP does not have access to e_r since it does not know $\mathbf{k}_A$ by design. Thus, in case $y_1 = \mathsf{false}$, TAP has access only to the false label $L_0^{w_0}$ and thereby cannot cheat AS. On the other hand, if $y_1 = \mathsf{true}$, TAP can return some garbage value L' such that $\mathsf{D}(L' \oplus \mathbf{k}_A, \tilde{d}) = \perp$. However, AS can detect this with the help of the HMAC. Moreover, TAP cannot send any of the hitherto unseen HMACs because

it cannot obtain the output labels without access to the corresponding X (TS's trigger data).

Case II - Authenticity of $y_2 = f_2(x, c)$.

From the collision resistance of H, the only way TAP can cheat is by generating a label $L^{w_i}_{1-y_2[i-1]}$ for some wire $i \in [1, m]$ where $y_2[i]$ denotes the i-th bit of y_2. However, as discussed above, TAP cannot compute any other label other than the one obtained from $\mathsf{Ev}(F, X, C)$.

The rest of the proof follows an identical sequence of hybrids as the proof of Theorem 1 in [204] assuming the random oracle model for H.

Protection from altering the timing of rule execution. An adversary cannot forge a message that the AS will accept due to the strong authenticity guarantee of eTAP protocol. However, it can alter the execution time of a rule by deliberately dropping, delaying, or replaying messages. TAP can successfully drop a message without being detected by AS. However, this would fall under the denial-of-service attack which is beyond eTAP's scope (Section 3.4). eTAP also protects against replayed or delayed messages.

Every message from TS is timestamped as they are sent which AS can check before performing any action. Therefore, AS will reject a message — outputting $\perp$ — if the received message is delayed more than τ seconds (a parameter set by AS) since the time it was sent from TS. (See the function ASExec in Fig. 3.5.) We acknowledge that the TAP can replay any message for which $f_1(x, c) = \mathsf{false}$ without getting detected by the AS.

Nevertheless, this does not lead to any undesirable outcome in practice because in this case AS performs no action. Note that the above attack (replay of false labels) could have been prevented by keeping track of the last seen circuit id of each rule at AS. However, maintaining such state information would violate the RESTfulness of AS.

Tampering with circuit id j**.** The malicious TAP can modify the circuit id j — a unique identifier given to every instance of a garbled circuit for synchronization between TAP, TS, and AS — in whatever way they want to.

But eTAP ensures AS will always be able to detect any such modification and rejects the message from TAP (by outputting $\perp$). This is done by having TS include the circuit id j in the encrypted payload ct — that TAP cannot modify. AS verifies that value against the circuit id forwarded by TAP, and any mismatch results in execution termination. Though TAP cannot tamper with j without being detected, it could learn the popularity of certain rules by observing circuit id values (which are passed to TAP in plaintext to help find corresponding garbled circuit F to execute). We acknowledge that metadata attacks are a limitation in eTAP and we discuss a cover traffic approach to address them (Section 3.9).

Security Analysis of TS and AS. We assume TS and AS are honest but curious. We define security as follows.

Theorem 3.3 ($\mathsf{Priv}_{\mathrm{TS}}$)**.** *TS does not learn anything about the user constants* (c_1, c_2).

Proof Sketch. TS only receives from the client k_r and j, which it uses to compute the seed $e = (e_s, e_r)$. Thus, it can only learn the pairs of labels for all the input wires (including the ones for user constants) to the garbled circuit. TS, *by design*, does not have access to the client constants.

AS should not learn about the user constants and the trigger data beyond what is revealed from the output of the function f. Let $y_1 = f_1(x, c)$ and $y_2 = f_2(x, c)$. We also need to ensure that when the output of the predicate function $y_1 = \mathsf{false}$, AS does not learn the output of the function f_2 and the payload v. We formally state these properties, using the theorem below.

Theorem 3.4 ($\mathsf{Priv}^0_{\mathrm{AS}}$)**.** *If* $y_1 = \mathsf{false}$*, then AS learns nothing about* (x, c, v) *other than what is revealed from* $y_1 = \mathsf{false}$.

Proof: To know the value of y_2, AS needs access to the decoding table d' (from the *obliviousness* guarantee of garbled circuits in [204]). AS will be

able to do this only if it has access to $L_1^{w_0}$ (from the IND-CCA security of the encryption scheme). Note, $L_1^{w_0}$ is available to TAP, and subsequently to AS, only if $f_1(x) = \mathsf{true}$ [204]. In case TAP returns some garbage value other than $L_1^{w_0}$, the decryption still fails. Additionally, v is protected by the IND-CCA security of the encryption scheme.

Theorem 3.5 (Priv_{AS}^{1})**.** *If* $y_1 = \mathsf{true}$*, then for any ppt adversary* $\mathcal{B}$*, the probability that* $\mathcal{B}$ *wins the game* $\mathsf{Priv}_{\mathcal{B}}^{etap,1}$ *is only negligibly more than random guessing. That is,*

$$\Pr\left[\,\mathsf{Priv}_{\mathcal{B}}^{etap,1} = 1\,\right] \leqslant 1/2 + \mathsf{negl}(\kappa)\,.$$

Proof: The indistinguishability of ct^b follows from the semantic security of the encryption scheme. Now note that $\mathsf{Priv}_{\mathcal{B}}^{etap,1}$ is equivalent to $prv.sim_{\mathcal{S}})$ [64] in [204]. The rest of the proof is based on the proof for the corresponding game ($prv.sim_{\mathcal{S}}$) in [204]). In fact in our setting, the view of the $\mathcal{A}$ is a strict subset of that of the adversary presented in [204]. Specifically, our adversary $\mathcal{A}$ does not have access to the garbled inputs X^b, C^b and the garbled circuit F. Note that in the above game, a malicious TAP instead of outputting $(Y, ct) \leftarrow \mathsf{TAPExec}\,((X, ct), (F, C))$, could generate some arbitrary message. However, from the obliviousness property of garbled circuits (Thm. 3.1, we know that this message has to be completely oblivious of (F, X, c) and hence the privacy guarantee is upheld trivially.

Proposition 1 (TAP Input Indistinguishability)**.** *For any ppt adversary* $\mathcal{A}$ *with access to a circuit garbled with the scheme in* [204]*, the probability that* $\mathcal{A}$ *distinguishes between a valid garbled input and randomly generated input is negligibly more than random guessing.*

Proof Sketch. Following Fig. 2 in [204], it is clear that $\mathcal{A}$ cannot validate inputs to XOR gates. For AND gates, the fact that at most one valid label for each input wire is revealed to $\mathcal{A}$ and the correlated robustness of the

hash function ensures that $F = (T_G, T_E)$ does not reveal information about the valid inputs.

3.7 Evaluation of eTAP

We prototyped eTAP and showed that it is competitive in performance with TAPs that do not provide any data privacy. We implemented the garbled circuit protocols described in Section 3.5 using EMP toolkit [191], a C++ library for multi-party computation. We build on EMP toolkit's semi-honest 2PC protocol. We use state-of-the-art optimizations (including free XOR [130] and half gates [204]) for improving efficiency and bandwidth. The security parameter is $\kappa = 128$. For other cryptographic operations we use Cryptography.io [6]. We use SHAKE-128 (a member of SHA-3 family [157]) as a cryptographic hash function, and AES in CBC mode with HMAC using SHA-256 as a semantically secure, non-malleable, robust symmetric-key encryption scheme. To convert regular expressions into DFAs we use the library dk.brics.automaton [153]. For all experiments, we used `n1`-standard instances in Google Cloud Platform configured with 2 vCPUs, 7.5 GB memory, and 1 Gbps network connection.

Performance of Basic Operations

eTAP supports Boolean, (integer) arithmetic, and string operations (which is sufficient to run most of the rules in Zapier and IFTTT). To evaluate the performance of these basic operations, we picked a set of representative operations from Fig. 3.3. For Boolean, we chose the AND operation since our circuits only contain AND and XOR gates, and the XOR gate can be computed without any encryption costs [130]. For numeric data, we selected comparison and multiplication between two 32-bit integers. For string operations, we divided them into two categories: operations that need regular expressions (`contain`, `replace`, `split`, and `extract_phone`)

	Operation	Computation time (ms) Client	TS	TAP	AS	GC size (KB)	# DFA states
Bool	`x & y`	4.0	3.7	3.7	3.9	0.03	–
Num	`x > n`	4.0	3.9	3.8	3.8	0.96	–
	`x * n`	4.0	3.7	4.0	3.7	31	–
Str	`x == t`	4.0	3.7	4.0	3.8	25	–
	`m.lookup(x)`	4.2	3.6	4.1	3.8	31	–
	`x.split(d,0)`	5.7	3.7	5.3	4.1	78	16
	`x.contain(s)`	7.8	3.9	7.4	3.9	123	47
	`x.replace(s,"")`	10.7	3.8	10.5	4.6	278	40
	`x.extract_-phone()`	24.7	3.6	25.5	4.1	2191	108

Figure 3.7: Execution time of different basic operations at the client (TC), the trigger service (TS), the action service (AS), and the TAP. We record the size of the garbled circuit sent from TC to TAP and the number of states in the DFA if applicable.

and those that do not (`lookup` and ==). We set the input `x` as a 100-character (800 bit) string, except for `lookup`, where we set `x` to a 10-character string. In the function `m.lookup(x)`, we set `m` to be a key-value store with 10 entries, where each key and each value is 10-characters long. For `x.replace(s, "")` and `x.contain(s)`, we set the `s` to a 4-character string. For `x.split(d, 0)`, we set `d` to be a single character.

While measuring the costs for above basic garbled circuit operations, we do not consider the overhead of other components like payload encryption, as they are independent of the operation. Fig. 3.7 shows the time required for each operation.

The circuit generation (at TC) and circuit evaluation (at TAP) take roughly the same amount of time for each operation, which is expected because they require roughly similar operations. Most of the Boolean, arithmetic, and some string operations (such as string equality or lookup) execute in less than 4 ms on the TAP. Complex string operations are also fast (takes less than 25 ms) under some reasonably sized inputs. TS and AS can encode/decode inputs in less than 5 ms.

We record the size of the garbled circuit ($|F|$) for each operation in

#	Rule description	Functions performed	GC size (KB)	Data transfer (KB) →TAP	TAP→
R1	Share your Tweets (excluding replies) in Slack	`! x[Text].startwith("@")`	0.2	43	20
R2	Get Slack notifications for new Twitter followers with more than 5,000 followers	`x[FollowerCount] > 5000`	1.0	29	3
R3	Copy New Events from Google Calendar into iOS Calendar	`x[StartTime] - x[EndTime]`	1.0	33	32
R4	Blink your lights when you receive email from a specific address	`x[Sender] == c`	5.8	29	3
R5	Send SMS messages for new Shopify orders	`x[Phone] != null; x[Phone].replace(" ", "")`	9.0	27	3
R6	Add new inbound emails as contacts in Ontraport	`x[SenderName].split(" ", 0); x[SenderName].split(" ", 1)`	30.5	34	13
R7	Create Asana tasks when new Slack messages start with `$request`	`x[Text].startwith("$request"); x[Text].replace("$request"); c2.lookup(x[Channel])`	92.4	29	4
R8	Save new liked Tweets with links to Pocket	`x[Text].contain("http")`	173.4	43	20
R9	Send SMS reminders for upcoming Google Calendar events	`x[Description].extract_phone()`	4,668.9	51	28
R10	Upload new videos in Google Drive to YouTube	`x[Filename].endwith("...")`	12.1	32,133	32,108

Figure 3.8: Selected real-world rules for our experiments from both IFTTT and Zapier.

the second-to-last column of Fig. 3.7. The garbled circuit $\tilde{F}$ needs to be periodically transferred from the client to TAP and the size of the circuit changes significantly for different operations. Although for Boolean AND the circuit is only 31 bytes, the circuit size for a complex regular expression extraction, which is one of the most expensive operations we found, is quite large (2.2 MB). The size of garbled circuit increases with the number of states in the DFA and the length of the input string. The string replacement circuits (`replace`) are larger, about 2.25x, than their equivalent matching circuits (`contain`), even though the required DFA is larger for the latter operation. The lookup circuit is small (31 KB).

Performance of Running Complete Rules

Next, we measure the performance of eTAP on real-world rules. We first picked ten rules from the combined IFTTT and Zapier dataset we collected in Section 3.3. These rules handle sensitive data of different sizes and cover a wide variety of operations (as noted Fig. 3.3). We list the rules with simple descriptions in Fig. 3.8. The first eight rules (R1-R8) involve frequently used functions, while the last two rules represent two rare but extreme scenarios. R9 requires a rarely-used `extract_phone` function, which appears only three times in our dataset and requires a complex regular expression to be evaluated over a long text, thus making it the most expensive rule to compute in our dataset. R10 is connected to a trigger that might have a large payload (videos), so its performance is more dependent on network bandwidth and latency.

For comparison, we built a skeleton version of each service following the current TAP model, where only plaintext data is exchanged and computed, as a baseline. We refer to this as PlainTAP. We used Python library Flask for the cloud component of TAP, as well as two RESTful servers that mimic the APIs provided in current trigger and action services. Two US-west instances hosting TS and AS, and two US-central instances for hosting TAP and the (simulated) TC. The network latency between US-west and US-central is 39 ms.

Latency. The end-to-end execution latency measures the time between a trigger event (trigger data and payload are available to TS) and AS receiving plaintext output (Fig. 3.9). The latency, except R9 and R10, is below 260 ms. When compared to PlainTAP (Fig. 3.9, top), the execution latency for eTAP is 55% more on average. The majority of the latency overhead is due to the higher amount of data transfer in eTAP between TS and TAP (27-51 KB) and between TAP and AS (3-32 KB), which is nearly 128x more than what it would require in PlainTAP. We show the data transfer in the last two columns in Fig. 3.8. Given that TS, AS, and

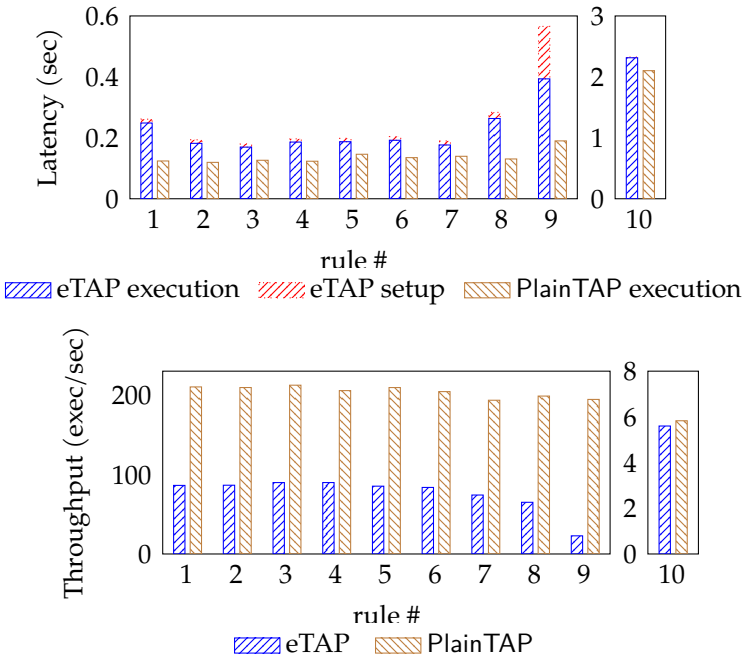


Figure 3.9: Latency (top) and throughput (bottom) for running each of the rules (X-axis) in eTAP and PlainTAP.

TAP are cloud-based services with high-bandwidth network links, the increase in data usage is reasonable. In addition, we list the time spent by TC to generate and upload a single circuit (as the red bar in Fig. 3.9, top). TC needs less than 12 ms to generate and transfer one circuit for most rules (except for R9, in which case it takes 172 ms). This metric represents the setup time for a new rule before it can be executed. In practice, TC can generate and upload circuits in bulk periodically at its convenience.

Throughput. We measured the throughput as the maximum number of executions per second by eTAP. We used Apache Bench [1] to compute the throughput, which simulates sending concurrent trigger messages to eTAP. We pre-computed the trigger labels to eliminate the bottleneck on TS. We gradually increased the concurrency level until the throughput saturated. We reported the maximum throughput of eTAP and PlainTAP

in Fig. 3.9 (bottom). eTAP is capable of executing 65-90 rules of type R1-R8 per second on a single server. Compared to PlainTAP, for all but one (R9) rule, eTAP provides around 41% throughput of PlainTAP. In the worst case, when executing R9, eTAP's throughput reduces to 11% of PlainTAP.

Large-scale Evaluation

To better characterize the performance of eTAP under realistic workloads, we performed a large-scale evaluation where we randomly sampled 100 rules from our combined IFTTT and Zapier dataset. Out of the 100 sampled rules, 55 require computations on the trigger data. For rules with no computations, we simply treated the trigger data as payload and encrypted them inside ct.

Computation overhead on TC. In eTAP, the trusted client TC has to periodically generate and distribute the garbled circuits $\tilde{F}$, associated garbled constants C, and encrypted decoding blobs $\tilde{d}$ to TAP. For simplicity, we will use the term garbled circuit to denote the set $(\tilde{F}, C, \tilde{d})$. On average it takes 4.1 ms to generate one garbled circuit. Based on prior work [89], we assume that an average user has 26 rules installed and that each rule will be executed once every 15 minutes, which is the default interval used by IFTTT to contact its trigger services [150]. Therefore, we estimate that the TC of an average user needs to spend 10.2 seconds per day to generate 2,496 circuits. Since the average circuit size is 25.3 KB, the estimated amount of data that TC has to send to TAP per day is 61.7 MB, which is less than the data required to back up 25 high-res photos or a 1-minute HD video (1920x1080 px @30 fps) to a cloud service [92, 116], a common task executed daily by modern smartphones.

Storage overhead. TAP needs to store all circuits uploaded by TC until they are executed. Based on the dataset in Section 3.3, there are 12.4 million rules (counted by number of installations) running in IFTTT that are

connected to private triggers. If we assume, conservatively, that all of them require computations, that each rule will be executed once every 15 minutes, and that each rule on average requires 25.3 KB of storage per execution based on the sampled rule set, the total storage overhead for using eTAP would be 28 TB. Given that cloud storage is inexpensive [48], the overhead is manageable. eTAP introduces little storage overhead to AS, TS, and TC. TS and AS only need to store a 16-byte key ($\mathbf{k}_T$ and $\mathbf{k}_A$) and the current circuit id j (4 bytes) for each user. TC needs to store the circuit id j and the keys for each service connected to the user's installed rules, since it can delete the circuits it generated after uploading them to TAP.

Latency and throughput. We first measured the end-to-end latency of running each rule individually and computed the average. The average latency of eTAP is 139 ms, which is similar to PlainTAP (110 ms). The increase in latency should be tolerable, considering the delays in current trigger-action systems are usually 1 to 2 minutes [150]. Then, we issued concurrent requests to trigger every rule at the same time and recorded the maximum throughput. The throughput of eTAP is 96 requests per second (RPS), which is 45% of the throughput of PlainTAP (211 RPS). Overall, we have shown that eTAP can run real rules with a modest performance impact.

3.8 Related work

A few studies have investigated the security issues in IFTTT-like systems. Most closely related is the work of Fernandes et al. [102] where they first introduce the compromised TAP model, and then built DTAP, a system to prevent the misuse of stolen OAuth tokens. They focus only on the integrity problem. By contrast, our work subsumes DTAP by providing confidentiality to the private trigger data passing through TAPs and

adding authenticity of trigger-compute-action rule execution.

Chiang et al. [84] recently propose Obfuscated TAP that handles metadata attacks. They propose techniques to hide trigger data arrival patterns and the types of trigger and action services from the untrusted TAP. Their work also performs end-to-end encryption of trigger data but cannot support computations. In contrast, eTAP focuses on protecting sensitive trigger data while allowing computation — a common use-case in real-world rules (e.g., filter codes in IFTTT).

Bastys et al. [60] classify the sensitivity of IFTTT's trigger and action services and show that 30% of IFTTT's apps may violate privacy by exfiltrating private information to a third-party. Xu et al. [198] analyze how much private data can be harvested by TAPs. They demonstrate that IFTTT has access to more data than necessary. For example, IFTTT monitors devices even if they do not trigger actions. This motivates our work in protecting all information from a malicious TAP.

A popular line of work investigates the semantics of rules and how they violate security policies or interfere with each other. Surbatovich et al. [180] present an empirical study of IFTTT apps and categorize the apps with respect to potential security and integrity violations. Wang et al. [190] design iRuler that uses SMT techniques to discover inter-rule vulnerabilities. This work is orthogonal to ours as it deals with rule semantics and the TAP is considered trusted. By contrast, our work protects trigger data from a malicious TAP.

Cryptographic techniques for secure computation. There is a large body of work on privacy-preserving outsourced computation. Garbled circuit is a particularly popular approach [70,117,131,133,176]. However, most practical approaches tend to be application-dependent [67,129]. Since our setting differs from a generic multi-party setting (as discussed in Section 3.5, we needed to develop a customized protocol).

For evaluation of string operations in a secure two-party computation

setting, Mohassel et al. [152] introduced Obliv-DFA, a custom non-GC based protocol that only supports regular expression matching. Extending Obliv-DFA to substring extraction and replacements would not be possible without drastically changing the protocol and incurring significant overhead. eTAP, on the other hand, supports string operations through a novel and efficient purely circuit-based approach. This allows functional composition with other operations such as substring extraction/replacements and simple transfer of security properties of GC.

