Practical Data Access Minimization in Trigger-Action Platforms

Yunang Chen, Mohannad Alhanahnah, Andrei Sabelfeld[†]*, Rahul Chatterjee, Earlence Fernandes*
University of Wisconsin–Madison, USA [†]Chalmers University of Technology, Sweden

Abstract

Trigger-Action Platforms (TAPs) connect disparate online services and enable users to create automation rules in diverse domains such as smart homes and business productivity. Unfortunately, the current design of TAPs is flawed from a privacy perspective, allowing unfettered access to sensitive user data. We point out that it suffers from two types of overprivilege: (1) attribute-level, where it has access to more data attributes than it needs for running user-created rules; and (2) token-level, where it has access to more APIs than it needs. To mitigate overprivilege and subsequent privacy concerns we design and implement minTAP, a practical approach to data access minimization in TAPs. Our key insight is that the semantics of a user-created automation rule implicitly specifies the minimal amount of data it needs. This allows minTAP to leverage *language-based data minimization* to apply the principle of least-privilege by releasing only the necessary attributes of user data to TAPs and fending off unrelated API access. Using real user-created rules on the popular IFTTT TAP, we demonstrate that minTAP sanitizes a median of 4 sensitive data attributes per rule, with modest performance overhead and without modifying IFTTT.

1 Introduction

Trigger-action platforms (TAPs) enable millions of end-users to automate interactions between a wide variety of third-party services and devices ranging from cloud services to IoT devices and social networks [38]. Popular TAPs include IFTTT [9], Zapier [51], and Microsoft Power Automate [40]. End-users create simple automation *rules* using the trigger-action paradigm. Fig. 1 shows an example rule that uses the TAP to connect Outlook email with Slack — an email arriving at the user's inbox from `bank@xyz.com` will *trigger* 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).

With their widespread adoption, TAPs have unfortunately become overprivileged hubs of user information [19, 25, 50] raising privacy concerns [12, 18, 20, 48, 49]. Considering the rule in Fig. 1, current TAPs will receive all the emails from the specified sender, independently of whether emails are relevant to the user-created rule. Even worse, a malicious TAP can alter the rule parameters to obtain *any* email received by the user at *any* time. Furthermore, current TAP architectures dictate that the sensitive data of all users transits through a centralized web service, making it an attractive target for cloud attackers [28, 36]. Consequently, some third-party services (e.g., Gmail) have become reluctant to interface with TAPs citing privacy concerns [30]. At the same time, users are becoming wary of the insufficient safeguarding of their data by TAPs [24].

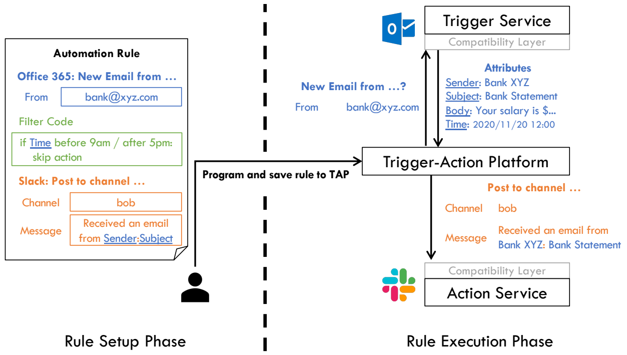


Figure 1: An example automation rule in trigger-action platforms. The boxed fields represent various information that the user needs to specify.

The core issue is that TAPs 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. [25] 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.

For our example rule in Fig. 1, 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 [44]. It can also create friction with legal frameworks like the General Data Protection Regulation (GDPR) [6] and the California Privacy Rights Act (CPRA) [8]. These frameworks stipulate *data minimization*, a principle restricting data collection to "what is necessary in relation to the purposes for which they are processed" [6].

Motivated by the above, we explore improving user data privacy on TAPs, a largely unexplored area in trigger-action platforms [18]. We design and implement 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* [17], 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 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 8).

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 [25], 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.

The paper offers the following contributions:

- We develop a general model for mitigating the privacy issues due to overprivilege (Section 5). Our system, minTAP, implements the principle of data minimization by using static and dynamic dependency analysis of TAP rules.
- We instantiate our model for IFTTT demonstrating how minTAP achieves practical data access minimization without changing IFTTT itself (Section 6).
- We have collected the first large-scale dataset of IFTTT rules with filter code and other detailed configurations. We evaluate the privacy benefits of minTAP on 34,419 rules. We use large-scale experiments with realistic workloads to show the performance impact is modest (Section 7).

Designing least-privilege systems often requires tailored solutions to strike a balance between functionality, security, and usability. We contribute to building least-privilege systems and showcase how one can practically adapt the recently-proposed theory of data minimization [17] through lightweight program analysis. Although we contextualize our work for TAPs, the principles are general and potentially useful for other types of API ecosystems. minTAP is available at https://github.com/EarlMadSec/minTAP.

2 Trigger-Action Platform Background

Trigger-action platforms (TAPs) connect disparate web services and enable end-users to build useful automation through a simple trigger-action paradigm. These web services range from digital resources like Dropbox, Gmail, and Slack to physical resources like IoT devices (e.g., ovens, smart door locks, smart lights). Web services can create *triggers* that will notify the TAP about an event (e.g., "new email arrived" or "door

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

Figure 2: Example filter code.

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. 1.

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. 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. 2 shows an example of filter code that sends a Slack message only when a new email with a subject containing the keyword "IFTTT" is received [7]. 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. Function `skip()` is used to skip an action.

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 [33, 52].

3 Data Privacy in Overprivileged Trigger-Action Platforms

Prior work [25] 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. 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. 3 showing that many sensitive attributes are being sent to IFTTT even though they are not required for rule execution. Recall that 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. 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 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. [25] 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 [6, 8] 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 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

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

IFTTT rule description	**Trigger**	**Trigger attributes** (unused ones shown in ***bold italics***)
Get notification before your next event starts [3]	Google Calendar: Any event starts	Title, ***Description***, ***Where***, Starts, ***Ends***, ***EventUrl***
Automatically save in Pocket the first link in a Tweet you like [2]	Twitter: New liked Tweet by you	***Text***, ***UserName***, ***LinkToTweet***, FirstLinkUrl, ***CreatedAt***, ***TweetEmbedCode***
Payments over ___ send you a phone call [4]	Square: New payments over a specific amount	***Merchant***, ID, TotalCollectedMoney, ***DeviceName***, ***PaymentAt***, ***RecordURL***

Figure 3: Examples of IFTTT rules, where several sensitive attributes of trigger data are not used by a rule but still sent.

under which we want to achieve these goals and then outline the design requirements of the solution. Finally, we discuss a few potential approaches and point out why they do not meet our security or functionality goals.

4.1 Threat Model

In line with prior work