3.9 Discussion and Limitations

Security against metadata leakage. Some rules reveal sensitive information just because they are executed. For example, consider the rule: "IF I leave home, THEN turn off the WiFi." TS sends a message to AP only when the user leaves the home. In our threat model, TAP knows the rule semantics. Therefore, when TAP observes a message from this particular TS, it will learn that the user has left the home. Such metadata leakage from side-channels is hard to prevent cryptographically. Recent work [84,198] has applied *cover traffic* to protect time-sensitive information in trigger-action rules by hiding the real trigger events among fake-but-identical ones. We discuss a simple modification to eTAP that uses cover traffic without requiring TC to generate new (fake) circuits.

TC generates a set of circuits and transmits them to TAP, as before. Let J denote circuit indices in this set. TS also internally keeps track of the set of circuit indices J' that have been used with *real* data. To send real data, TS picks random $j \in J \setminus J'$, updates $J' \leftarrow J' \cup \{j\}$, and continues as before (Section 3.5). To send fake data, it picks random $j \in J$, and then sets the garbled trigger data to random bits. TAP executes the chosen circuit ID as before and sends output to AS. When fake data is evaluated on a circuit, the decryption at the AS will fail with very high probability and thus, it

will ignore the message. We present an example to illustrate this approach. Assume TC generates two circuits with ids $J = \{j_1, j_2\}$ and TS has to send five events $e_1, e_2, \ldots, e_5$, among which only e_2 and e_3 are real. Following the scheme above, it transmits the following sequence of circuit ids to TAP: j_1, j_1, j_2, j_1, j_2. We see that j_1 is used multiple times: first for e_1, then for e_2, and finally for e_4. TAP will notice that j_1 circuit was executed thrice, but it cannot distinguish which of these executions was on real data.

This approach is secure due to two reasons: (1) TAP cannot distinguish between executions on real or fake data due to the garbling procedure we use in eTAP [204] (see Proposition 1 for a proof-sketch); and (2) TAP cannot learn anything from circuit usage statistics because of how TS selects j. In addition, circuits can be executed multiple times but *at most only one* of them will be on real data. Such re-evaluation of circuits on random data does not affect GC security properties [64].

Integrating with existing Trigger-Action Systems. We contribute a clean-slate redesign for trigger-action platforms providing data confidentiality from the ground up. As such, it is not immediately backward compatible. However, eTAP's design attempts to minimize these required changes as follows:

First, we create a new TC, a mobile app that users must install on their phones to interact with eTAP. The app mimics the user experience that trigger-action platforms like IFTTT or Zapier currently offer. For example, the user clicks on buttons in a wizard-style user interface to program a rule. TC transparently generates keys and GCs in the background and shares them with TS, TAP, and AS (accordingly) — the user does not have to take any additional action.

Second, the existing TS and AS need to adapt to eTAP protocol. Specifically, both need to communicate with TC to receive keys $(\mathbf{k}_T, \mathbf{k}_A)$. Additionally, TS has to send encoded labels to TAP instead of plaintext trigger data, and AS has to run the decoding function on circuit output (Fig. 3.5).

We have built a library that trigger/action services can use to upgrade their APIs to perform the above operations.

Third, TAP has to evaluate GCs. It also has to cache circuits it receives from TC. We observe that TAP is already setup to perform these tasks — executing code at large scale and managing user-specific data. Although this incurs a resource cost, we believe that it is acceptable given the strong confidentiality and integrity guarantees our work provides.

Rule semantics. A malicious TAP can learn about a user's automation patterns using its knowledge of rule semantics. Although we encrypt the trigger data, TAP can still observe the source endpoint of the trigger data and the destination of the encrypted result. As future work, we envision using results from anonymity networks like Tor [97] to hide the sources (trigger service) and destinations (action service) of messages.

Circuit id synchronization. eTAP requires TC and TS to synchronize on the circuit id j. TAP in eTAP cannot execute a rule if the j specified by TS is not present in its database of GCs sent by TC. This can happen, for example, if TC fails to generate circuits for a certain day due to technical glitches, but TS continues to generate trigger data. We do not want TS to support additional APIs to inform TC about its current circuit id j. Instead, we can rely on TAP to provide this information. TS attaches an encrypted (using $\mathbf{k}_T$) blob containing the circuit id and the timestamp to TAP along with other data during rule execution. TAP forwards that blob to TC on request from TC. Thus TC can learn the current value of j and can detect if TAP sends a stale message.

Loss of the trusted client (TC). TC in our setting is the "root" of trust for generating garbled circuits. TC can be an app running on user's personal mobile device. However, the app has to store a number of important states necessary for continued execution of a rule, such as $\mathbf{k}_T, \mathbf{k}_A$, OAuth tokens, j, f, c, etc. Therefore, the states on the trusted client

must be preserved in case the device is lost. We can use standard cloud-based solutions to back up the states. For example, the states can be encrypted under a user's password and backed up in a cloud drive. The client can recover the states and continue to operate on a new device once the user connects their cloud drive accounts.

Circuit usage feedback. Different rules execute at varying rates. TAP can monitor rule execution frequency to make predictions about future circuit usage and optimize the number of circuit generations and transmissions. TAP can lie about these statistics; however, it does not affect on the security of eTAP. We leave its implementation to future work.

3.10 Summary

Trigger-action platforms allow users to connect independent web-based or IoT services to achieve useful automation. They provide a simple interface that helps end-users create trigger-compute-action rules that pass data between disparate Internet services. Unfortunately, TAPs introduce a large-scale security risk: if they are compromised, attackers will gain access to sensitive data for millions of users. To avoid this risk, we propose eTAP, a privacy-enhancing trigger-action platform that executes trigger-compute-action rules without accessing users' private data in plaintext or learning anything about the results of the computation. We use garbled circuits as a primitive, and leverage the unique structure of trigger-compute-action rules to make them practical. We formally state and prove the security guarantees of our protocols. We prototyped eTAP, which supports the most commonly used operations on popular commercial TAPs like IFTTT and Zapier. Specifically, it supports Boolean, arithmetic, and string operations on private trigger data and can run 100% of the top-500 rules of IFTTT users and 93.4% of all publicly-available rules on Zapier. Based on ten existing rules that exercise a wide variety of operations, we show that

eTAP has a modest performance impact: on average rule execution latency increases by 70 ms (55%) and throughput reduces by 59%.

4 MINTAP: MINIMIZING DATA ACCESS IN TRIGGER-ACTION PLATFORMS

In this chapter, we present minTAP, another design solution to tackle the privacy problems in trigger-action platforms. Compared to the eTAP protocol we introduced in the previous chapter, minTAP offers a different tradeoff. Instead of providing zero access to plaintext data, minTAP does allow the trigger-action platforms to learn a subset of attributes in the data, but only to the amount that is necessary to perform the platform's functionality, thus achieving least privilege. Meanwhile, minTAP achieves significantly better performance and requires no modification to the platform.

4.1 Introduction

The core privacy issue in Trigger-Action Platforms, or TAPs, is that they receive more data than they need to execute user-created rules. Specifically, there are two key design flaws in TAPs that can cause data privacy problems. (1) *Attribute-level* overprivilege allows exploiting the APIs designed by third-party services to send significantly more data attributes than what is necessary to execute the rule. (2) *Token-level* overprivilege of the OAuth tokens that TAPs receive from the third-party services allows exploiting the tokens to use various APIs on the service, even if they are completely unrelated to the rule. While Fernandes et al. [102] consider token overprivilege for integrity, we point out that, when combined with attribute overprivilege, these tokens also permit TAPs to read more sensitive information than needed, thus creating more opportunity for privacy violations.

Considering the rule in Fig. 4.1. The rule connects Outlook email with Slack — an email arriving at the user's inbox from `bank@xyz.com` will *trig-*

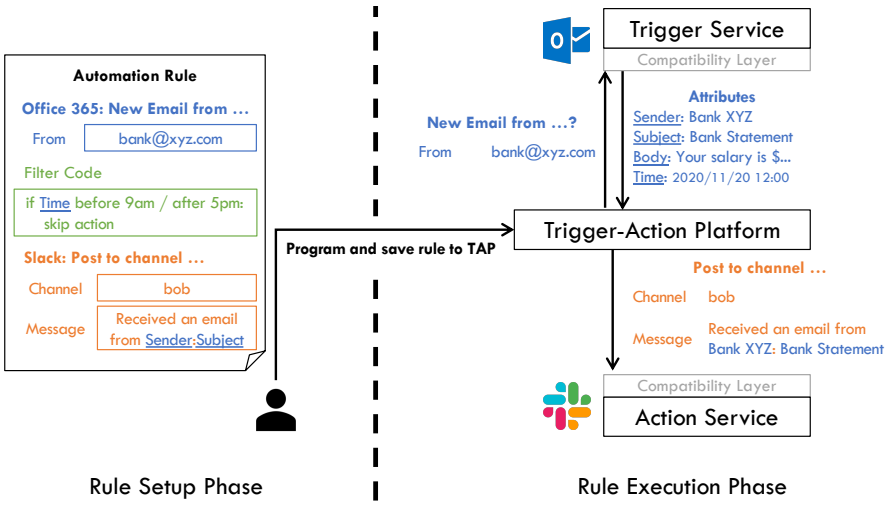


Figure 4.1: An example automation rule in trigger-action platforms. The boxed fields represent various information that the user needs to specify.

ger the rule that performs the *action* of sending a Slack notification with the email sender's address and subject line *if* the email arrives between 9 am and 5 pm (otherwise, no action is performed). For this rule, the TAP only needs the email sender and subject line, and only for emails that arrive between 9am and 5pm. However, currently the TAP will receive *all* email data from that sender at *all* times. A fundamental design choice in TAPs is to favor ease-of-use for third-party services and end-users. Coarse-grained APIs and tokens avoid frequent permission requests, saving users from going through the authentication prompts multiple times. Unfortunately, this comes at the cost of privacy — it violates the principle of least-privilege [170]. It can also create friction with legal frameworks like the General Data Protection Regulation (GDPR) [107] and the California Privacy Rights Act (CPRA) [22]. These frameworks stipulate *data minimization*, a principle restricting data collection to "what is necessary in relation to the purposes for which they are processed" [107].

Motivated by the above, we explore improving user data privacy on TAPs, a largely unexplored area in trigger-action platforms [59]. We

contribute to the design and implementation of minTAP, a system that follows the principle of data minimization to ensure that the TAP only receives the user data it needs to execute user-created trigger-action rules. Specifically, our techniques automatically detect and withhold unnecessary trigger data. There are two challenges in achieving this property: (1) Determine the amount of data a trigger-action rule needs. (2) Mitigate privacy issues in a practical way that does not require changes to the TAP while only negligibly impacting user interaction.

Automatically determining the amount of data that a rule needs is challenging because that amount can vary depending on the rule semantics. IFTTT rules may contain user-created code snippets (called *filter code* in IFTTT terminology) where the set of required data depends on code behavior. In our running example, if the time is outside of the 9am to 5pm window, then the rule needs *no* data. Inspired by recent work on *language-based data minimization* [54], we leverage program dependency analysis to enforce data-minimality in rules. Our approach demonstrates how to construct lightweight *static and dynamic minimizers*, which take as input a rule and output the information needed by the rule while sanitizing unused data (Section 4.5). We also provide guidance to trigger service developers on what types of minimizers would best suit their needs and the related trade-offs (Section 4.9).

For the second challenge on remaining compatible with existing TAPs and not burdening end-users, we decouple trust in rule creation and execution steps. On current systems, these steps occur on infrastructure provided by the TAP vendor (e.g., rule creation occurs on a webpage that IFTTT hosts while rule execution occurs in the IFTTT backend). This implies that a user has to trust this entire stack. Per our threat model, the TAP is untrustworthy, and thus, rule creation must occur elsewhere (otherwise, the TAP can simply modify a rule to request all information independently of the user's needs). Inspired by recent work on decentralizing trust in

TAPs [102], we introduce a trusted client application that helps a user create rules. Thus, rather than trusting the TAP to correctly help a user create rules, each user in the system only trusts their client application. The minTAP client transparently rewrites user-created rules with program dependency information and then stores that information on the TAP with cryptographic integrity protection. We demonstrate this technique on IFTTT without requiring any cooperation. Finally, minTAP requires a small modification to the trigger service's existing IFTTT-compatibility layer. We build a portable Python library that services can use to update their compatibility layer and perform data minimization based on the program dependency information when a rule executes.

We evaluate the privacy benefits of minTAP on 34,419 real-world IFTTT rules that operate on sensitive data — it correctly identifies and removes a median of 4 sensitive attributes that IFTTT does not need for rule execution. Examples include users' emails and downloadable links to private files. We also find 376 filter codes inside these rules and detect that 84% of them may lead to the skipping of actions: when skipped, minTAP will remove a median of 5 attributes.

4.2 Filter Code in Trigger-Action Platforms

In trigger-action platforms, connected web services can create *triggers* that will notify the TAP about an event (e.g., "new email arrived" or "door unlocked") or *actions* that allow the TAP to issue operations (e.g., "send a message" or "turn on the light"). For each trigger and action, the services host APIs to handle the communication with the TAP. Trigger APIs feed trigger data containing a number of *attributes*,[1] such as `Sender`, `Subject`, `Body`, and `Time` in the context of our example rule in Fig. 4.1.

[1]Different TAPs may use different terminologies. For example, in IFTTT, rules are called *applets* and attributes are called *ingredients*.

```
let str = Office365Mail.newEmail.Subject
if (str.indexOf('IFTTT') === -1) {
  Slack.postToChannel.skip()
} else {
  Slack.postToChannel.setMessage('Email ' +
↪   Office365Mail.newEmail.Subject + ' just received!')
}
```

Figure 4.2: Example filter code.

A *rule* connects a trigger to an action. The service providing the trigger is referred to as *trigger service*, and the service providing the action as *action service*. Each trigger and action can provide multiple user-configurable *fields*. These fields represent the parameters that the TAP appends to its API call to the corresponding services. In Fig. 4.1, the trigger has a From field which can be used to customize which email address can trigger the rule. Similarly, Channel and Message are the action fields that customize the API call that the TAP sends to Slack. The user may also specify the values of action fields with the trigger attribute names.

A rule can do further processing of trigger attributes using *filter code*. On IFTTT, filter code is a JavaScript code snippet that may customize the action fields based on trigger data. Filter code may also *skip* the action event altogether based on some condition. Fig. 4.2 shows an example of filter code that sends a Slack message only when a new email containing the keyword "IFTTT" is received [21]. The variable Office365Mail.newEmail is an object that holds the trigger attributes, such as Subject, and Slack.postToChannel provides a list of functions, such as setMessage(), to set the values for different action fields. When one of these functions is called, the original value of the corresponding action field is overwritten. The skip() is a special function: upon invocation, rule execution aborts.

For interoperability with third-party services, popular TAPs like IFTTT and Zapier specify a compatibility layer that the participating services must implement to host TAP-specific APIs and translate the service's original

authorization and data APIs into a format that the TAP understands [123, 205].

4.3 Data Privacy in Overprivileged Trigger-Action Platforms

Prior work [102] has examined the integrity issues that overprivilege causes. We provide a first look at the *data privacy* issues that result from overprivilege. This motivates the design of our data minimization framework.

Attribute-level overprivilege. Typically, each trigger API contains multiple attributes. Unfortunately, under the current practice, the trigger service transmits *all* these attributes to the TAP *regardless* of whether the rule needs them. These unneeded attributes can contain sensitive information, leading to *attribute-level overprivilege*. Consider the example rule in Fig. 4.1, the trigger service provides four attributes (i.e., Sender, Subject, Body, and Time) in the trigger data sent to the TAP. However, one of the attributes (i.e., Body) is never accessed in the rule's execution. We give three example IFTTT rules in Fig. 4.3 showing that many sensitive attributes are being sent to IFTTT even though they are not required for rule execution. Users of TAPs can further customize the behavior of a rule by writing *filter code* that can access trigger attributes and modify action fields. Thus, based on the execution path of the filter code, the set of attributes a rule uses can change. Consider the filter code in Fig. 4.2. When the condition in the *if* statement holds, the entire action will be *skipped* and hence no trigger attributes are required; otherwise, the TAP only needs the email's Subject to correctly execute the rule.

Token-level overprivilege. Privacy concerns on TAPs extend beyond attribute overprivilege. As noted in Section 4.2, TAPs acquire OAuth tokens

with a broad *scope* for enhanced usability, so that users can enter their password for a trigger/action service only once even if they create multiple rules using them. These tokens enable TAPs to execute a large number of APIs on behalf of the user. If an attacker obtains such a token (either by compromising the TAP, or by tricking a user), they can use the token to get unfettered access to all of the user's sensitive trigger data serviced by the APIs that are in the scope of the token, even if these data are not required for any of the TAP's supported rules. While this issue was first identified by Fernandes et al. [102] our experiments confirm that it is yet to be addressed by current TAPs. Although finely-scoped tokens could mitigate this overprivilege, they will drastically hamper usability as users will have to authenticate to services every time they create a rule.

From overprivilege to minimization. The constraints of usability and functionality that lead to attribute- and token-level overprivilege are fundamental to the design of trigger-action platforms. Nevertheless, such overprivilege violates the principle of data minimization that mandates sharing only necessary user data [22, 107] and puts users' privacy at risk should the TAP (or the tokens intended for the TAP) be compromised. Our work identifies a sweet spot in the design space that mitigates attribute- and token-level overprivilege while respecting the usability and functionality constraints, with negligible change to the user's experience and no modifications on the TAP.

4.4 Threat Model and Design Goals

Our goal is to ensure that trigger services release the *minimal* amount of data that user-created rules need without modifying the trigger-action platform or requiring significant changes to the existing user experience. We first discuss the threat model under which we want to achieve these goals and then outline the design requirements of the solution. Finally, we

IFTTT rule description	**Trigger**	**Trigger attributes** (unused ones shown in ***bold italics***)
Get notification before your next event starts [9]	Google Calendar: Any event starts	Title, ***Description***, ***Where***, Starts, ***Ends***, ***EventUrl***
Automatically save in Pocket the first link in a Tweet you like [3]	Twitter: New liked Tweet by you	***Text***, ***UserName***, ***LinkToTweet***, FirstLinkUrl, ***CreatedAt***, ***TweetEmbedCode***
Payments over ___ send you a phone call [14]	Square: New payments over a specific amount	***Merchant***, ID, TotalCollectedMoney, ***DeviceName***, ***PaymentAt***, ***RecordURL***

Figure 4.3: Examples of IFTTT rules, where several sensitive attributes of trigger data are not used by a rule but still sent.

discuss a few potential approaches and point out why they do not meet our security or functionality goals.

Threat Model

In line with prior work on security and privacy of TAPs [80, 84, 102, 172, 198, 206], we assume that the TAP is untrustworthy, meaning that it may deviate from the protocols with the goal of stealing user data that it should not know about (i.e., user data that is not involved in any user-created rules). It can, for example, try to modify the user's installed rules or impersonate the user. Action integrity attacks (e.g., changing or dropping the action of a rule) are orthogonal to this work and are addressed in complementary approaches [84, 102], which we envision will compose well with minTAP. Denial of service is also outside of our scope. Therefore, our focus is on privacy issues arising from overprivilege and how the TAP can take advantage of this fundamental flaw.

We assume that the third-party trigger services, which are the originators of user data, are trusted and do not collude with the TAP. For example, services like Outlook and Dropbox are the source of sensitive user data and have no incentive to collude with the TAP to reduce the privacy of their users. These services have a TAP-mandated compatibility layer to

host APIs that communicate with the TAP. We also assume that the users who create trigger-action rules never act against the interests of their own data privacy, but they might be malicious towards the data of *other users*. For example, an attacker can sign up to the TAP as a user with the goal of trying to steal other user data from trigger services.

As noted earlier, users interact with the trigger-action platform via an app running on a smartphone or computer. We assume that the client device and the app they use to interface with the TAP are trusted and not compromised. We adopt this decentralized trust model from existing work [80, 84, 102]. Unlike the current setting where all users trust a single entity (i.e., IFTTT), in our design, each user only trusts their own device and the apps running on it.

Design Goals

Security. Our primary goal is to ensure that the TAP is correctly privileged at both the token- and attribute-level, so that it can only obtain the data that is absolutely necessary for executing the user-created automation rules. This security goal is in line with the data minimization principle. Therefore, the trigger services should only send the necessary amount of user data to the TAP. Such information may vary dynamically based on the attributes of trigger events for different executions of user rules. In addition, the design must not open up new vulnerabilities in the trigger/action services.

Functionality. The approach to reducing overprivilege in TAP must abide by the following functionality goals: (1) It must be compatible with existing trigger-action platforms, such as IFTTT, Zapier, MS Power Automate, etc., without any modification. In our case, we prototype with IFTTT, a widely popular TAP with 20 million users [122]; (2) It should support current real-world trigger-action rules that can include filter code; (3) User experience should remain similar and any security-relevant changes

should be handled transparently; (4) The trusted client that users use to set up rules should not be required during rule execution; (5) All changes to the trigger service should be contained within its existing IFTTT-compatibility layer; (6) It should transparently support consumers of trigger APIs that are not minTAP-aware (e.g., users who do not want privacy preserving features, non-TAP consumers of trigger APIs). These functionality goals are necessary to ensure we preserve the characteristics of trigger-action platforms that made them popular among users and trigger/action services.

With the threat model and design goals set, we show why naive solutions do not fulfill our security and design goals.

Potential Solutions and Challenges

A trigger service could create rule-specific APIs to reduce attribute-level overprivilege and API-specific tokens to reduce token-level overprivilege. However, the former will require the trigger service to know the rules that users create with their services (a challenge on its own). The latter will require recurring updates to trigger APIs to provide desirable functionality in the face of changing user demands, increasing API maintenance burden. The API-specific tokens will also create a usability burden as users will then have to authorize the TAP every time they create a new rule.

Another potential solution could be to run the rules on the trigger service and communicate the results to the action service directly, without requiring the TAP. However, this will break the independence between trigger and action services, a key property that allows them to evolve independently of each other. For example, there is no reason for an email service provider to know the API details of a chat room. With this naive solution, the trigger service will be required to learn the API details of every service for which the user creates automation rules. TAPs provide a critical layer of abstraction that permits cross-service automation without the

services knowing about each other. Thus, a practical solution to mitigate privacy issues resulting from overprivilege cannot require changes to how the ecosystem functions. A variant of this potential solution is to run the rule on the trigger service and only transmit the results of rule execution to the TAP which then simply forwards the results to the action service. However, this, in addition to the problem stated above, requires modifications to the TAP, violating our design goals.

We therefore take a different approach and build minTAP that operates with existing TAPs and enable trigger services to apply data minimization to their trigger APIs. During rule creation on the client device, minTAP will create a data minimizer for the rule and store that on the TAP as a trigger parameter. The trigger service will receive the parameter during rule execution, apply the minimizer, and send only the minimized trigger data to the TAP. The minimizer ensures all but the attributes necessary for the rule execution are replaced with some default values (e.g., empty strings). Next, we discuss how to generate such practical minimizer functions in Section 4.5, and how we design minTAP to use minimizers without modifying IFTTT (Section 4.6).

4.5 Data Minimization Model

Data minimization reduces the set of trigger attributes sent to TAPs by only transmitting the ones necessary for rule execution. For rules without filter code, we can identify the minimized trigger attributes as the ones that are used in the action fields. For rules with filter code, we develop a practical minimization model that uses data-flow dependency analysis.

Language-based data minimization. We draw on the recently proposed theory of *language-based data minimization* [54]. Intuitively, a *minimizer* is a function that reduces the inputs to a rule without changing the rule's behavior. An *optimal minimizer* removes *all* redundancy from the inputs.

In an ideal scenario, we want to construct an optimal minimizer for a given TAP rule. Unfortunately, finding it is undecidable [54]. Prior work on building minimizers either resorts to verification [54] relying on manual intervention or testing value coverage [161] to produce meaningful results. *Automatically* building *practical* minimizers is an open problem.

We propose an automatic approach to building practical minimizers for TAP rules. As confirmed by our experiments from Section 4.8, filter code consists of small code snippets not written with adversarial intent [21]. This makes *static and dynamic code data-flow analysis* of filter code feasible and thus opens up opportunities for building *practical* minimizers that can protect sensitive user data from unnecessarily being exposed to the TAP.

Data Minimization Model

From Preprocessing to Data Minimization

An abstract setting [54] that allows us to model the essence of data minimization, modeling rules *functions* from input to output. This matches the setting of TAPs where the rules correspond to functions initialized by the ingredients and conditions from the fields in the user interface and computing the action ingredients by running the filter code. IFTTT's filter code is batch-job, with no allowed I/O [121], which justifies its model as a function from input to outputs without side effects.

Informally, a minimizer m for a function f is a function that can reduce the input to f without changing the behavior of f. Let f be a function computed by a given rule. Without loss of generality assume $f : \mathcal{I} \mapsto \mathcal{O}$, mapping inputs $(i_1, \ldots, i_n)$ to output o, such that $f(i_1, \ldots, i_n) = o$, or $f(i) = o$ for short. We recall the definition of *preprocessor* by Antignac et al., where $\circ$ denotes function composition:

Definition 1 (Function preprocessor [54, 161])**.** *A function* $m : \mathcal{I} \mapsto \mathcal{I}$ *with the same domain and range as* f *is a* preprocessor *for* f *iff* $f \circ m = f$ *and*

$m \circ m = m$.

Intuitively, a preprocessor is an idempotent function that, when run ahead of the main function, does not change the behavior of the main function. A preprocessor property is useful to reason about the correctness of data minimization. Indeed, reducing the input to the function must not changes the behavior of the function. The idempotence property ensures that all the redundant input is reduced all at once. For example, dropping a redundant input/attribute (by replacing it with a default value) is a valid preprocessor. Indeed, the default value for the redundant input would not change the value of the function.

To be able to reason about individual runs, we generalize this definition to value-sensitive preprocessor, which we call a *run preprocessor*. A run preprocessor allows us to tune minimization depending on the input data at hand.

Definition 2 (Run preprocessor)**.** *Function* m *is a run preprocessor for* f *on input* i *iff* $f \circ m\,(i) = f(i)$*, and* $m \circ m = m$.

Note that while being a preprocessor is a necessary correctness condition for a data minimizer, not all preprocessors actually minimize data. For example, the identity function is a valid preprocessor for any function and yet it does not reduce the input.

Minimization by (In)dependency Analysis

We now discuss a basic principle for creating practical minimizers. Our key insight is to identify input attributes that have no impact on the rule functionality, so that they can be dropped by the minimizer. A function is *independent* of a subset of input attributes if the function output never depends on what values those attributes take. That is varying the values of that subset of attributes will not affect the function's result. This relates to the well-studied notion of *noninterference* [109]. For our purposes, we

term this *regular independence*. If the independence is specific to particular values certain attributes take, then we call it *run independence*. Under run independence with respect to a *particular* subset of attributes, varying the input values of the remaining attributes will not affect the function's result. Let $i|_J$ denote projection of input tuple k to the subset restricted to indices J.

Definition 3 (Run independence)**.** *Assuming* $J \subseteq \{1, \ldots, n\}$*, a function* f *on input tuple* i *is* independent *of input indices* J *if for all input tuples* j *whenever* $j|_D = i|_D$ *then* $f(j) = f(i)$ *where* $D = \{1, \ldots, n\} \setminus J$.

Definition 4 (Regular independence)**.** *Assuming* $J \subseteq \{1, \ldots, n\}$*, a function* f *is* independent *of input set* J *if for all input tuples* $i \in \mathcal{I}$*, the output of* f *on* i *is* (*run*) *independent of* J.

Regular and run independence open up possibilities for building practical minimizers. This is achieved by static and dynamic program analysis to track whether a given input is used in computing the output of the rule. Recall the example rule from Fig. 4.1. By statically analyzing the rule, we conclude that the body of the email is never used in the output. Therefore, the rule is regularly independent of the email body. A minimizer can thus withhold the email body (e.g., by replacing it by the empty string) without changing the functionality of the rule. Similarly, when invoked outside the working hours, the rule run is independent of all inputs in which case no data needs to be sent to the TAP.

The following two theorems establish the correctness of the minimizers in the sense that a dynamic minimizer that enforces run independence is a correct run preprocessor and that a static minimizer that enforces regular independence is a correct function preprocessor.

Theorem 4.1 (Dynamic minimizer)**.** *If the output of the function* f *on* i *is independent of* J*, then the function* m*, defined by* $m(i_1, \ldots, i_n) = (j_1, \ldots, j_n)$*, is*

a preprocessor for f *on* i*, where*

$$j_k = \begin{cases} \textit{default}, & \textit{if } k \in J \\ i_k, & \textit{otherwise} \end{cases}$$

Proof: By Definition 2, we need to show that $f \circ m\,(i) = f(i)$ and $m \circ m = m$. Clearly, $f \circ m(i) = f(i)$ because replacing independent inputs in J with default values *default* will not be reflected on the output of f by Definition 3. In addition, m is idempotent by construction, and so we have $m \circ m = m$.

Theorem 4.2 (Static minimizer)**.** *If a function* f *is independent of* J *then function* m *is a preprocessor for* f*, defined by* $m(i_1, \ldots, i_n) = (j_1, \ldots, j_n)$ *where*

$$j_k = \begin{cases} \textit{default}, & \textit{if } k \in J \\ i_k, & \textit{otherwise} \end{cases}$$

Proof: By Definition 1, we need to show that $f \circ m = f$ and $m \circ m = m$. Regular independence on I entails $f \circ m = f$ because replacing independent inputs I by default values by m will not reflect on the output of f, as guaranteed by Definition 4. Furthermore, m is idempotent by construction, and so we have $m \circ m = m$.

Practical data minimizers for TAPs.

We contribute practical minimizers that use data-flow analysis. We define a practical minimizer to be a function that takes as input the trigger data D_T and some auxiliary information m computed based on the rule r, and outputs modified trigger data where values of the unused attributes in rule r are removed. minTAP supports two types of minimizers: static and dynamic. A static minimizer computes the list of required trigger attributes by statically analyzing the rule (including the filter code), leveraging regular independence. A dynamic minimizer computes the list of

required trigger attributes for rule execution by running an instrumented version of the filter code that tracks trigger attribute usage, leveraging run independence.

Generating auxiliary information for minimizers. The auxiliary information assists the minimizers in computing the set of required trigger attributes. Algorithm GenMinimizerInfo in Fig. 4.4 presents the algorithm for generating the auxiliary information. It takes a rule $r = (T, A, f)$ where T is the set of trigger attributes (e.g., `Sender` and `Subject` in the example rule in Fig. 4.1), A consists of the value of each action field (e.g., `Channel` and `Message`), and f represents the filter code.

This algorithm first computes the dependency set T', which includes T_{a_i}, the set of trigger attributes required by each action field $a_i \in A$, and T_f, the set of trigger attributes appearing in the filter code f. In addition, it also transforms f into f' by (1) adding data-flow tracking logic to track the access of trigger attributes and action fields, (2) replacing skip() with an empty return, and (3) replacing action API calls with stubs. The last modification serves two purposes: to track which action fields are overwritten and anonymize the action API semantics because we do not want to leak them to the trigger service. We name f' as the transformed filter code and, along with the dependency set T', they form the minimizer auxiliary information m.

Executing data minimizers. Once the minimizer information is generated, the trigger service can choose to run one of the minimizers on trigger data D_T, which contains the trigger attributes and their associated values. In case of static minimization (SMinimizer), the trigger service simply crosses off the value (e.g., replaces with some default value $\perp$) for attribute in D_T if it does not belong to any of the sets in T'. In case of dynamic minimizer (DMinimizer), the trigger service executes the instrumented filter code f' on the current trigger data D_T, which, during the course of its execution, records the set of trigger attributes accessed ($T_{f'}$) and set of action fields

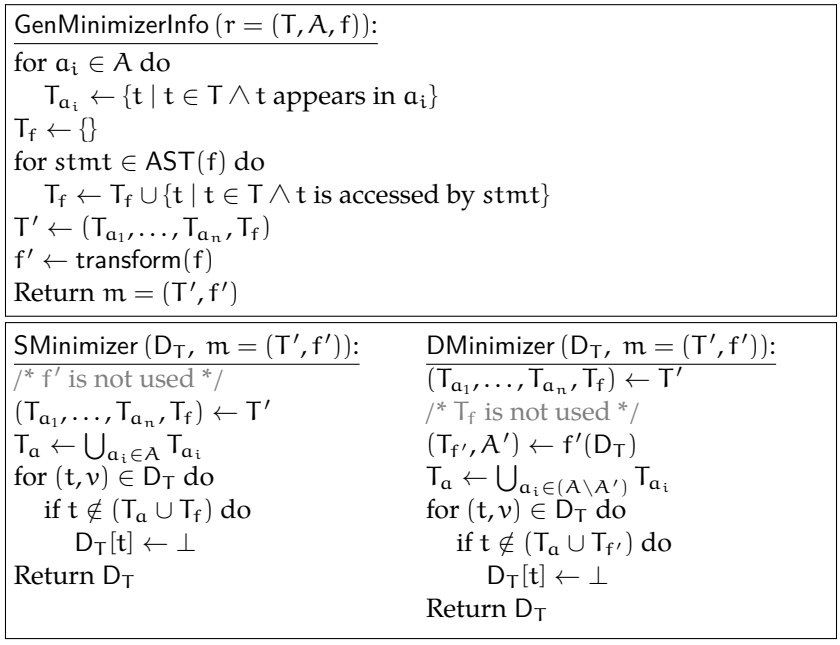


Figure 4.4: Generating the auxiliary information required for running static and dynamic minimization is shown at the top, and how this auxiliary information is used is shown in the bottom two procedures. For a rule r, T is the set of trigger attributes, A is the values of action fields, f is a filter code, D_T is the trigger data.

overwritten (A'). If an action field a_i is over-written by f', the minimizer adjusts T' by removing T_{a_i}. Then, similar to static minimization, the dynamic minimizer replaces all the values for attributes that do not belong to any dependency sets.

4.6 minTAP Framework

We discuss the design of the minTAP framework and show how it uses the minimizers from the previous section to ensure that the TAP only receives the necessary amount of sensitive attributes it needs to execute user-created rules. This tackles both the attribute- and token-level over-

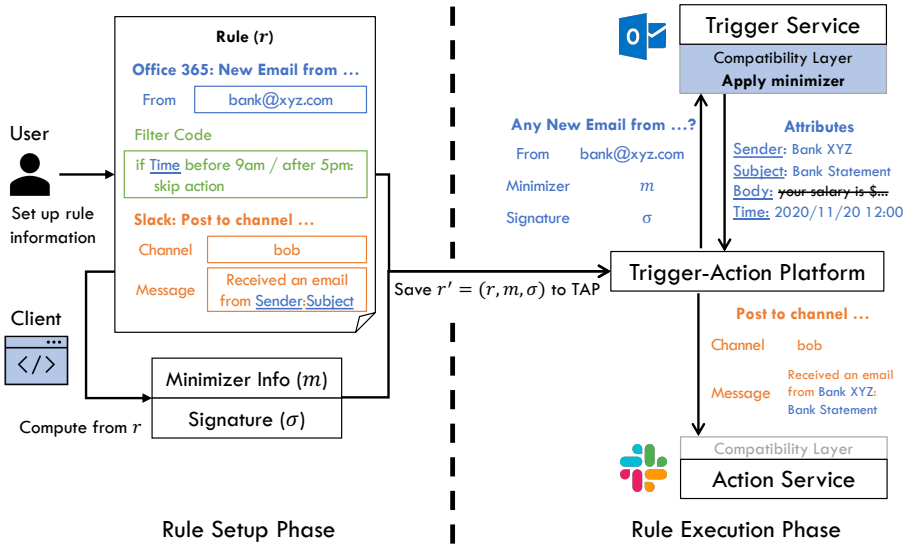


Figure 4.5: minTAP Framework. The blue-shaded background represents the components of minTAP: a client application and a modification to the existing IFTTT-compatibility layer of trigger services. The user creates a rule r, which is then transformed by the client into r′ that contains minimizer information (m) with integrity protection (σ). During rule execution, the TAP contacts the trigger service with (m, σ). The trigger service returns minimized data by removing attributes not needed for rule execution. All of this works transparently to users and the TAP.

privilege privacy issues. We also discuss how the design achieves the functionality and security goals from Section 4.4. We integrate minTAP with IFTTT due to its wide user base [122]. minTAP ensures that real-world rules run with the minimum amount of trigger attributes they need without changing IFTTT or the rules themselves. Thus, it is a practical technique that privacy-conscious trigger services can use with lightweight changes to their infrastructure.

Design Overview. minTAP framework consists of two components (Fig. 4.5): a compatibility layer (or shim) that trigger service installs on top of its existing IFTTT-compatibility layer, and a client for each user in the form of a trusted browser extension. Together, they ensure that IFTTT is cor-

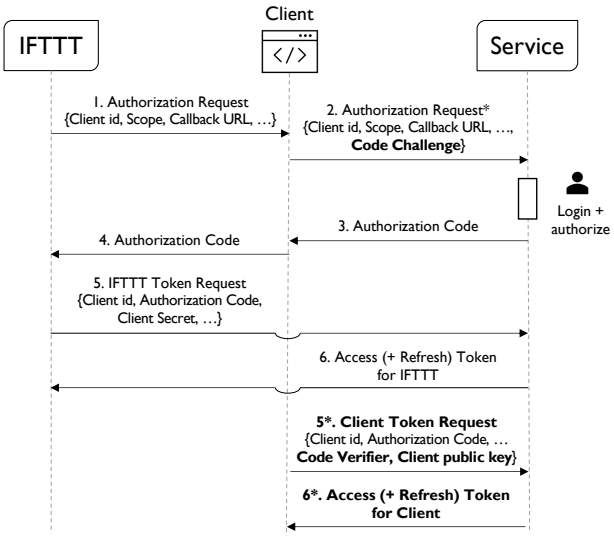


Figure 4.6: minTAP authorization phase: The non-bold text represents the original OAuth 2.0 authorization code flow used between IFTTT and the service, while the bold parts highlight the changes introduced by minTAP's trusted client.

rectly privileged at the attribute- and token-level without requiring any co-operation from IFTTT. At a high level, when the user configures a rule, the client will read the information on the rule setup interface to generate the *minimizer auxiliary information*, based on the algorithm described in Section 4.5, as well as a *signature*, which ensures the rule's integrity. During rule execution, these information will be forwarded to the trigger service, which will apply the minimizer (either statically or dynamically) and filter out unused attributes.

We organize the following discussion around the life-cycle of a trigger-action rule in the case of IFTTT: (1) Authorizing IFTTT to trigger/action services; (2) Setting up rules; and (3) Rule execution.

Service Authorization Phase

The trigger service has to verify that the information it receives from IFTTT is authentic and not modified. Our design achieves this by signing the information. Thus, the trigger service needs to receive the public key of the client for signature verification. This client's key is service- and user-specific. Although the client could upload this key by initiating a separate OAuth session with the user and trigger service, it hinders the usability. Therefore, we integrate the OAuth Proof Key for Code Exchange (PKCE) protocol [17] into the current OAuth protocol that runs during the service authorization phase to simultaneously authorize both IFTTT and the client.

Before a user can create a new trigger-action rule, they must authorize IFTTT to access their data on the trigger and action services through the standard OAuth 2.0 authorization code flow. This is a one-time operation that occurs the first time the user programs a rule with a new service. Subsequent rules involving the same services do not go through the authorization process — this is a key usability trade-off in IFTTT. Our work maintains this trade-off while mitigating the negative privacy effects.

During service authorization, minTAP's client, deployed as a browser extension, intercepts and transparently modifies the first two steps of the OAuth sequence, namely, the authorization request and code response. This is possible because these steps are implemented through browser redirects. The client also creates a service-specific key pair $(\mathsf{sk}, \mathsf{pk})$. Fig. 4.6 shows the extended OAuth flow. When the client encounters an authorization request from IFTTT to a service (*Step 1*), it generates a large random string and computes its cryptographic hash value. We refer to this string as code verifier and to its hash value as code challenge. The client appends the code challenge to the authorization request (*Step 2*). After the user successfully logs into the account and approves the requests, the service will redirect the browser to the callback URL, which is an endpoint of IFTTT,

with the authorization code appended (*Step 3-4*). The client records this authorization code silently for later use. Then, in the background, IFTTT's server will post a request for the access token using its client secret (*Step 5*). The service will reply with a special access token (*Step 6*), which can access the service's APIs only when accompanied by a valid signature.

Concurrent to IFTTT's token request, the client will also issue a token request with its code verifier and public key pk (*Step 5**). Upon checking that the code verifier is consistent with the code challenge, the service will accept and store the public key pk. Finally, a special token is returned to the client (*Step 6**), which can be used to revoke a public key or upload a new one if desired.

We note that our protocol combines both the current OAuth authorization code flow and the PKCE flow, and thus inherits their security properties (see Section 4.7 or more details). Finally, all protocol-level extensions occur transparently to IFTTT and the end-user, thus achieving our goals of not creating changes to the user's experience or to how IFTTT works.

Rule Setup Phase

In this phase, minTAP achieves two main goals. First, it generates the auxiliary information for *minimizing* user-created rule that can be used by the trigger service to filter out unused attributes. Second, it computes a digital *signature* to ensure the authenticity and integrity of the information, preventing the attacker (i.e., IFTTT) from modifying it. The signature, in combination with the access token that IFTTT acquires from the service authorization phase, serves as a logically-fine-grained token that uniquely identifies the rule and prevents token-level overprivilege. If IFTTT tries to invoke a trigger service API, it must always present a valid token and a valid signature — any other requests are automatically denied.

Fig. 4.7 shows the workflow of the setup phase. The user creates a

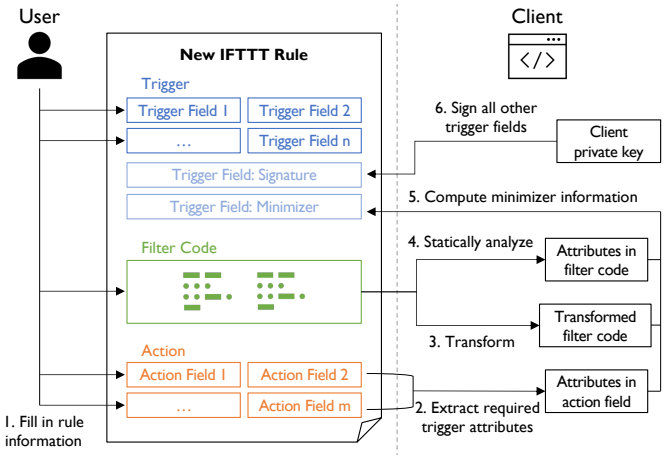


Figure 4.7: Rule setup phase. The left part represents a high-level abstraction of IFTTT's rule setup interface. The right part details the steps performed by the client (as a browser extension) in the background.

new rule (or modifies an existing one) through the interface provided by IFTTT (*Step 1*). This involves selecting a trigger and an action from the appropriate services, specifying trigger and action fields, and optionally writing the filter code. We define the combination of all these data to be the rule information. Before the user saves the rule to IFTTT, the client transparently and atomically captures the rule information. It then computes the auxiliary information m required for the static and dynamic minimizers (*Step 2-4*) as instructed in Fig. 4.4.

This information is needed by the trigger service during rule execution to apply the minimizer. As the trusted client might not be online during execution (functionality requirement, Section 4.4), we store the minimizer as part of IFTTT's rule information. To achieve this, minTAP's compatibility layer registers an additional trigger field with IFTTT to hold this special minimizer parameter. This appears as an additional user-configurable trigger field in the rule setup interface. The client will automatically fill in the value of this new trigger field with the minimizer information (*Step 5*).

Because IFTTT could tamper with the minimizer information before

sending it to the trigger service, the client sets up another user-configurable trigger field to hold a signature σ. We additionally observe that even if IFTTT does not modify the minimizer information, it can still modify other trigger fields to request unauthorized data — in our running example (Fig. 4.1), IFTTT can change the `From` field to get the email from another person. Therefore, in addition to the minimizer, the client signs all of the original trigger fields as well as the identity of the trigger. The client also automatically fills in the signature value (*Step 6*). Once the user hits save, all this data is persisted inside IFTTT.

The trigger, trigger fields, and minimizer information define the amount of data the user wants IFTTT to access. Together with the signature guaranteeing integrity and authenticity, this forms a correctly-privileged fine-grained token that mitigates privacy issues from overprivilege.

Rule Execution Phase

When a rule executes, IFTTT contacts the trigger service to obtain data attributes [123]. This HTTP request from IFTTT bundles all the trigger fields as query parameters, including the auxiliary information of minimizer $\mathfrak{m}$ and the signature σ. Upon receiving this information, the trigger service will first verify the integrity and authenticity of the request, which includes checking if the access token is valid (per standard OAuth procedure) and if the signature is correct using the public key pk corresponding to that user.

Once verified, the trigger service will use the minimizer information ($\mathfrak{m}$) to apply the minimizer on the trigger attributes to sanitize unused values. As mentioned in Section 4.5, minTAP provides two minimizers — static and dynamic — with varying levels of precision and performance overhead. The trigger service can run one of the two functions SMinimizer or DMinimizer (in Fig. 4.4) on the trigger data D_T. While running SMinimizer is straightforward, running DMinimizer could require executing

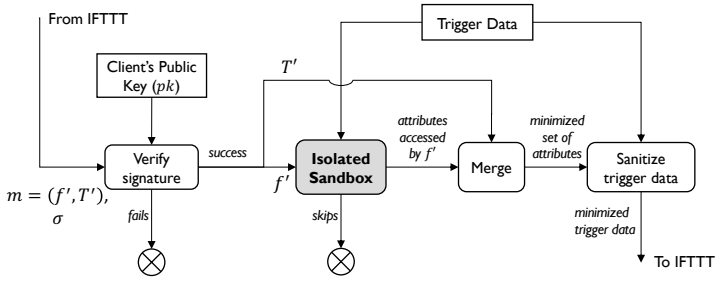


Figure 4.8: Figure shows the rule execution steps with *dynamic minimization* at the trigger service. IFTTT queries the service with the minimizer auxiliary information $m = (f', T')$ and the signature (σ). The trigger service applies DynamicMinimizer on the trigger data m, and responds with the sanitized trigger data to IFTTT.

untrusted client/user-provided code f'. We show the dynamic execution flow in Fig. 4.8. Based on our threat model, a malicious user could use this opportunity to violate the security of the trigger service or its other users. Therefore, we deploy an isolated JavaScript container with strict security policies to prevent the code from affecting anything outside the container. We provide more details on how to configure the container and integrate it into our system in Section 4.8.

The static minimizer is straightforward to deploy, requiring no additional computing infrastructure on the trigger service. However, static minimizers are inherently conservative because they do not have access to the actual values of trigger data. On the other hand, dynamic minimizers can benefit from knowing the trigger data when minimizing the set of necessary attributes. For example, if the filter code hits a *skip*, then no action will be performed and hence no data will be sent to IFTTT. This provides significant privacy benefits if the rule executes only in very specific conditions, such as the example shown in Fig. 4.1 and 4.2. Section 4.9 provides a set of guidelines to help trigger services decide which minimizer to run.

We note that the trigger service can continue to support IFTTT users who do not use minTAP: if the request does not come with the mini-

mizer information, the trigger service will reject the request if the user has uploaded its public key (indicating IFTTT maliciously drops the information), and accept otherwise (indicating the user does not use minTAP). In addition, while a user can connect to a mixture of minTAP services and non-minTAP services, all rules they create with minTAP service must be minTAP-compatible, since the attacker (per our threat model, an untrustworthy IFTTT) will gain access to all user data in this service through token- and attribute-level overprivilege even when just one rule is not minTAP-compatible.

We provide a security analysis of minTAP's protocol in Section 4.7, where we show that it upholds three security invariants: (1) only the user's client obtains the client access token, (2) the trigger service only accepts the public keys from the client, and (3) any modifications to the original rule configuration or the information generated by the client will be detected by the trigger service. Together they ensure that the attacker cannot tamper with the protocol to request unwarranted data.

4.7 Security of minTAP

We consider an adaptive attacker (per our threat model, an untrustworthy TAP) who, given knowledge of how minTAP works, tries to circumvent its protections. minTAP enforces three security invariants: (1) only the client should obtain the client access token, (2) the trigger service should only accept the public keys from the client, and (3) the attacker cannot modify the user's intended rule configuration or minimizer information without being detected. We consider each phase of a trigger-action rule's lifecycle and discuss how minTAP maintains the invariants despite the attacker's actions without introducing new security vulnerabilities.

Service Authorization Phase

This phase has to ensure the first two security invariants: only the user's client can obtain the client token and successfully upload its public key to the trigger service. As the attacker is not a global network attacker and has not compromised the victim's browser, it cannot manipulate communication between the client and the trigger service (Step 2-3, 5*-6*). However, it can try to trick the trigger service by impersonating the user's client in the following ways:

Directly request client token. The attacker could try to directly request a client token for a specific victim user by initiating the OAuth protocol in the background. However, this requires either the user's credential for the trigger service account or the code verifier generated by the client — neither is accessible to the attacker per our threat model.

Interfere with ongoing authorization. IFTTT could try to tamper with an ongoing authorization session (e.g., by appending its own code challenge). However, per our threat model, the client is trusted (and in the case of our implementation, the client is an extension that is protected from IFTTT by the browser security model), thus preventing IFTTT from manipulating this process — the client extension will always intercept any redirects pertaining to OAuth.

Modify OAuth parameters. If the attacker modifies any OAuth parameters (e.g., scope or redirect URL), it will deviate from the original OAuth code authorization flow and result in an authorization failure, amounting to a denial of service (outside the scope of our work).

Upload its own key. As mentioned in Section 4.6, the access token acquired by IFTTT in Step 6 does not have the permission to upload new public keys to the trigger service — only the client token has such permission. As we have shown above, the attacker cannot obtain the client token under our threat model. Therefore, the second invariant holds.

Rule Setup Phase

The attacker could try to manipulate the rule and any support information that minTAP generates. We discuss how minTAP detects any manipulation during this phase. At a high level, the client only retrieves a trusted list of triggers and actions directly from service endpoints and directly communicates the entire rule and signature information to the IFTTT backend.

Modify trigger and action fields. The attacker may present false information to the user client during Step 1. For example, it may add a fake action field, tricking the user to use more trigger attributes. As discussed, minTAP's compatibility layer provides an API for the client to directly retrieve a trusted set of triggers and attributes.

Modify user's inputs. This is not possible because the user only interacts with the client that is isolated from the IFTTT frontend code by the browser security model. The client eventually communicates the programmed rule and its signature directly to the IFTTT backend. At that point, the attacker can attempt to manipulate the information, but that will violate the signature, as we show next.

Rule Execution Phase

Finally, we discuss how minTAP prevents the attacker from changing its request to the trigger service, which consists of the rule configuration and the minimizer-signature tuple, to access unwarranted user data. This completes the analysis and fully ensures the third security invariant.

Modify trigger fields. This will cause the signature verification to fail, since all of the original trigger fields are among the information signed during Step 5 of the rule setup phase.

Modify minimizer-signature tuple (m, σ)**.** Dropping the (m, σ) tuple

for users who have uploaded their public keys will lead to a denial of service. As ensured by the second security invariant, the attacker cannot upload its own public key to the trigger service and thus cannot forge the signature. However, it may attempt to swap the correct (m, σ) tuple of this rule with another tuple, (m', σ'), from a different rule. If (m', σ') is generated by another user, σ' will not match the current user's public key. If (m', σ') is generated by the same user but for a different trigger provided by the same service, it will also lead to a signature mismatch, as the trigger info (trigger name and trigger fields) is also among the information signed. If (m', σ') is generated by the same user and for the same trigger but more overprivileged (i.e. requires more trigger attributes compared to the one in question), this request will be accepted but the attacker cannot gain any new information, as it may also acquire this information by honestly executing that overprivileged rule (which is just another valid rule created by the user). Finally, the attacker can send (m, σ) for a rule that was previously deleted — this attack will not work because deletion would trigger a change in the signing key, invalidating older signatures (Section 4.9).

4.8 Evaluation

To evaluate minTAP, we have collected a large-scale dataset of publicly available IFTTT rules, which includes the detailed configurations (such as filter code) of each rule (Section 4.8). Then, we analyze the privacy benefits of minTAP on this dataset in Section 4.8. Finally, we discuss our implementation and evaluate its performance overhead in Section 4.8.

Dataset

Existing IFTTT datasets [150, 185] do not support our evaluation, due to the absence of crucial information like filter code and configurations of

trigger/action fields. These internal configurations of the rule are necessary to determine the auxiliary information of the minimizer. To better evaluate the privacy benefits and performance overhead of minTAP, we have crawled IFTTT[2] and curated a dataset of 59,009 trigger-action rules that are publicly published on IFTTT. To the best of our knowledge, this is the first large-scale dataset that collects the internal configurations (including the configurations of trigger/action fields and filter code) of each rule.

Data collection. IFTTT's developer platform provides an API for accessing the rule configurations for all public rules by their IDs. We obtained the 59,009 valid rule IDs by analyzing the URLs in IFTTT's public sitemap in April 2021. All rules in our dataset are accessible by search engines. They can be installed by IFTTT users and their configurations can be inspected by IFTTT users. For each rule in the dataset, we thus obtained its general information, such as title, description, and the connected trigger/action service, as well as its configuration, which includes the configurations of trigger/action fields and filter code (when available). Out of these rules, 554 contained filter code.

Rules with private triggers. We are only interested in the rules that can access sensitive trigger attributes. Based on the classifications proposed by Bastys et al. [60], we find 34,419 (58%) rules that are connected to *private* triggers (such as emails, documents, and locations, as opposed to public triggers like news reports). In addition, out of the rules with filter code, 376 (68%) are connected to private triggers. For the rest of the section, we will use these private-trigger rules and filter code to evaluate minTAP.

[2]Legal counsel at our institution has confirmed this is considered as fair use under DMCA and does not violate IFTTT's terms of use.

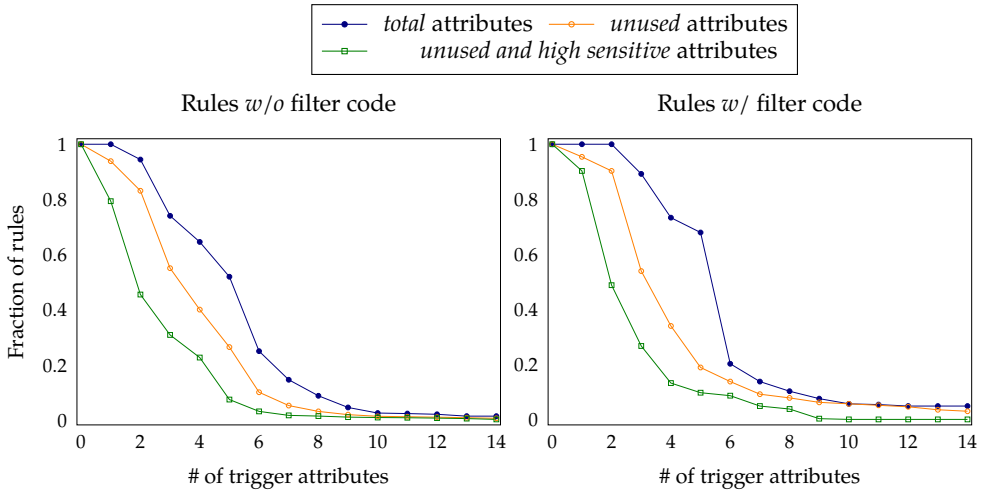


Figure 4.9: CDFs showing the percentage of rules that have at least x total / unused / unused-and-highly-sensitive attributes.

Privacy Benefits

We study the extent to which minTAP mitigates the privacy issues arising from attribute- and token-level overprivilege in IFTTT. The presence of the signature in minTAP's design (Section 4.6) ensures that IFTTT's token can only be used to query data from the connected trigger API, preventing any token-level overprivilege. Therefore, we measure the privacy savings of minTAP in terms of the following two metrics that measure the degree of reduction in attribute-level overprivilege: (1) The number of unused attributes for each rule that would *not* be transmitted to IFTTT; and (2) When filter code is present, the estimated frequency of *skips*, resulting in *no* attributes being transmitted.

Rules *without* filter code. If a rule does not contain filter code, minTAP will apply the static minimizer: each trigger attribute that does not appear in the rule's default action fields will be labeled as unused. Across the 34,419 rules that are connected to private triggers, we find that a median of 4 trigger attributes (or 3.7 on average) are unused. The orange line in Fig. 4.9 shows a cumulative distribution of rules based on the number of

Sensitivity	Resource Type: Example Attributes	% Attr	% Rule
Low	Timestamp: CreatedAt, OccurredAt	26.7%	84.0%
	Access-controlled link: PublicUrl	2.2%	7.2%
	Total	28.9%	86.2%
High	Event description: EventName, About	23.0%	43.7%
	User info: FullName, Email, Number	13.0%	32.7%
	Location: Longitude, Latitude	9.0%	19.1%
	Downloadable link: PhotoUrl, Mp3Url	6.5%	16.9%
	Bookmark: Article, Website	5.3%	4.7%
	Message: Body, Subject, Message, Text	3.3%	9.4%
	Other: SensorValue, Duration, List	4.7%	10.9%
	Total	60.7%	79.4%
Unknown	Generic link: Url, Link	5.0%	15.3%
	Misc name: SheetName, ChannelName	3.5%	11.1%
	Total	8.5%	20.5%

Figure 4.10: Breakdown of unused attributes by sensitivity. Each row represents a category of attributes. The third column denotes, out of all occurrences of unused attributes, the percentage that contains this category's keywords and the fourth column denotes the percentage of rules that have at least one unused attribute with such keywords.

unused attributes. We find that more than 90% of rules have at least two unused attributes. With minTAP, *all* these unused attributes will not be transmitted to the TAP.

We also examine the sensitivity of the unused attributes in these rules. Even if the trigger is considered a private source, not every attribute represents a sensitive resource. We conducted a case study by first randomly sampling 10% out of the 3,255 unique unused attributes and grouping them into different categories based on the types of resources they represent (second column of Fig. 4.10). Then, we picked out the attributes that, when leaked, do not grant IFTTT access to any additional information. We labeled the attributes based on the following sensitivity criteria.

- *Low*: This attribute does not carry sensitive information or represents the event's timestamp. We specifically label timestamps as low since IFTTT can infer them by observing the arrival time of the trigger service's

messages.

- *High*: Exposing this attribute to IFTTT will reveal sensitive information, including personal identifiable information or private files. In some cases, the value of an attribute may be publicly available, such as websites, but the user's access to it can be sensitive. These information is also labeled High. In Fig. 4.9, the green line shows the distribution of unused High sensitive attributes in our rule dataset.
- *Unknown*: Given this attribute's name alone, we cannot distinguish its sensitivity. For example, if an attribute is named `URL`, it can be either a downloadable link to a private file or an access-controlled link that does not reveal any information without user's login credential, depending on the corresponding service's implementation.

Finally, we observed the typical keywords appearing in the attribute's names for each category (the detailed criteria are listed in Section A.3) and estimated the prevalence of each category in the entire dataset based on the occurrences of these keywords. In summary, we found that 60% of the unused attributes are labeled as highly sensitive and 79% of the rules contain at least one highly sensitive attribute (Fig. 4.10).

Rules *with* filter code. We show the CDFs of unused attributes for the 376 rules with filter codes in Fig. 4.9. Most of these rules contain very simple snippets with a few lines of code (left part of Fig. 4.11). 315 (84%) rules include conditions that lead to the skipping of actions. For these rules, trigger service can choose to use either static or dynamic minimizers. The main benefit of dynamic minimizer is that it can determine when a rule needs to be skipped, leading to maximum privacy savings. These 315 rules have a median of 5 attributes — *all* of which will be sanitized if the rule skips, compared to the median of 3 attributes sanitized by the static minimizer. Even when the skipping does not happen, we still find three rules where the dynamic minimizer is more precise than the static

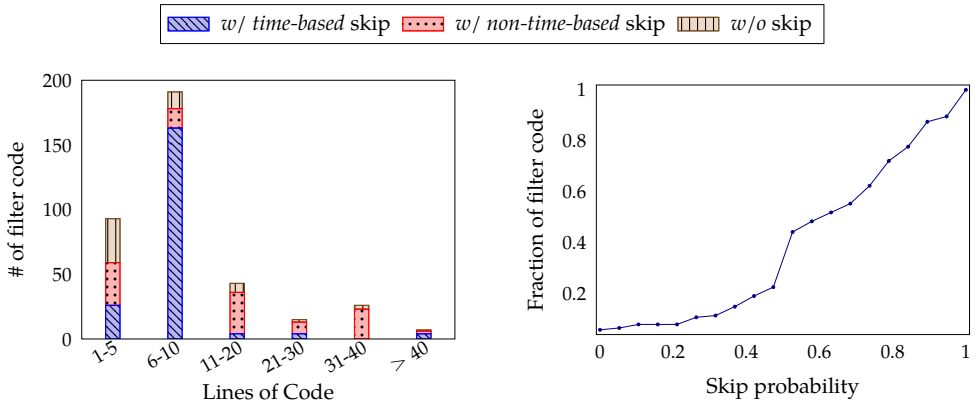


Figure 4.11: Filter code characterizations. (**left**) Histogram of filter codes based on lines of code. (**right**) CDF of the simulated skip probability for time-based filter code.

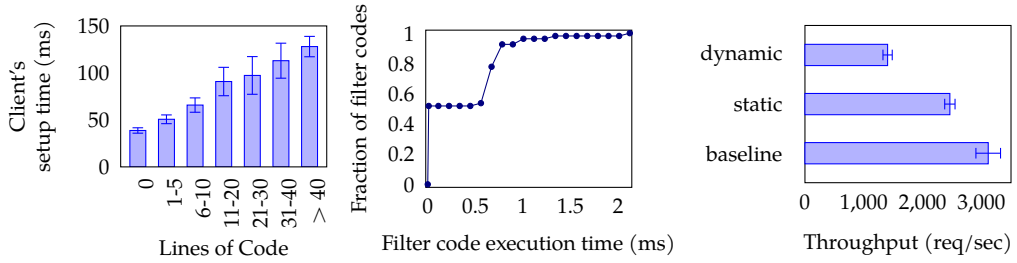


Figure 4.12: Evaluation results of minTAP. (**left**) The average execution time for the client during the rule setup. The rules are separated into different groups, based on the lines of code. (**middle**) CDF of the filter code's execution times in the trigger service's isolated environment. (**right**) The throughput of the trigger service for using static or dynamic minimzer, or baseline (i.e., w/o modification to the compatibility layer).

one, sanitizing 1.7 attributes more on average. In addition, we found 201 (54%) rules compute their skip conditions purely based on the trigger timestamp. If we assume that their triggers occur uniformly throughout the week, then these rules on average will skip 62% of the time (right part of Fig. 4.11).

Performance Evaluation

We evaluate the performance of two components of minTAP, namely the client-side browser extension and the modified compatibility layer on the trigger service. To simulate a trigger service with minTAP additions, we build and deploy an IFTTT-compatible trigger service for testing. The service is hosted on n1-standard-2 instance with 2 vCPUs and 7.5 GB memory on Google Cloud. We install the client on a Macbook Pro with a 2.2 GHz 6-Core CPU and 16 GB memory running Chrome version 87. We measure the performance of minTAP based on the execution latency, service throughput, and memory overhead. Across the board, we find these impacts are modest and acceptable. We did not observe any noticeable effect in the performance of TAP rules due to minTAP.

Implementation Notes. We implement the client as a Chrome extension that monitors the user's interactions with the IFTTT webpage by analyzing the endpoints being visited. For example, it will launch the authorization phase if the user visits URLs like `ifttt.com/[service]/redirect_to_connect`. The shim on service's compatibility layer consists of two pieces: (1) A Python library that will upgrade the trigger service's APIs so that they can engage in minTAP's protocol, and (2) A runtime environment that can securely execute transformed filter code for dynamic minimization. We use Isolated-VM [137] to provide a restricted execution environment. For efficiency, our implementation maintains a pool of 4 warmed-up Isolated-VMs and routes each incoming request into a new sandboxed execution context created inside the VM with least memory usage. We also compile moment.js, a library used by IFTTT for advanced date parsing [121], into the execution context if required. All Isolated-VMs are configured with an explicit timeout of 15 sec and a memory limit of 128 MB to further protect the trigger service's infrastructure.

Latency of the Client

The client's overhead consists of the time it takes to hash the OAuth PKCE code verifier during service authorization phase and the time to compute and sign the aux. minimizer info during the rule setup phase. Hashing for computing the PKCE code verifier takes less than 0.15 ms, and is thus negligible. We setup all the rules in our filter code dataset and report the average latency for setting up each rule. The rule setup time varies with the size of the filter code, measured in lines of code (LoC). The latencies for filter codes of different sizes are shown on the left of Fig. 4.12.

We observe that the client takes approximately 39 ms to compute the minimizer information and its signature when no filter code is present. For the most complicated filter code in our dataset (with more than 40 LoC), it only takes 128 ms. This overhead has a negligible impact on user experience because it hides within larger latencies introduced by the UI — it takes approximately 6000 ms for the browser to fully load the rule setup page and 800 ms to save a programmed rule.

Overhead of the Modified Compatibility Layer

During rule execution, minTAP requires the trigger service to apply (static or dynamic) data minimization of the trigger data. We examined how it affects the overall throughput and latency of the trigger service. For static minimization, we randomly sampled 50 rules from our no-filter-code dataset. For dynamic minimization, we randomly sampled 50 rules from filter code dataset and manually prepared input trigger data to ensure the longest path in each filter code is executed.

Latency. We compute the latency overhead of minTAP as the relative increase in the request serving time with respect to the unmodified trigger service (that does not support data minimization). On average, the latency increases by 5.5 ms when dynamic minimization is used, and only 0.45 ms

when static minimization is used. Since the end-to-end latency of a trigger-action rule in IFTTT is more than two minutes on average [150], the extra latency caused by minTAP's compatibility layer is not noticeable in practice. We further looked into the time to execute the transformed filter code inside the Isolated-VM, to understand the impact of different filter codes on the latency of dynamic minimizer. In the middle of Fig. 4.12, we show the CDF of execution times for different filter codes. We observe that execution is efficient: 96% of the filter codes take less than 1 ms.

Service throughput. We measured the throughput of the trigger service as the number of requests handled per second under concurrent requests. We gradually increase concurrency levels until the throughput saturates (and latency increases). We measure throughput in three conditions: (1) baseline (without minTAP modification to the compatibility layer); (2) with dynamic minimizer; (3) with static minimizer. The throughput for different settings is shown on the right of Fig. 4.12. Overall the compatibility layer is lightweight, throughput is only reduced by less than 50% when dynamic minimizer is used, and by less than 20% when static minimizer is used. Prior work has characterized trigger rates on popular services and determined that the most popular one executes approximately 1,702,353 times, while IFTTT contacts the trigger service every 15 minutes [150], which translates to an average of 1,892 requests per second. With minTAP enabled, the trigger service can handle 1,404 requests per second with dynamic minimizer or 2462 requests per second with static minimizer even on our basic test setup. Considering that many rules do not contain filter code and, therefore, no need to use the dynamic minimizer, even with very limited computational budgets, it will be easy for trigger service to use minTAP modifications on their existing IFTTT compatibility layer.

Memory and storage overhead. With a pool of four Isolated-VM, we recorded a maximum memory usage of 341 MB under the peak throughput (with dynamic minimizer enabled). minTAP's compatibility layer imposes

little storage overhead on the trigger service that only needs to store one public key for each user. Considering IFTTT already requires compatible services to store the data of the past 50 events and recommends them to store a unique trigger id [119] for every rule, the additional overhead of minTAP is negligible.

4.9 Discussion

Adopting minTAP. Trigger services who wish to protect their user data (and possibly reduce friction with legal frameworks) can use minTAP as a lightweight method to mitigate data misuse. They obtain these benefits at the minimal cost of upgrading their existing IFTTT-compatibility layers to include minTAP improvements. As described in Section 4.2, this layer hosts a number of APIs that follow IFTTT's specifications for authorization and data querying, and handles all communications between IFTTT and the service. We provide minTAP as a portable Python library that enables a seamless upgrade. The service provider could potentially increase its computational capacity for the modest performance overhead (Section 4.8), however, existing elastic services might handle this automatically. Finally, we note that other parts of the service's infrastructure do not need to be changed. The technique of minTAP also applies to other commercial TAPs (e.g., Zapier) with slight adjustments in implementation.

minTAP-Client usage. Each end-user trusts only their minTAP-client and it serves as the main contact point between users and the TAP. While the client can take many forms (e.g., mobile or desktop app), we prototyped it as a browser extension for ease-of-use. Once the client is installed, the user does not need to perform any extra operations to create minTAP rules. The client only has permission to interact with `ifttt.com` and send requests to compatible services authorization APIs. It does not save any personal data except OAuth tokens and cryptographic keys using local storage. As

mentioned in Section 4.6, these tokens cannot be used to request user data from the services. We envision that the client and the cloud-based TAP will be separate entities adhering to the minTAP protocol (e.g., similar to the current diversity of Telnet, FTP, SSH client and server software). A user can switch between multiple clients (e.g., if a client device is lost) if they support encrypted cloud backups of the keys and tokens. We leave implementing this as future work.

Deleting/modifying rules. If a user deletes or modifies a rule, the minimizer and signature for the old version should be invalidated — a problem similar to certificate revocation. minTAP-client creates a new signing keypair during a rule-update operation and sends the public key to the trigger service using its special OAuth token. It also transparently updates the signature on existing rules in a background page.

Static vs. dynamic minimizer. minTAP offers the trigger services a choice of whether to run static or dynamic minimization. Recall that static minimization determines necessary attributes at rule setup time, whereas the dynamic minimizer instruments filter code during rule setup and then requires the trigger service to run the instrumented version to learn about necessary attributes. We outline a few considerations to help trigger services make an informed decision.

The advantages of static minimization are: (1) Lower overhead on trigger services; (2) No additional security challenge of sandboxing filter code; and (3) Possibility to run *distributed minimizers* [54]. Distributed minimizers focus on minimizing data that is provided by multiple sources. This is relevant to IFTTT's emerging feature of *queries* [124] that allow pulling data from multiple trigger services. A static extension to handle queries is straightforward: based on the filter code, the client can determine the set of used attributes and pass this information to the relevant trigger services. Note that queries are a challenge for the dynamic approach because the trigger service has no access to data from the other services

that is required to run the filter code.

The advantages of dynamic minimization are: (1) High precision because the set of used attributes may depend on runtime values passed to filter code (static analysis approximates these values). In some extreme cases, the imprecision of JavaScript's static analysis may also in theory deem a used attribute as redundant, although we have not encountered such imprecision in our evaluation due to the non-adversarial nature of filter code. (2) Precise modeling of *skips* and timeouts. When filter code reaches a skip or times out, there is no need to send *any* attributes to IFTTT. Predicting skip and timeout reachability is particularly hard for static analysis.

Encrypting trigger fields and attributes. The OTAP system encrypts trigger attributes and fields when *no* filter code is present [84]. We sketch a simple approach to extend minTAP to fully integrate OTAP's approach. During the service authorization phase, minTAP's client exchanges an additional symmetric encryption key with the trigger and action services. During rule setup, the client encrypts the trigger fields with this key and stores them in the TAP. During rule execution, the trigger service receives the encrypted trigger fields, obtains the minimized trigger data, encrypts the trigger data using the same key, and sends them to the TAP. Thus, minTAP can support OTAP guarantees when no filter code is present.

Performance benefits of minTAP. We remark that in addition to the privacy benefits, minTAP collaterally brings some performance benefits. While there are performance penalties incurred by minTAP's additional computation, minTAP liberates trigger services from generating and sending redundant attributes. The results of the privacy evaluation from Section 4.8 are thus encouraging not only for boosting privacy but also for reducing communication overhead.

Data-specific minimization. The precision of dynamic minimizer can

be further improved by incorporating symbolic execution to achieve data-specific attribute minimization. Symbolic execution allows for automated exploration of the program control-flow graph, precise program state reasoning, and generation of the input that leads to a given program point. For example, if a string attribute is used only in a condition for substring matching, we can replace this attribute with just the substring. Currently, as shown on the left of Fig. 4.11, only a small fraction of filter codes that have non-time-based conditions can benefit from such symbolic analysis. However, rules in other types of trigger-action settings (such as Node-RED [37] and OpenWhisk [34]) where more complicated programming paradigms are required may benefit from symbolic execution. We leave this for future work, bearing in mind that when filter codes contain nested conditions symbolic analysis may become inefficient due to path explosion [144].

4.10 Related work

We refer the reader to the recent work [45, 59, 72] outlining the state-of-the-art on securing TAPs. Our work is inspired by the principles of *least privilege* and *need-to-know* [170].

Privileges on TAPs. Prior work has shown that TAPs obtain overprivileged access to trigger/action APIs [102] allowing them to harvest private information without the user knowing [198] and opening for malicious rule makers to exploit TAP's privileges [43, 60]. This motivates our work.

The DTAP system protects the integrity of rules under a malicious TAP [102]. By contrast, we address the orthogonal question of data privacy. In addition to mitigating the privacy issues that arise from token-level overprivilege, minTAP goes further and addresses the attribute-level overprivilege. DTAP relies on extending the OAuth protocol with so-called XTokens to express fine-grained privileges and requires modifications to

existing TAPs for deployment, whereas minTAP is fully compatible with existing unmodified TAPs.

The OTAP system uses encryption and cover-traffic schemes to protect the confidentiality of data while it transits through an untrusted TAP [84]. This approach can protect data end-to-end, but it does *not* allow computations (i.e., filter code) — a primary feature on TAPs. By contrast, minTAP only releases the attributes that rules need, supports computations on data, making it practical and readily deployable. OTAP and minTAP occupy different points in the design spectrum but can be unified and supported in a single framework leveraging the minTAP infrastructure.

The eTAP system (Chapter 3) uses garbled circuits for rule execution [80]. It provides strong confidentiality and integrity guarantees, but at the price of requiring extensive architectural changes to the TAP, supporting a limited subset of filter code and higher overhead. By contrast, minTAP works with unmodified TAPs and supports more expressive filter code with minimal overhead.

Filter-and-Fuzz analyzes how events from a smart home can be sanitized to ensure that IFTTT does not learn more information than necessary [198]. It relies on textual analysis to identify unnecessary events. By contrast, minTAP uses program analysis to identify unused data attributes. minTAP can benefit from hiding statistical patterns of sensitive events by composing them with the Fuzzing piece of Filter-and-Fuzz.

Secure hardware. Recent efforts leverage secure hardware for protecting users' data from TAPs. Hardware-based trusted execution environments (TEEs) enable computing over the trigger data on the TAP, while preserving the confidentiality [172, 206]. Besides requiring hardware changes to the TAP backends, current TEEs suffer from fundamental security design issues [77, 154, 186].

Language-based data minimization. Data minimization is a principle restricting data collection to "what is necessary in relation to the purposes

for which they are processed" [107]. Antignac et al. [54] formalize the notions of monolithic and distributed minimization for programs with single and multiple sources of information, respectively. They reason about best minimizers that remove all redundant information before passing the data to the data processor. They demonstrate that although computing the best minimizers is in general undecidable, it is possible to approximate data minimizers by symbolic execution techniques. Unfortunately, these techniques require coming up with invariants for programs with loops, a long-standing challenge in program verification [103]. Pinisetty et al. [161] utilize testing techniques to improve the precision of minimizers for programs and leave synthesizing minimizers as future work. Drawing on the work by Antignac et al., we contribute a lightweight data minimization technique that focuses on the attributes used by programs. We generalize the definition by Antignac et al. to be sensitive to individual program runs and show that a simple (and fully automatic) dependency analysis can be used for data minimization by ruling out unused attributes in program runs.

Minimum exposure. Related to our ideas is the line of work on minimum exposure in data collection by authorities. Anciaux et al. [49–51] focus on the case of collecting forms (like tax forms) for governments. They consider the number of inputs to withhold for the privacy of the applicants and discuss data-dependent minimum exposure. However, the computational model is that of assertions on particular shapes of formulas that represent form collection logic, making their algorithmic solutions less applicable to scenarios of general programs. By contrast, our approach naturally extends the language-based approach to data minimization which applies to arbitrary (runs of) programs.

4.11 Summary

We have presented minTAP, a framework for practical data access minimization in trigger-action platforms. We study two levels of overprivileges that are common on TAPs: attribute-level overprivileges, e.g., sending to the TAP the content of emails even if the rule only involves the headers, and token-level overprivileges, e.g., granting the TAP full access to cloud services. To address both types of overprivilege, we put language-based data minimization to work and demonstrate how dependency analysis can identify redundant attributes. We deploy minTAP on IFTTT, showing how to minimize trigger data before it is sent, thus boosting privacy while preserving the functionality. We evaluate the security and performance of minTAP on a set of realistic benchmarks to conclude that minTAP on median sanitizes 4 sensitive trigger attributes per rule, with a tolerable performance overhead.

5 MOHITO: SCALABLE METADATA-HIDING FOR IOT PLATFORMS

In this chapter, we shift our focus to another type of online integration platforms – smart-home platforms. Smart-home platforms can be deemed as a specialized version of the trigger-action platforms we discussed in the previous two chapters. However, instead of triggering on events from third-party services, smart-home platforms act on commands directly from users.

5.1 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 [85,209] 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. Such information can be exploited for targeted advertising and surveillance. The privacy risks associated with IoT data call for a robust privacy protection mechanism that hides interactions between users and devices from IoT servers.

To enhance the data privacy of IoT systems, recent works [79,84] have proposed encrypting data that is communicated through the systems.

However, the mere *existence* of communication with (encrypted) data can still pose privacy risks. 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 [57, 177] and governments [27, 28] 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 [27].

Thus a major challenge in designing privacy protection mechanisms for IoT systems is *metadata* privacy. Although a long line of works have proposed anonymous communication systems that provide metadata privacy [135, 136, 138, 184, 187, 194], 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 [11], and IFTTT [26], 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.

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 [91,95,100], making them less compatible with the demands of our 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 [150]. We should assume that these smaller vendors only have 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) [106, 162], 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 [93,127,148]. 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 [100], the state-of-the-art general-purpose metadata-hiding system, can handle only 40 messages per second under similar settings.

5.2 Background & Motivation

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 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. 5.1 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 interacting with IoT devices [85, 209].

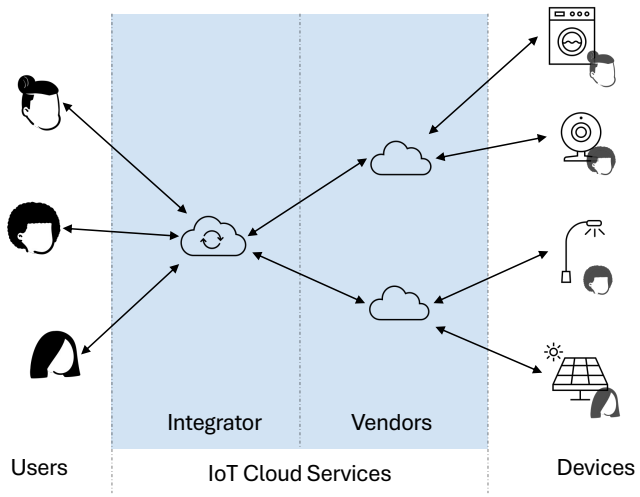


Figure 5.1: Overview of IoT Ecosystem. Users communicate with an integrator, which communicates with various vendors. Each vendor communicates with its own devices.

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 permissions via login credentials, authorizing the integrator service 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.

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 service should

learn of the user's interaction with their devices. There are three main challenges in designing such system.

Metadata-hiding. One challenge in privacy-preserving IoT is that it 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 from the integrator service the user's interaction with the lock by using end-to-end encryption. Nevertheless, the integrator service can still observe that the user sends a command 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. Indeed, both researchers [57,177] and government regulations [27,28] have issued warnings that metadata in IoT systems can reveal sensitive personal information and deserves privacy protection.

Scalability. A key characteristic of IoT system is the scale of data. With millions of IoT devices deployed worldwide, they collectively generate a significant amount of traffic in every second. Therefore, it is crucial for our system to focus on the challenge of scalability first to accommodate the continuous growth of devices in the IoT landscape. We note that there are many works [91, 100] on anonymous communication systems that share similar security goals, but they mostly focus on single message performance, whereas we need to efficiently handle a number of concurrent messages in batches 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 an IoT system. Based on the dataset in [150], the top 20 vendors in IFTTT on average have 3,130 times more connected devices than the bottom 20, which means that smaller vendors may not have the resources prepared to handle the overhead caused by larger vendors' traffic if we were to split the load equally. Therefore, we need to ensure the overhead of each vendor scales proportionally to the number of devices it owns.

5.3 Designing a metadata-hiding IoT system

We present Mohito, a privacy-preserving system tailored to the IoT setting. In this section, we describe our security goals and the architecture of Mohito's design.

System Model

We model an IoT ecosystem as a set of IoT cloud services connected to various users and IoT devices. The cloud services consist of a single integrator service and many device vendors.

Vendor. A device vendor provisions IoT devices. Each vendor may provision multiple types of devices, and it periodically communicates with its devices.

Integrator. The integrator service is 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 in the system 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. Our model enforces this specific communication graph (as shown in Fig. 5.1), since it is common in today's commercial IoT systems. 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.[1]

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.

We model the system to proceed in a round-based fashion. That is, the integrator gathers all messages from users participating in the current round, processes, and delivers them to the corresponding vendor, before advancing to the next round. This round-based assumption closely follows the API design of today's commercial integrator service [29], as they require vendors to handle messages in batches.

[1] In certain scenarios, it might be desirable to let a device actively push a message to its user. We discuss how our system can support this operation at the end of Section 5.6.

Threat Model

In our threat model, IoT services (i.e., the integrator and vendors) are *honest-but-curious*, 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.

Since it is unlikely for an adversary to compromise two IoT services at the same time, we assume that the integrator and the vendors *do not* collude. However, it is reasonable to assume that an IoT service may register accounts with other services or buy devices from other vendors to help it gain information about its competitors. Therefore, we *do* allow a number of users and devices to collude with an IoT service. Specifically, the number of messages that can be generated by these colluding users and devices in each round is at most δ.

We assume that when two parties communicate, they learn each other's identity (for example, through the IP address or an authentication process). For example, when a user contacts the integrator, the integrator learns the identity of the user. We believe this assumption is necessary, as real-world services often rely on user identities to implement rate limiting or to prevent denial-of-service attacks.

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.

Note that prior works have shown that the size of IoT messages can be used to infer device activities [58]. To prevent this, messages can be padded to some fixed size. Padding messages efficiently is an important, albeit orthogonal, research problem [55, 56, 58]. 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.

Security Goals

Our goal is to support a privacy-preserving IoT system 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:

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

❷ **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 user's message.
- For a vendor, it should not learn which non-colluding devices receive a command in each round. We enforce this requirement for vendors because if it learns a device has received a command from user, it also learns the identity of the device and therefore the user that sends the command (per our threat model).

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

❸ **Data integrity.** Although we assume IoT services are honest-but-curious, a malicious user or device can attempt to corrupt the integrity of messages intended for other users/devices. We ensure that the mes-

sage sent by a benign user or device cannot be tampered with by another user/device.

We provide a more formal definition in Section 5.7.

5.4 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 5.5 and finally provide the full detailed protocol in Section 5.6.

One of the fundamental building blocks of Mohito is an Oblivious Key-Value Store (OKVS) [106]. Section 5.4 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 5.4) and *ephemeral ids* (Section 5.4).

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 5.6.

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 [106, 162] 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 $m = (id, cmd)$ to the integrator, where id is the device id and c is the encrypted command.

2. The integrator, after collecting messages $m_1, \ldots, m_n$ from users participating in this round, encodes the messages as an OKVS S, where device ids are keys and encrypted commands are values. The integrator sends S to every vendor.

3. Each vendor decodes from S the encrypted commands belonging to its devices, and it forwards these commands to the corresponding devices.

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 trace 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 (i.e., encrypted commands that also appear random).

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 by the number of vendors and making the bandwidth overhead unbearable. Additionally, small vendors must process an OKVS that consists mostly of commands for other large vendors, violating our load-balancing goal. 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.

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 5.3).

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 5.9. The shuffler functions similarly to a node in a typical mix-net — it receives a list of user messages from the integrator, shuffles them, and sends the shuffled list back to the integrator, 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, each consisting 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.

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 command 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 Section 5.7.

5.5 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 as intersection attack or statistical disclosure attack [93, 127, 148].

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 [91]. 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 certain scenarios common in the IoT settings that can break this assumption. Section 5.5 gives an example attack, illustrating how intersection attacks in Mohito may lead to privacy leakage; Section 5.5 outlines Mohito's defense.

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 receives 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

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

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.

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 [62, 69, 173] and client-side cover traffic [55, 58, 148, 195, 198]. 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 5.9.

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 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 never learns the number of "real" commands received by a particular 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 5.4).

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 add $\mathbf{B}_v$ fake commands for the v^{th} vendor to the shuffled list of messages before returning them to the integrator.

In this way, the integrator now sees that vendor v^{th} receives $\mathbf{C}_v$ commands respectively. 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, $\mathbf{A}_i$ and $\mathbf{B}_i$. 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.

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

[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.

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.

Choosing C.

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. 5.2 shows how the number of fake commands we need to inject changes as $\mathbf{C}_v$ changes and the number reaches 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 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 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.

5.6 Mohito Protocol

In this section, we formalize the full Mohito protocol. Each device in Mohito must be first properly set up 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. Devices respond to the commands with status update messages.

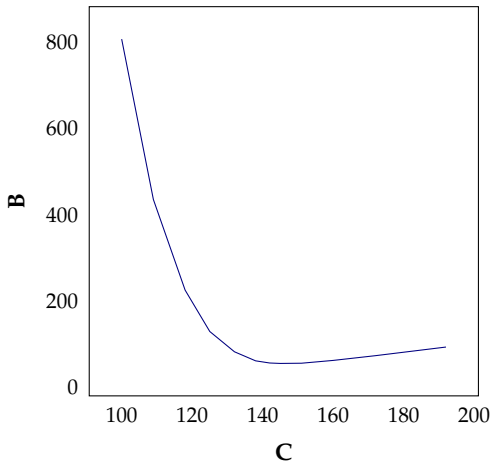


Figure 5.2: Number of fake messages **B**, assuming a total of 100 vendors and **A** follows $\mathcal{N}(100, 20)$.

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 $\mathsf{D} \in \mathcal{D}$ has three attributes: $(\mathsf{User}, \mathsf{Vendor}, \mathsf{ID})$, 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.

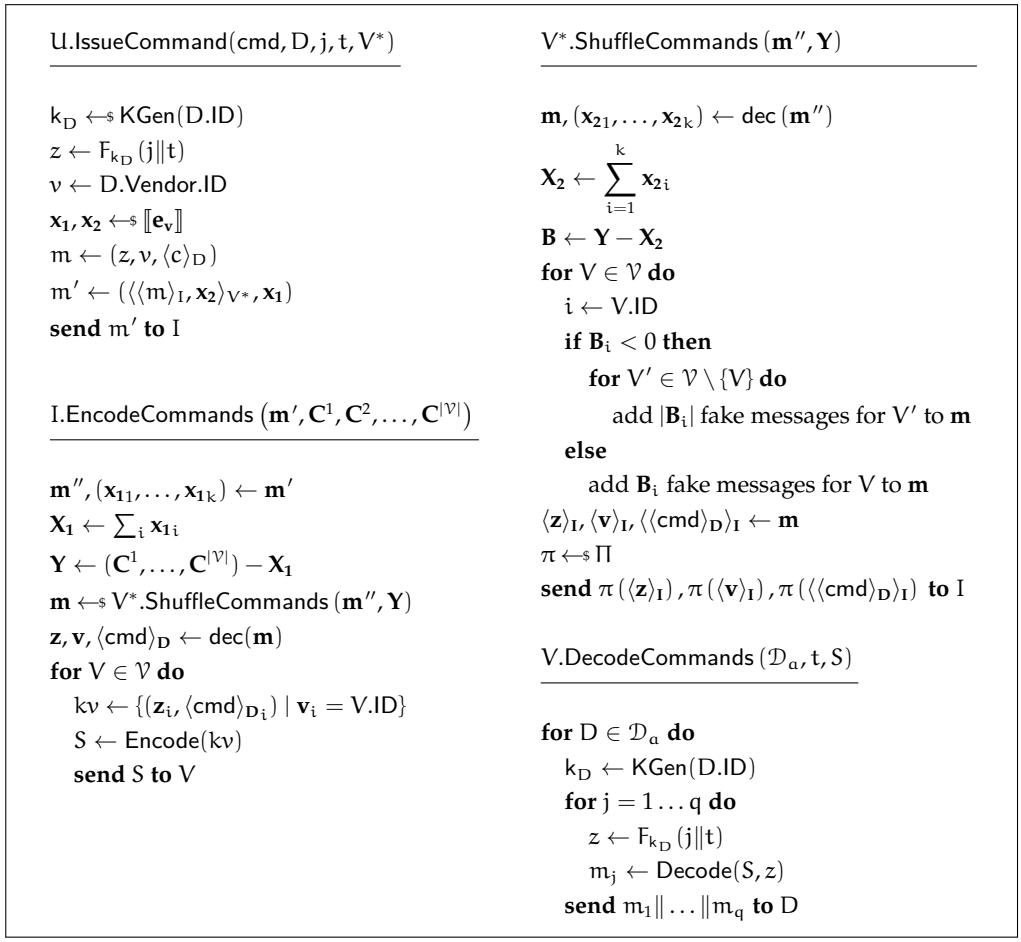


Figure 5.3: Mohito protocol for command sending phase. Each procedure is executed by different entities: user's mobile phone app (U), integrator (I), shuffler vendor (V^*), and the device vendor (V).

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.

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. 5.3.

We design Mohito to proceed in a round-based fashion. The integrator gathers all commands from users that participate in the current round and, together with the vendors, distributes them to the devices. We discuss the appropriate setting for round duration in Section 5.8.

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.

Users issue requests

When a user U wants to issue command cmd to device D, they invoke the function $\mathsf{IssueCommand}$ and send the output to the integrator I. This includes the following operations:

(a) 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.
(b) 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 (which is obtained during the device setup phase of D).
(c) Compute the additive shares $\mathbf{x_1}$ and $\mathbf{x_2}$ as described in Section 5.5.
(d) 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 $1 \leqslant j \leqslant q$. This restriction is reasonable as many real-world servers already enforce similar rate-limiting techniques in their APIs.

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 stores for each vendor using the following steps:
(a) 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^*.
(b) V^* adds the fake messages based on the scheme described in Section 5.5 and returns the messages to I after shuffling (V^*.ShuffleCommands). 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.
(c) I groups the resulting messages by the vendor id v attached in each message, encodes each group into an OKVS S using the ephemeral id z as key and the encrypted commands $\langle \mathsf{cmd} \rangle_D$ as value, and sends S to the corresponding vendor.

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 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. As a result, in Mohito, we force 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 [58, 208], 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 observes a decryption error.

Device Response Phase

After a device D receives a message from its vendor, 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 is a random string.

We note that the protocol in this phase is essentially 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 of the integrator's message list.

This phase starts after a device has received a command from its user and is ready to make a response r. We list the pseudocode of the protocol in Fig. 5.4.

Devices issue responses

The device D performs an onion encryption on its response r by using first D's own symmetric encryption key k_{DD}, then the public key of the shuffler V^*, and finally the public key of the integrator I. The ephemeral

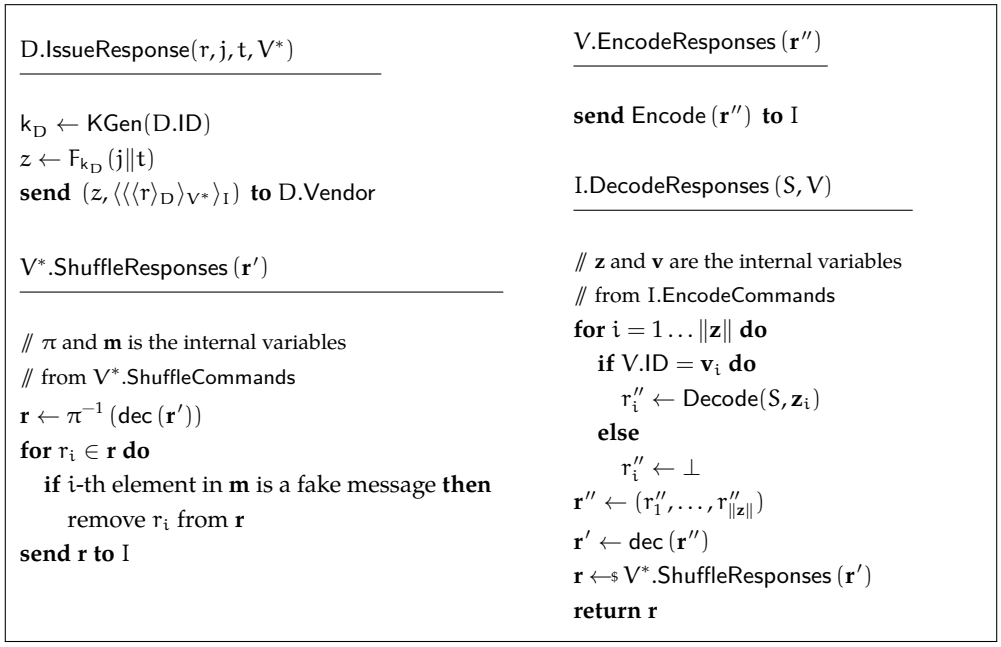


Figure 5.4: Mohito protocol for device response phase.

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 devices actually receive commands by checking whether the device's response is in the response list it received from I during the future decoding step (V^*.ShuffleResponses).

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.

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 $\mathsf{I}.\mathsf{EncodeCommands}$ 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 $V^*.\mathsf{ShuffleCommands}$.

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 in the beginning of $\mathsf{I}.\mathsf{EncodeCommands}$. That is, $\mathbf{r}'_i$ is the response to $\mathbf{m}'_i$.

The user U who initially sent $\mathbf{m}'_i$ can be easily traced back by I. For example, U may still have the TCP connection with I open and awaiting for a response.

Device-initiated messages. In some cases, we may want the system to allow devices to actively push messages to their users. To support this operation, we can switch the order of Mohito's command sending phase and device responding phase. The only difference is that the users who are online and ready to receive push messages should submit a list of their ephemeral ids to the integrator.

5.7 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 $\mathcal{A}_{\mathrm{I}}$ as a party that passively corrupts the integrator and actively corrupts a small subset of users and devices, as discussed in Section 5.3. 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 $\mathcal{A}_{\mathrm{I}}$, produces an output that is computationally indistinguishable from the information listed above.

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

Claim. *There exists an algorithm* $\mathsf{Sim}_{\mathsf{I}}$ *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* $\mathcal{A}_{\mathrm{I}}$*.*

Proof Sketch. The algorithm $\mathsf{Sim}_{\mathsf{I}}$ 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, $\mathsf{Sim}_{\mathsf{I}}$ plays the role of the shuffler. When the integrator requests to shuffle the messages, $\mathsf{Sim}_{\mathsf{I}}$ 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 encode 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.

5.8 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

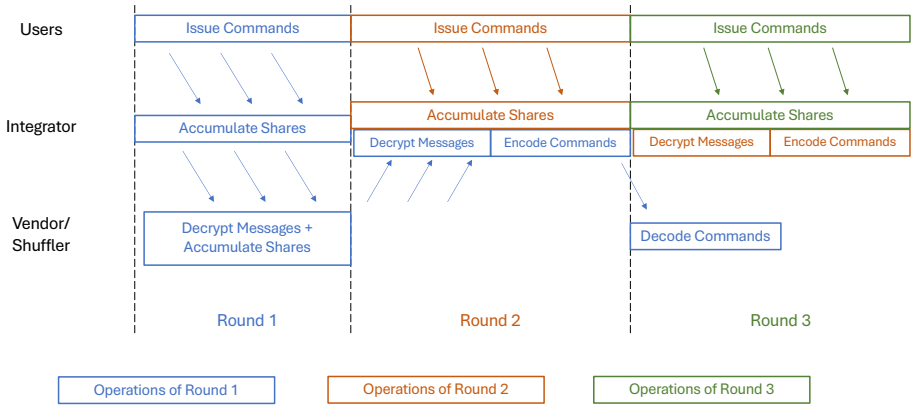


Figure 5.5: Execution graph of our Mohito implementation. Message streaming is used to reduce system idle time.

system, we can achieve 600× more throughput.

Implementation

We prototype the Mohito protocol in Go and choose PaXoS [162] as the underlying implementation for OKVS. 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. 5.5. 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 them back; otherwise, the

integrator would learn that some messages rank lower than others in the permutation.

We also pre-generate 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.

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.

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. Since cost scales linearly with command size, we set the command size to 1 KB and show the results in Fig. 5.6 (left).

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. 5.6 (right) shows how communication changes

[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.

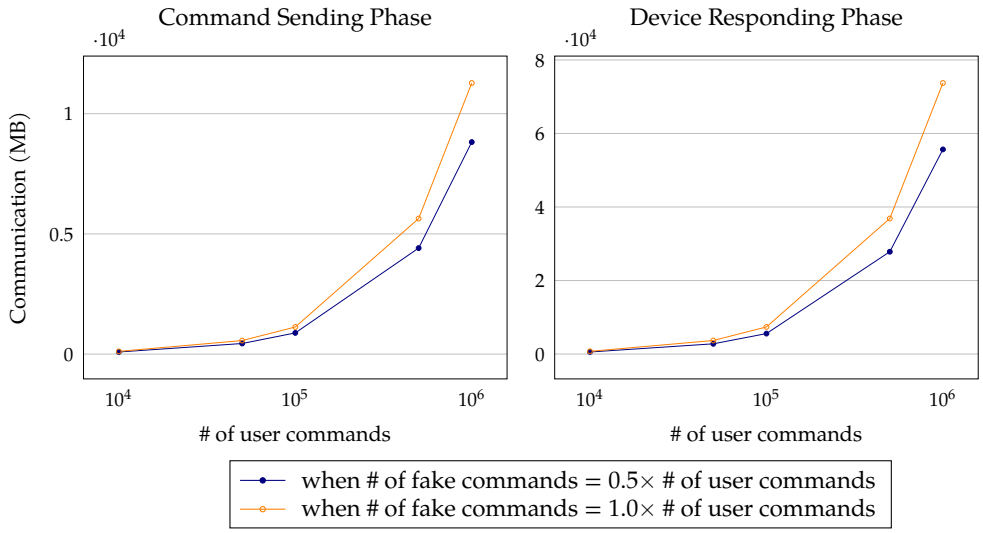


Figure 5.6: The communication cost of Mohito, given a command size of 1 KB. The device responding phase costs more bandwidth than the command sending phase, since the size of OKVSs in former scale with the number of active devices, while the size in latter scales with the number of commands.

when we assume that in each round only 10% of active devices receive commands.

Performance

This section focuses on the performance of IoT servers, since the protocol we run on each end user and each device is relatively simple (it takes only 0.03 seconds on our machine and should not be a bottleneck for modern embedded microprocessors). 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. 5.5, the operations of different rounds overlap due to the use of message streaming. The integrator begins the second round as soon as it finishes forwarding all user messages from

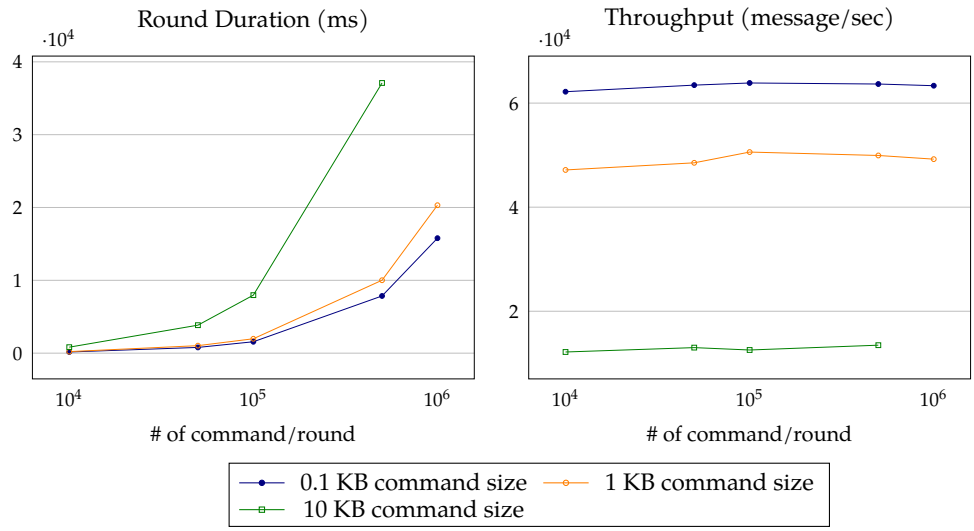


Figure 5.7: Performance of Mohito. Based on the duration of each round (left), we can compute the system throughput (right), which remains unaffected by the number of commands we put into each round.

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 round duration is determined by the execution time of these two operations.

In Fig. 5.7, 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. We note that 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 5.5. 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. 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, encoding/decoding cost is relatively small compared by the decryption cost. For example, given 100,000 commands and a command size of 1 KB, the time needed 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. 5.7, 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. Although we cannot compare Mohito with existing non-metadata-hiding IoT systems due to their proprietary nature, we note that Express [100], a state-of-the-art general-purpose metadata-hiding messaging system that uses a two-server PIR technique, supports around 40 messages per second while running on machines with twice the number of CPU cores than ours. Hence, being a tailored system, Mohito is able to achieve a much higher throughput, so that it can accommodate

the high volume of concurrent requests in IoT environments.

Latency. We define the 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 [100] 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. We note that the bottleneck of our system is decrypting a large list of messages, a highly parallelizable operation. Therefore, performance can be improved by adding more machines.

5.9 Discussion

Practicality of Shuffler. The integrator can choose a different vendor as the shuffler in each round and report this vendor (as well as the public

key of this vendor) to every participating party. 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 putting the load of larger vendors onto smaller ones, the integrator can select the shuffler in a way that ensures the average work 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 [150], 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.

Multiple Shufflers. We note that instead of choosing one vendor as the shuffler, we can also select multiple (or even all) vendors as shufflers. Now each shuffler will be responsible for shuffling a fraction of the user messages the integrator receives and injecting fake traffic to the same fraction of the pre-determined constant $\mathbf{C}$. However, having multiple shufflers does not necessarily improve the overall performance. As shown in Section 5.8, even if we reduce the shuffler's running time, the total time of each round is still determined by the time spent by the integrator's operation, which is not affected by the number of shufflers.

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.

Malicious Security. The current protocol of Mohito only protects against honest-but-curious IoT services, as a malicious integrator can lie about its aggregated sum of secret shares. However, we can upgrade our security guarantee to defend against a malicious integrator by running secret-shared non-interactive proofs (SNIP) [90] on a subset of vendors as long as one of the vendors remains honest.

Preventing intersection attacks via anonymous credentials. As we note in Section 5.5, 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 [62,69,173] and they can be plugged into Mohito directly, as Mohito does not place additional requirement 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 [197], so the IP address alone may not always identify a user.

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 [55,58,198] 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.

5.10 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 analyze how passive network observers, such as Internet service providers and WiFi eavesdroppers, can infer device activities based on encrypted IoT traffic [42,57,58,98,149,183]. 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 are equally powerful, as they both have access to such metadata. In addition, 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 [55,56,58,94,202]. Traffic shaping modifies traffic generated by IoT devices by padding messages and injecting cover traffic. We note that if a user can ensure its client is always online, this approach can be used to complement Mohito's server-side cover traffic approach. EPIC [143] 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.

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 [194], Atom [135], and XRD [136]. 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 [91], Pung [52], Talek [82], and Express [100] instead achieve anonymous communication by reading/writing user messages from/to a private database via private information retrieval. These approaches allow reading and writing to happen in different rounds, but they prioritize 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 [138, 139, 184, 187]. 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 do not degrade over time.

5.11 Summary

We have presented Mohito, a privacy-preserving IoT system that achieves metadata protection. It prevents both the integrator service and device vendors from learning which user communicates with which device. In addition, we tailor the protocol of Mohito to support two key requirements of an IoT system, which are the handling of large concurrent traffic

and the load-balancing among device vendors with various processing powers. 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 a general-purpose metadata-hiding system that provides similar security guarantees.

6 CONCLUSIONS AND FUTURE WORK

In this chapter, we first propose several future work directions based on the scopes of our work and findings, and then conclude with our contributions to the understanding of various security and privacy issues in online integration platforms and potential defenses.

6.1 Future Work

Metadata-hiding with support for computations in trigger-action platforms. A malicious TAP can learn about the metadata of a user's automation rules using the traffic pattern and rule semantics. Although eTAP can encrypt the trigger data, a TAP can still observe the source service of the trigger data and the destination of the encrypted result as well as the timestamp of each execution. As we discuss in Section 5.2, these metadata may sometime allow the platform to infer user activities. Meanwhile, Mohito provides metadata protection but cannot support computations due to the use of end-to-end encryption. While one may substitute the standard encryption scheme with homomorphic encryption schemes [155], the performance overhead would be unbearable. For reference, TFHE [31], a state-of-the-art library for fully homomorphic encryption, takes 4.45 seconds to compute a typical function in automation rules. As Mohito needs to deal with thousands or millions requests in each round to hide traffic patterns, a practical scheme that support both metadata-hiding and computations remains an area of future investigation.

Minimization with symbolic execution. The precision of dynamic minimizer in minTAP can be further improved by incorporating symbolic execution to achieve data-specific attribute minimization. Symbolic execution allows for automated exploration of the program control-flow graph,

precise program state reasoning, and generation of the input that leads to a given program point. For example, if a string attribute is used only in a condition for substring matching, we can replace this attribute with just the substring. Currently, as shown on the left of Fig. 4.11, only a small fraction of filter codes that have non-time-based conditions can benefit from such symbolic analysis. However, rules in other types of trigger-action settings (such as Node-RED [37] and OpenWhisk [34]) where more complicated programming paradigms are required may benefit from symbolic execution. We do note that one particular challenge is that, when filter codes contain nested conditions, current techniques for symbolic analysis may become inefficient due to path explosion [144].

Support for multi-trigger automation rules. Recently IFTTT has developed a feature called *queries* [124], which allows a single automation rule to pull data from multiple trigger services. Both eTAP and the dynamic approach in minTAP do not support this new paradigm of automation rule. This still remains a challenge, as the set of trigger services needed for a rule may be determined at runtime and vary by each execution. One future direction is to build an efficient synchronization strategy between different trigger services to support this feature.

Automated security analysis of access control. In our study of the app ecosystems in business collaboration platforms, we choose only two representatives, namely Microsoft Teams and Slack, due to their popularity and access to their officially hosted app directory. While we do find these two platforms use a similar model and therefore suffer from similar attacks, it is still unknown whether our finding can be applied to other platforms. A promising future direction is to develop an automated approach to discover the potential privacy escalations based on an OAuth-based access control scheme. This could lead to a more generalizable finding and serve as a foundation for a universal defense mechanism.

6.2 Conclusions

This dissertation has delved deeply into the intricate landscape surrounding security and privacy in online integration platforms. Two primary threat vectors are examined in detail.

The first threat revolves around vulnerabilities stemming from access control mechanisms for third-party services. Our research has uncovered shortcomings in the design of these mechanisms, rendering them susceptible to exploitation by malicious parties. This is illustrated through a systemic analysis of the app ecosystem within popular business collaboration platforms including Slack and Microsoft Teams, where we have found that a malicious third-party app can obtain unauthorized access to private messages and gain control of other benign apps. To mitigate such vulnerabilities, a more dynamic and fine-grained permission model is required to adequately define the capabilities of each app.

The second threat is rooted inherently in the current design of integration platforms. These platforms, by acting as centralized data hubs, amass vast quantities of user data, giving rise to potential privacy risks. We have addressed this issue through an exploration of system designs, each providing a distinct trade-off among security, performance, and functionality: (1) in eTAP, we have leveraged the unique structure of trigger-action platform to design an efficient protocol that optimizes garbled circuits and supports secure computations in automation rules; (2) in minTAP, we have developed a framework that provides practical language-based data minimization for trigger-action platforms and achieves least-privilege by releasing only the necessary attributes of user data to the platforms; (3) in Mohito, we have built a privacy-preserving protocol for smart home IoT platforms, offering the advantages of metadata hiding while ensuring high throughput. In constructing these system designs, we have devised a number of novel techniques, some of which (such as Mohito's server-side cover traffic) are tailored towards each platform's unique requirements while

others (such as eTAP's evaluations of string functions in garbled circuits) have broader applications beyond the scope of integration platforms.

In summary, securing integration platforms remains of paramount importance for the security and privacy of user data in modern digital ecosystem, and this dissertation has shown two key strategies: first, a more dynamic and fine-grained access control system is the key to fend off threats from connected third parties; second, the development of new secure and efficient protocols that tailor to each platform's unique system requirement and data communication flow is required to mitigate the privacy concern stemming from the platforms themselves. These strategies serve as vital steps toward fortifying integration platforms against potential security and privacy vulnerabilities.

A.1 Implementation Details of Attacker Apps in BCPs

In this section, we provide more implementation details of our attacker apps demonstrated in Sections 2.4 to 2.6. All apps are implemented by following the official guideline and APIs.

App-to-App Delegation Attacks

In Section 2.4, we demonstrate five delegation attacks. For each attack, the attacker registers a malicious app that provides benign functionality and requests a legitimate set of permissions (detailed below). After that, the attacker either installs the malicious app to their workspace (where the attacker is a curious user) or tricks a user into installing apps in the user's workspace. Once installed and granted permission, the malicious app gets notified and starts the attack by interacting with other targeted apps in the workspace.

The first four malicious apps request permission to send messages on behalf of the user. They launch the attack by sending specific messages that the targeted apps were designed to read and process. The last malicious app requests permission to react to messages on behalf of the user. It launches the attack by reacting with an emoji that the targeted app is designed to notice and retweet.

User-to-App Interaction Hijacking

In Section 2.5, we demonstrate the command hijacking attack on Zoom, which requires implementing a malicious app that mimics the appearance and behavior of the official Zoom app. To this end, we register an app

with slash command permission but deliberately implement the command responses with Zoom APIs (of the attacker's controlled Zoom account) to mimic the official Zoom app. As BCPs permit installing apps from just a public URL, we do not have to publish the apps on official app stores. This approach avoids any accidental distribution of malicious apps to other BCP users.

Furthermore, this attack can be extended to hijack any other apps, as long as the attacker can re-implement the proper functionalities of the targeted app. The appearance of an app is publicly available in the official app directory.

App-to-User Confidentiality Violations

We provide more details of how the attacker can obtain the channel and message IDs described in Section 2.6.

Obtaining channel ID. Each channel ID is a random string. The direct way to learn the ID of a private channel is by requesting a less alarming scope, `groups:read`, which provides the read access to a private channel's metadata. Alternatively, if the attacker knows the name of the channel (through side channels or guessing; per our threat model the attacker can be a curious workspace member who has some prior knowledge), it can use the `chat:write` scope to write a new message. It can just provide the channel name to the corresponding `chat.postMessage` API, which will accept this request and return the channel ID as part of the response.

Obtaining message ID. The direct way to learn the message ID requires `groups:history`, which also grants the ability to directly read messages, avoiding the need for any attack because an app can simply misuse that permission to leak messages. However, unlike channel ID which is completely randomized, the format of a message ID follows a simple, intuitive pattern, consisting of only the current timestamp and a counter value. An

example message ID is shown below:

$$\underbrace{1616604187}_{\text{Timestamp}}\underbrace{0000600}_{\text{Counter}}$$

The first 10 digits represent the UNIX epoch timestamp of the message in seconds, and the last 7 digits is a counter that gets increased for each consecutive message and resets to 0 after approximately 5 days of inactivity. We conducted a series of controlled experiments and empirically found that the counter increments according to the following rules:

1. The increment between two consecutive messages is always a multiple of 100. Although this increment is usually 200, it may change based on the user actions listed in Fig. A.1.

2. The counters are independent across different channels, as well as user actions in different channels.

Due to the first rule, the attacker cannot predict the exact message ID given the previous ID, as Slack does not provide a way to learn how many drafts are saved internally. However, if the attacker is given two valid IDs separated by a small time interval, then it is straightforward to guess the valid IDs in between. We describe two ways of learning a valid ID. The first way is, again, to rely on the `groups:read` scope, since the metadata of the channel includes the ID of the latest message in the channel. The second way is to write a new message to the channel, which will cause the Slack API to return the ID of the newly posted message.

Attack workflow.

1. The attacker obtains a valid combination of channel ID and message ID using the techniques described above. We refer to the message ID as (t_0, c_0). If it obtains the message ID via posting new messages,

User Action	Counter Increment
User Posting a message with text	200
App Posting a message with text	100
Posting a message with only file	100
Saving a draft (happens automatically 10 seconds after the user stops typing)	100

Figure A.1: Slack Message Counter Increment. For each consecutive message, the counter value is increased by 100x, where x starts at 0 and gradually increases based on actions of the users in the channel.

then it immediately deletes the message to hide its trace, which is also permitted by the `chat:write` scope.

2. After a short time τ, the attacker obtains another valid message ID $(t_0 + \tau, c_1)$.

3. The attacker guesses all possible message IDs, which is the cartesian product of $(t_0, t_0 + 1, ..., t_0 + \tau)$ and $(c_0 + 100, c_0 + 200, ..., c_1 - 100)$.

4. The attacker uses the guessed IDs to generate the message URL and posts it to the user's personal channel. The URLs of the valid IDs will get unfurled.

By repeating this attack over and over again for different message IDs, the attacker can eventually pull every message from any private channel that the victim user has joined, effectively granting the malicious app the power of the `groups:history` scope even though this scope is never explicitly requested. We note that the attacker should adjust the time interval τ dynamically based on the messaging frequency to aim for $c_1 - c_0 \leqslant 500$, so that it can post all possible IDs in step 3 under Slack's rate limit (which allows unfurling of up to 5 URLs per second).

A.2 Implementing Supported Function in eTAP

In this appendix section, we describe how to implement each operation that appears in Fig. 3.3, except for Boolean and arithmetic operations, since existing GC frameworks like EMP toolkit [191] already provide built-in functions to efficiently translate them.

- `x == s` and `x.startwith(s)`. A bit-wise comparison between `x` and `s` is performed up to the `min(len(x), len(s))` bit, and results are feed into a large AND gate as output. For `x == s`, We additionally check if the next remaining character in `x` or `s` is a padding character.

- `x.endwith(s)` and `x.contain(s)`. These functions need to be first converted to a correspondingly regular expression and then matched against `x`.

- `x.replace(s, t)`. We can apply the DFA replacement technique described in Section 3.5 directly for this type of functions.

- `x.extract_phone()` and `extract_email()`. We apply the DFA extraction described in Section 3.5 by constructing appropriate regular expressions. However, as we need the matching results to be non-overlapping, one modification is needed : we can append `[^a-Z0-9]` to the regular expression and shift the final matching position forward by 1 character.

- `x.split(d,i)`. Without loss of generality, we assume `d` is a single character. First we need to create two regular expressions, Γ_1 and Γ_2, to that output accepting states when the `i`-th and `i+1`-th occurrences of `d` is encountered. Once we have the starting and ending location of the substring, we can proceed with substring extractions.

- `x.truncate(n)`. We can keep a variable counter `c` that gets increased after each bit in `x` is processed. And each output bit `y[i]` is computed by `x[i] & (n > c)`.
- `x.tolowercase()`. Assume ASCII encoding, for each character in `x`, we first check if the last five bits are in the valid ranges; if so, we flip the sixth bit.
- `m.lookup(x)`. First, we compare `x` with each key of `m`, and store the matching results into a `len(m)`-bit sequence. We denote this sequence as `b`. Then, the output `y` is computed iteratively by `y = (b[i] & v[i]) | (!b[i] & y)` as `i` ranges from 1 to `len(m)`.

A.3 Attribute Category Criteria in minTAP

We list below the detailed criteria for how we determine which category each attribute belongs to in our evaluation of minTAP's privacy benefits (Section 4.8). We note that there are overlappings between different categories. For example, an attribute named `LocationMapImageUrl` counts as both location and downloadable link. However, we use the third criteria to ensure there are no overlappings between categories of different sensitivity levels.

Timestamp. In IFTTT, attributes representing timestamps are conventionally named in the format of *xxxx*At, such as OccurredAt or CreatedAt. Other common attribute names include Date and Time.

Link. For all attributes we inspected of this category, their names include either Link or Url. Furthermore, we found that, out of these attributes, the ones whose links are access-controlled (i.e. user's login credentials are required to access the information) are usually named as PublicUrl or EventUrl. On the contrary, links that can be used to directly download files often start with one of the following keywords in their names: Image, File,

Video, Download, Record, Document, Mp3, Photo, Audio, Picture, Share, and Source.

Location. Location attributes contain one of these keywords: Location, Longitude, Latitude, Where, and Address.

User Info. Attributes in this category reveal information about the user, including the user's real name (FullName), online identity (Username, User, Member), and their contact information (Contact, Email, Number, From, To).

Event description. Attributes in this category provide descriptive texts to the trigger event, including ProjectName, TaskName, EventName, Description, About, Note, Title, Tag, Summary, HTML, Section, Field, Column, Row, Caption, FirstLinkUrl, and EmbeddedCode.

Message. Attributes pertain to a text message or an email includes Message, Body, Text, Content, and Subject.

Bookmark. Attributes related to a article or webpage bookmarked by the user usually includes the keywords Article, Website, or Page.

Other. Other attributes that we found containing sensitive information include financial information (Transaction, Money, Payment, Amount), smart home (Temp, Pm, Co2, Humidity, Indoor, Air, Concentration, Device, Sensor, Camera,Thermostat, Switch, Doorbell, Home, EnterOrExited), event duration (Ends, Duration), and reminder lists (List).

BIBLIOGRAPHY

[1] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html.

[2] Alexa Developer Documentation. https://developer.amazon.com/en-US/docs/alexa/documentation-home.html.

[3] Automatically save in Pocket the first link in a Tweet you like. https://ifttt.com/applets/DfUKrQkt.

[4] Bitbucket. https://appsource.microsoft.com/en-us/product/office/WA200002405.

[5] Chatlio. https://slack.com/apps/A03BS4Q25.

[6] Cryptography.io. https://cryptography.io.

[7] Dokkio. https://slack.com/apps/AJ4H0RRBJ.

[8] Dropbox. https://slack.com/apps/AES7B2V7D.

[9] Get a notification 15 minutes before your next GCal event starts. https://ifttt.com/applets/cFX5ETAs.

[10] Google calendar. https://slack.com/apps/ADZ494LHY.

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

[12] Lucidcharts. https://appsource.microsoft.com/en-us/product/office/WA104381935.

[13] Mailclark. https://slack.com/apps/A0JUW1X96.

[14] Payments on Square send you a phone call. https://ifttt.com/applets/TafsT2nY.

[15] Ziri. https://slack.com/apps/A8256N5BK.

[16] Zoom. https://slack.com/apps/A5GE9BMQC.

[17] Proof Key for Code Exchange by OAuth Public Clients. https://tools.ietf.org/html/rfc7636, 2015.

[18] 2017 Cybersecurity Incident & important consumer information-list of data breaches. https://www.equifaxsecurity2017.com/consumer-notice/, 2017.

[19] Business Chat Apps in 2018: Top Players and Adoption Plans. https://community.spiceworks.com/blog/3157-business-chat-apps-in-2018-top-players-and-adoption-plans, 2018.

[20] Be more productive. Automatically. https://flow.microsoft.com/en-us/, 2020.

[21] Building with filter code. https://help.ifttt.com/hc/en-us/articles/360052451954-Building-with-filter-code, 2020.

[22] California Privacy Rights Act (CPRA). https://oag.ca.gov/privacy/, November 2020.

[23] Creating applets - ifttt platform. https://platform.ifttt.com/docs/applets#creating-applets, 2020.

[24] Filter by zapier integrations. https://zapier.com/apps/filter/integrations, 2020.

[25] Formatter by zapier integrations. https://zapier.com/apps/formatter/integrations, 2020.

[26] IFTTT: If This Then That. https://ifttt.com, 2020.

[27] Privacy guidance for manufacturers of Internet of Things devices. https://www.priv.gc.ca/en/privacy-topics/technology/gd_iot_man, 2020.

[28] 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, 2020.

[29] Service API Requirements - IFTTT. https://ifttt.com/docs/api_reference#triggers, 2020.

[30] Service api requirements - ifttt platform. https://platform.ifttt.com/docs/api_reference#service-authentication, 2020.

[31] Tfhe: Fast fully homomorphic encryption library over the torus. https://github.com/tfhe/tfhe, 2020.

[32] Zapier: Connect your apps and automate workflows. https://zapier.com/home, 2020.

[33] Add Apps to Slack | Apps and Integrations | Slack App Directory. https://slack.com/apps, 2021.

[34] Apache OpenWhisk is a serverless, open source cloud platform. https://openwhisk.apache.org, 2021.

[35] Best practices for security | Slack. https://api.slack.com/authentication/best-practices, 2021.

[36] Business Apps - Microsoft AppSource. https://appsource.microsoft.com/en-us/marketplace/apps, 2021.

[37] Node-RED. https://nodered.org, 2021.

[38] Slack quickly removes message invites in its new DM feature over harassment concerns . https://www.theverge.com/2021/3/24/22348743/slack-connect-dm-abuse-harassment-disable-message-invite-response, 2021.

[39] That Slack email you just got asking to reset your password is legit, not a scam. https://www.androidpolice.com/2021/02/05/that-slack-email-you-just-got-asking-to-reset-your-password-is-legit-not-a-scam/, 2021.

[40] Microsoft Teams store validation guidelines. https://docs.microsoft.com/en-us/microsoftteams/platform/concepts/deploy-and-publish/appsource/prepare/teams-store-validation-guidelines, 2022.

[41] Teams + Power Automate. https://powerautomate.microsoft.com/en-US/connectors/details/shared_teams/microsoft-teams/, 2022.

[42] Abbas Acar, Hossein Fereidooni, Tigist Abera, Amit Kumar Sikder, Markus Miettinen, Hidayet Aksu, Mauro Conti, Ahmad-Reza Sadeghi, and Selcuk Uluagac. 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*, pages 207–218, 2020.

[43] Mohammad M. Ahmadpanah, Daniel Hedin, Musard Balliu, Lars Eric Olsson, and Andrei Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security Symposium*, 2021.

[44] Abdulaziz Alhadlaq, Jun Tang, Marwan Almaymoni, and Aleksandra Korolova. Privacy in the amazon alexa skills ecosystem. *Star*, 217(11), 2017.

[45] Mohannad Alhanahnah, Clay Stevens, and Hamid Bagheri. Scalable analysis of interaction threats in iot systems. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 272–285, New York, NY, USA, 2020. Association for Computing Machinery.

[46] Joël Alwen, Jonathan Katz, Yehuda Lindell, Giuseppe Persiano, abhi shelat, and Ivan Visconti. Collusion-free multiparty computation in the mediated model. In Shai Halevi, editor, *Advances in Cryptology - CRYPTO 2009*, pages 524–540, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[47] Joël Alwen, Abhi Shelat, and Ivan Visconti. Collusion-free protocols in the mediated model. In David Wagner, editor, *Advances in Cryptology – CRYPTO 2008*, pages 497–514, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[48] Amazon. Pricing for Elastic Block Storage. https://aws.amazon.com/ebs/pricing/, 2020.

[49] Nicolas Anciaux, Walid Bezza, Benjamin Nguyen, and Michalis Vazirgiannis. Minexp-card: limiting data collection using a smart card. In *EDBT*, pages 753–756. ACM, 2013.

[50] Nicolas Anciaux, Danae Boutara, Benjamin Nguyen, and Michalis Vazirgiannis. Limiting data exposure in multi-label classification processes. *Fundam. Informaticae*, 137(2):219–236, 2015.

[51] Nicolas Anciaux, Benjamin Nguyen, and Michalis Vazirgiannis. Limiting data collection in application forms: A real-case application of a founding privacy principle. In *PST*, pages 59–66. IEEE Computer Society, 2012.

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

[53] Pauline Anthonysamy, Awais Rashid, James Walkerdine, Phil Greenwood, and Georgios Larkou. Collaborative privacy management for third-party applications in online social networks. In *Proceedings of the 1st Workshop on Privacy and Security in Online Social Media*, pages 1–4, 2012.

[54] Thibaud Antignac, David Sands, and Gerardo Schneider. Data minimisation: A language-based approach. In *SEC*, volume 502 of *IFIP Advances in Information and Communication Technology*, pages 442–456. Springer, 2017.

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

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

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

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

[59] Musard Balliu, Iulia Bastys, and Andrei Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.

[60] Iulia Bastys, Musard Balliu, and Andrei Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *ACM Conference on Computer and Communications Security (CCS)*, 2018.

[61] D. Beaver, S. Micali, and P. Rogaway. The round complexity of secure protocols. In *Proceedings of the Twenty-Second Annual ACM Symposium on Theory of Computing*, STOC '90, pages 503–513, New York, NY, USA, 1990. Association for Computing Machinery.

[62] Mira Belenkiy, Jan Camenisch, Melissa Chase, Markulf Kohlweiss, Anna Lysyanskaya, and Hovav Shacham. 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*, pages 108–125. Springer, 2009.

[63] M. Bellare and C. Namprempre. Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Advances in Cryptology – ASIACRYPT 2000*, pages 531–545. Springer, 2000.

[64] Mihir Bellare, Viet Tung Hoang, and Phillip Rogaway. Foundations of garbled circuits. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 784–796, 2012.

[65] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, CCS '93, pages 62–73, New York, NY, USA, 1993. Association for Computing Machinery.

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

[67] Julien Bringer, Mélanie Favre, Hervé Chabanne, and Alain Patey. Faster secure computation for biometric identification using filtering. In *2012 5th IAPR International Conference on Biometrics (ICB)*, pages 257–264. IEEE, 2012.

[68] Sven Bugiel, Lucas Davi, Alexandra Dmitrienko, Thomas Fischer, Ahmad-Reza Sadeghi, and Bhargava Shastry. Towards taming privilege-escalation attacks on android. In *NDSS*, volume 17, page 19. Citeseer, 2012.

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

[70] Henry Carter, Benjamin Mood, Patrick Traynor, and Kevin Butler. Secure outsourced garbled circuit evaluation for mobile devices. *Journal of Computer Security*, 24(2):137–180, 2016.

[71] Z Berkay Celik, Leonardo Babun, Amit Kumar Sikder, Hidayet Aksu, Gang Tan, Patrick McDaniel, and A Selcuk Uluagac. Sensitive information tracking in commodity iot. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 1687–1704, 2018.

[72] Z. Berkay Celik, Earlence Fernandes, Eric Pauley, Gang Tan, and Patrick D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Computing Surveys*, 2019.

[73] Z Berkay Celik, Patrick McDaniel, and Gang Tan. Soteria: Automated iot safety and security analysis. In *2018 {USENIX} Annual Technical Conference ({USENIX}{ATC} 18)*, pages 147–158, 2018.

[74] Abdelberi Chaabane, Yuan Ding, Ratan Dey, Mohamed Ali Kaafar, and Keith W Ross. A closer look at third-party osn applications: Are they leaking your personal information? In *International conference on passive and active network measurement*, pages 235–246. Springer, 2014.

[75] Chia-Hsiang Chang and Robert Paige. From regular expressions to dfa's using compressed nfa's. In *Annual Symposium on Combinatorial Pattern Matching*, pages 90–110. Springer, 1992.

[76] Eric Y Chen, Yutong Pei, Shuo Chen, Yuan Tian, Robert Kotcher, and Patrick Tague. Oauth demystified for mobile application developers. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 892–903, 2014.

[77] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. Sgxpectre: Stealing intel secrets from sgx enclaves via speculative execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*, pages 142–157, 2019.

[78] Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. Practical data access minimization in trigger-action platforms. In *31st USENIX Security Symposium (**USENIX Security**)*, 2022.

[79] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. Data privacy in trigger-action systems. In *42nd IEEE Symposium on Security and Privacy (**IEEE S&P**)*.

[80] Yunang Chen, Amrita Roy Chowdhury, Ruizhe Wang, Andrei Sabelfeld, Rahul Chatterjee, and Earlence Fernandes. Data Privacy in Trigger-Action Systems. In *IEEE Symposium on Security and Privacy*, 2021.

[81] Yunang Chen, Yue Gao, Nick Ceccio, Rahul Chatterjee, Kassem Fawaz, and Earlence Fernandes. Experimental security analysis of the app model in business collaboration platforms. In *31st USENIX Security Symposium (**USENIX Security**)*, 2022.

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

[83] Yuan Cheng, Jaehong Park, and Ravi Sandhu. Preserving user privacy from third-party applications in online social networks. In *Proceedings of the 22nd International Conference on World Wide Web*, pages 723–728, 2013.

[84] Yu-Hsi Chiang, Hsu-Chun Hsiao, Chia-Mu Yu, and Tiffany Hyun-Jin Kim. On the privacy risks of compromised trigger-action platforms. In *ESORICS*, 2020.

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

[86] Catalin Cimpanu. Millions of exim servers vulnerable to root-granting exploit. https://www.zdnet.com/article/millions-of-exim-servers-vulnerable-to-root-granting-exploit/, 2019.

[87] Catalin Cimpanu. Stack overflow hacker went undetected for a week. https://www.zdnet.com/article/stack-overflow-hacker-went-undetected-for-a-week/, 2019.

[88] Dwaine Clarke, Jean-Emile Elien, Carl Ellison, Matt Fredette, Alexander Morcos, and Ronald L Rivest. Certificate chain discovery in spki/sdsi. *Journal of Computer security*, 9(4):285–322, 2001.

[89] Camille Cobb, Milijana Surbatovich, Anna Kawakami, Mahmood Sharif, Lujo Bauer, Anupam Das, and Limin Jia. How risky are real users'{IFTTT} applets? In *Sixteenth Symposium on Usable Privacy and Security ({SOUPS} 2020)*, pages 505–529, 2020.

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

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

[92] Sam Costello. How much video can you record on an iphone? https://www.lifewire.com/how-much-video-can-iphone-record-2000304, 2020. Accessed: 2020-06-02.

[93] George Danezis and Andrei Serjantov. 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*, pages 293–308. Springer, 2005.

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

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

[96] Lucas Davi, Alexandra Dmitrienko, Ahmad-Reza Sadeghi, and Marcel Winandy. Privilege escalation attacks on android. In *international conference on Information security*, pages 346–360. Springer, 2010.

[97] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM '04, page 21, USA, 2004. USENIX Association.

[98] Shuaike Dong, Zhou Li, Di Tang, Jiongyi Chen, Menghan Sun, and Kehuan Zhang. 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*, pages 47–59, 2020.

[99] William Enck, Peter Gilbert, Seungyeop Han, Vasant Tendulkar, Byung-Gon Chun, Landon P Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N Sheth. Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones. *ACM Transactions on Computer Systems (TOCS)*, 32(2):1–29, 2014.

[100] Saba Eskandarian, Henry Corrigan-Gibbs, Matei Zaharia, Dan Boneh, et al. Express: Lowering the cost of metadata-hiding communication with cryptographic privacy. In *USENIX Security Symposium*, pages 1775–1792, 2021.

[101] Earlence Fernandes, Jaeyeon Jung, and Atul Prakash. Security analysis of emerging smart home applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 636–654. IEEE, 2016.

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

[103] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202. ACM, 2002.

[104] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*, pages 377–396. IEEE, 2016.

[105] Tore Kasper Frederiksen, Thomas P Jakobsen, and Jesper Buus Nielsen. Faster maliciously secure two-party computation using the gpu. In *International Conference on Security and Cryptography for Networks*, pages 358–379. Springer, 2014.

[106] Gayathri Garimella, Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 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*, pages 395–425. Springer, 2021.

[107] General Data Protection Regulation, EU Regulation 2016/679, 2018.

[108] Irene Giacomelli, Somesh Jha, Marc Joye, C. David Page, and Kyonghwan Yoon. Privacy-preserving ridge regression with only linearly-homomorphic encryption. In Bart Preneel and Frederik Vercauteren, editors, *Applied Cryptography and Network Security*, pages 243–261, Cham, 2018. Springer International Publishing.

[109] Joseph Goguen and José Meseguer. Security policies and security models. In *IEEE S&P*, 1982.

[110] O. Goldreich, S. Micali, and A. Wigderson. How to play any mental game. In *Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing*, STOC '87, pages 218–229, New York, NY, USA, 1987. Association for Computing Machinery.

[111] Don Goodman-Wilson. Bot-to-bot communication models for slack, Sep 2016.

[112] Michael I Gordon, Deokhwan Kim, Jeff H Perkins, Limei Gilham, Nguyen Nguyen, and Martin C Rinard. Information flow analysis of android applications in droidsafe. In *NDSS*, volume 15, page 110, 2015.

[113] Zhixiu Guo, Zijin Lin, Pan Li, and Kai Chen. Skillexplorer: Understanding the behavior of skills in large scale. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 2649–2666, 2020.

[114] A. Hern. Uber employees 'spied on ex-partners, politicians and beyoncé', 2016. https://www.theguardian.com/technology/2016/dec/13/uber-employees-spying-ex-partners-politicians-beyonce.

[115] Grant Ho, Derek Leung, Pratyush Mishra, Ashkan Hosseini, Dawn Song, and David Wagner. Smart locks: Lessons for securing commodity internet of things devices. In *Proceedings of the 11th ACM on Asia conference on computer and communications security*, pages 461–472, 2016.

[116] Sean Hollister. The iphone 6s camera is a huge storage hog (but it might be worth it). https://www.cnet.com/news/iphone-6s-camera-filesizes-4k-live-photos-hdr/, 2015. Accessed: 2020-11-23.

[117] Yan Huang, David Evans, Jonathan Katz, and Lior Malka. Faster secure two-party computation using garbled circuits. In *USENIX Security Symposium*. USENIX Association, 2011.

[118] IFTTT. How people use IFTTT today. https://ifttt.com/blog/2016/11/connected-life-of-an-ifttt-user, 2016.

[119] IFTTT. Applets Cookbook. https://platform.ifttt.com/docs/applets#applets-cookbook, 2018.

[120] IFTTT. Terms of Use. https://ifttt.com/terms, 2018.

[121] IFTTT. IFTTT: Creating Applets. https://platform.ifttt.com/docs/applets, 2020.

[122] IFTTT. IFTTT: Number of Users and Online Services. https://platform.ifttt.com/plans, 2020.

[123] IFTTT. IFTTT: Service API requirements. https://platform.ifttt.com/docs/api_reference, 2020.

[124] IFTTT. IFTTT's Glossary: Query. https://platform.ifttt.com/docs/glossary#query, 2020.

[125] Jinseong Jeon, Kristopher K Micinski, Jeffrey A Vaughan, Ari Fogel, Nikhilesh Reddy, Jeffrey S Foster, and Todd Millstein. Dr. android and mr. hide: fine-grained permissions in android applications. In *Proceedings of the second ACM workshop on Security and privacy in smartphones and mobile devices*, pages 3–14, 2012.

[126] Yizhen Jia, Yinhao Xiao, Jiguo Yu, Xiuzhen Cheng, Zhenkai Liang, and Zhiguo Wan. A novel graph-based mechanism for identifying traffic vulnerabilities in smart home iot. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 1493–1501. IEEE, 2018.

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

[128] Donald E Knuth, James H Morris, Jr, and Vaughan R Pratt. Fast pattern matching in strings. *SIAM journal on computing*, 6(2):323–350, 1977.

[129] Vladimir Kolesnikov, Ahmad-Reza Sadeghi, and Thomas Schneider. A systematic approach to practically efficient general two-party secure function evaluation protocols and their modular design. *Journal of Computer Security*, 21(2):283–315, 2013.

[130] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free xor gates and applications. In *International Colloquium on Automata, Languages, and Programming*, pages 486–498. Springer, 2008.

[131] Vladimir Kolesnikov and Thomas Schneider. Improved garbled circuit: Free XOR gates and applications. In *ICALP (2)*, volume 5126 of *Lecture Notes in Computer Science*, pages 486–498. Springer, 2008.

[132] H. Krawczyk, M. Bellare, and R. Canetti. Rfc2104: Hmac: Keyed-hashing for message authentication, 1997.

[133] Benjamin Kreuter, Abhi Shelat, and Chih-Hao Shen. Billion-gate secure computation with malicious adversaries. In *USENIX Security Symposium*, pages 285–300. USENIX Association, 2012.

[134] Deepak Kumar, Riccardo Paccagnella, Paul Murley, Eric Hennenfent, Joshua Mason, Adam Bates, and Michael Bailey. Skill squatting attacks on amazon alexa. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 33–47, 2018.

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

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

[137] Marcel Laverdet. Secure & Isolated JS Environments for Node.js. https://github.com/laverdet/isolated-vm, 2020.

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

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

[140] D. Lee. Uber concealed huge data breach, 2017. http://www.bbc.com/news/technology-42075306.

[141] Christopher Lentzsch, Sheel Jayesh Shah, Benjamin Andow, Martin Degeling, Anupam Das, and William Enck. Hey alexa, is this skill safe?: Taking a closer look at the alexa skill ecosystem. In *28th Annual Network and Distributed System Security Symposium (NDSS 2021). The Internet Society*, 2021.

[142] Yehuda Lindell, Benny Pinkas, and Nigel P Smart. Implementing two-party computation efficiently with security against malicious adversaries. In *International Conference on Security and Cryptography for Networks*, pages 2–20. Springer, 2008.

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

[144] Blake Loring, Duncan Mitchell, and Johannes Kinder. Expose: practical symbolic execution of standalone javascript. In *Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software*, pages 196–199, 2017.

[145] Haoran Lu, Luyi Xing, Yue Xiao, Yifan Zhang, Xiaojing Liao, XiaoFeng Wang, and Xueqiang Wang. Demystifying resource management risks in emerging mobile app-in-app ecosystems. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 569–585, 2020.

[146] Ingrid Lunden. IFTTT raises $24M led by Salesforce to expand its platform to 'connect everything'. https://techcrunch.com/2018/04/26/ifttt-raises-24m-led-by-salesforce-to-expand-its-platform-to-connect-everything/, 2018.

[147] Claudio Marforio, Aurélien Francillon, and Srdjan Capkun. Application collusion attack on the permission-based security model and its implications for modern smartphone systems. Technical report, ETH Zurich, 2011.

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

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

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

[151] Payman Mohassel. A closer look at anonymity and robustness in encryption schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 501–518. Springer, 2010.

[152] Payman Mohassel, Salman Niksefat, Saeed Sadeghian, and Babak Sadeghiyan. An efficient protocol for oblivious dfa evaluation and applications. In *Cryptographers' Track at the RSA Conference*, pages 398–415. Springer, 2012.

[153] Anders Møller. dk.brics.automaton – finite-state automata and regular expressions for Java, 2017. `http://www.brics.dk/automaton/`.

[154] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *S&P*, 2020.

[155] Michael Naehrig, Kristin Lauter, and Vinod Vaikuntanathan. Can homomorphic encryption be practical? In *Proceedings of the 3rd ACM Workshop on Cloud Computing Security Workshop*, CCSW '11, pages 113–124, New York, NY, USA, 2011. Association for Computing Machinery.

[156] Moni Naor, Benny Pinkas, and Reuban Sumner. Privacy preserving auctions and mechanism design. In *EC*, 1999.

[157] NIST. SHA-3 standard: Permutation-based hash and extendable-output functions. 2015.

[158] Oauth 2.0. https://oauth.net/2/, 2018.

[159] Danny Palmer. These hackers broke into 10 telecoms companies to steal customers' phone records. https://www.zdnet.com/article/these-hackers-broke-into-10-telecoms-companies-to-steal-customers-phone-records/, 2019.

[160] Andrea Peterson. eBay asks 145 million users to change passwords after data breach. https://www.washingtonpost.com/news/the-switch/wp/2014/05/21/ebay-asks-145-million-users-to-change-passwords-after-data-breach/, 2014.

[161] Srinivas Pinisetty, Thibaud Antignac, David Sands, and Gerardo Schneider. Monitoring data minimisation. *CoRR*, abs/1801.02484, 2018.

[162] Benny Pinkas, Mike Rosulek, Ni Trieu, and Avishay Yanai. 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*, pages 739–767. Springer, 2020.

[163] Jon Porter. Elevating trust in our api ecosystem. https://developers.googleblog.com/2018/10/elevating-user-trust-in-our-api.html, 2018.

[164] Michael O. Rabin. How to exchange secrets with oblivious transfer. Cryptology ePrint Archive, Report 2005/187, 2005. https://eprint.iacr.org/2005/187.

[165] Reuters. Database of 191 million U.S. voters exposed on Internet. https://www.reuters.com/article/us-usa-voters-breach-idUSKBN0UB1E020151229, 2015.

[166] Peter Rindal and Mike Rosulek. Faster malicious 2-party secure computation with online/offline dual execution. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 297–314, 2016.

[167] Nicky Robinson and Joseph Bonneau. Cognitive disconnect: understanding facebook connect login permissions. In *Proceedings of the second ACM conference on Online social networks*, pages 247–258, 2014.

[168] Amrita Roy Chowdhury, Chenghong Wang, Xi He, Ashwin Machanavajjhala, and Somesh Jha. Cryptε: Crypto-assisted differential privacy on untrusted servers. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 603–619. ACM, 2020.

[169] Jerome H Saltzer and Michael D Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[170] Jerome H. Saltzer and Michael D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[171] Roman Schlegel, Kehuan Zhang, Xiao-yong Zhou, Mehool Intwala, Apu Kapadia, and XiaoFeng Wang. Soundcomber: A stealthy and context-aware sound trojan for smartphones. In *NDSS*, volume 11, pages 17–33, 2011.

[172] Sandy Schoettler, Andrew Thompson, Rakshith Gopalakrishna, and Trinabh Gupta. Walnut: A low-trust trigger-action platform, 2020. https://arxiv.org/pdf/2009.12447.pdf.

[173] Siamak F Shahandashti and Reihaneh Safavi-Naini. 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*, pages 198–216. Springer, 2009.

[174] Mohamed Shehab, Anna Cinzia Squicciarini, and Gail-Joon Ahn. Beyond user-to-user access control for online social networks. In *International Conference on Information and Communications Security*, pages 174–189. Springer, 2008.

[175] Kapil Singh, Sumeer Bhola, and Wenke Lee. xbook: Redesigning privacy control in social networking platforms. In *USENIX Security Symposium*, pages 249–266, 2009.

[176] Ebrahim M Songhori, Siam U Hussain, Ahmad-Reza Sadeghi, Thomas Schneider, and Farinaz Koushanfar. Tinygarble: Highly compressed and scalable sequential garbled circuits. In *2015 IEEE Symposium on Security and Privacy*, pages 411–428. IEEE, 2015.

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

[178] Dan Su, Jiqiang Liu, Sencun Zhu, Xiaoyang Wang, and Wei Wang. " are you home alone?"" yes" disclosing security and privacy vulnerabilities in alexa skills. *arXiv preprint arXiv:2010.10788*, 2020.

[179] San-Tsai Sun and Konstantin Beznosov. The devil is in the (implementation) details: an empirical analysis of oauth sso systems. In *Proceedings of the 2012 ACM conference on Computer and communications security*, pages 378–390, 2012.

[180] Milijana Surbatovich, Jassim Aljuraidan, Lujo Bauer, Anupam Das, and Limin Jia. Some recipes can do more than spoil your appetite: Analyzing the security and privacy risks of IFTTT recipes. In *WWW*, 2017.

[181] Iraklis Symeonidis, Gergely Biczók, Fatemeh Shirazi, Cristina Pérez-Solà, Jessica Schroers, and Bart Preneel. Collateral damage of facebook third-party applications: a comprehensive study. *Computers & Security*, 77:179–208, 2018.

[182] Sarath Tomy and Eric Pardede. Controlling privacy disclosure of third party applications in online social networks. *International Journal of Web Information Systems*, 2016.

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

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

[185] Blase Ur, Melwyn Pak Yong Ho, Stephen Brawner, Jiyun Lee, Sarah Mennicken, Noah Picard, Diane Schulze, and Michael L Littman. Trigger-action programming in the wild: An analysis of 200,000 ifttt recipes. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, pages 3227–3231, 2016.

[186] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. Foreshadow: Extracting the keys to the Intel SGX kingdom with transient out-of-order execution. In *Proceedings of the 27th USENIX Security Symposium*. USENIX Association, August 2018.

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

[188] Bimal Viswanath, Emre Kiciman, and Stefan Saroiu. Keeping information safe from social networking apps. In *Proceedings of the 2012 ACM workshop on Workshop on online social networks*, pages 49–54, 2012.

[189] Hui Wang, Yuanyuan Zhang, Juanru Li, Hui Liu, Wenbo Yang, Bodong Li, and Dawu Gu. Vulnerability assessment of oauth implementations in android applications. In *Proceedings of the 31st annual computer security applications conference*, pages 61–70, 2015.

[190] Qi Wang, Pubali Datta, Wei Yang, Si Liu, Adam Bates, and Carl A. Gunter. Charting the attack surface of trigger-action iot platforms. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, pages 1439–1453, New York, NY, USA, 2019. Association for Computing Machinery.

[191] Xiao Wang, Alex J. Malozemoff, and Jonathan Katz. EMP-toolkit: Efficient MultiParty computation toolkit. https://github.com/emp-toolkit, 2016.

[192] Fengguo Wei, Sankardas Roy, and Xinming Ou. Amandroid: A precise and general inter-component data flow analysis framework for security vetting of android apps. In *Proceedings of the 2014 ACM SIGSAC conference on computer and communications security*, pages 1329–1341, 2014.

[193] Wired. Hack Brief: 4-Year-Old Dropbox Hack Exposed 68 Million People's Data. https://www.wired.com/2016/08/hack-brief-four-year-old-dropbox-hack-exposed-68-million-peoples-data/, 2016.

[194] David Isaac Wolinsky, Henry Corrigan-Gibbs, Bryan Ford, and Aaron Johnson. Dissent in numbers: Making strong anonymity scale. In *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, pages 179–182, 2012.

[195] David Isaac Wolinsky, Ewa Syta, and Bryan Ford. Hang with your buddies to resist intersection attacks. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 1153–1166, 2013.

[196] Michelle Y Wong and David Lie. Intellidroid: A targeted input generator for the dynamic analysis of android malware. In *NDSS*, volume 16, pages 21–24, 2016.

[197] Yinglian Xie, Fang Yu, Kannan Achan, Eliot Gillum, Moises Goldszmidt, and Ted Wobber. How dynamic are ip addresses? In *Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 301–312, 2007.

[198] Rixin Xu, Qiang Zeng, Liehuang Zhu, Haotian Chi, Xiaojiang Du, and Mohsen Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 7:63457–63471, 2019.

[199] Andrew C Yao. Protocols for secure computations. In *23rd annual symposium on foundations of computer science (sfcs 1982)*, pages 160–164. IEEE, 1982.

[200] Andrew Chi-Chih Yao. How to generate and exchange secrets. In *27th Annual Symposium on Foundations of Computer Science (sfcs 1986)*, pages 162–167. IEEE, 1986.

[201] F.F. Yao and Y.L. Yin. Design and analysis of password-based key derivation functions. In *Topics in Cryptology – CT-RSA 2005*, pages 245–261. Springer, 2005.

[202] Keyang Yu, Qi Li, Dong Chen, Mohammad Rahman, and Shiqiang Wang. 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)*, pages 62–76, 2021.

[203] Bin Yuan, Yan Jia, Luyi Xing, Dongfang Zhao, XiaoFeng Wang, and Yuqing Zhang. Shattered chain of trust: Understanding security risks in cross-cloud iot access delegation. In *29th {USENIX} Security Symposium ({USENIX} Security 20)*, pages 1183–1200, 2020.

[204] Samee Zahur, Mike Rosulek, and David Evans. Two halves make a whole. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 220–250. Springer, 2015.

[205] Zapier. Zapier Platform CLI Docs. https://platform.zapier.com/cli_docs/docs, 2020.

[206] I. Zavalyshyn, N. Santos, R. Sadre, and A. Legay. My House, My Rules: A Private-by-Design Smart Home Platform. In *EAI MobiQuitous*, 2020.

[207] Nan Zhang, Xianghang Mi, Xuan Feng, XiaoFeng Wang, Yuan Tian, and Feng Qian. Dangerous skills: Understanding and mitigating security risks of voice-controlled third-party functions on virtual personal assistant systems. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1381–1396. IEEE, 2019.

[208] Wei Zhang, Yan Meng, Yugeng Liu, Xiaokuan Zhang, Yinqian Zhang, and Haojin Zhu. Homonit: Monitoring smart home apps from encrypted traffic. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1074–1088, 2018.

[209] Serena Zheng, Noah Apthorpe, Marshini Chetty, and Nick Feamster. User perceptions of smart home iot privacy. *Proceedings of the ACM on human-computer interaction*, 2(CSCW):1–20, 2018.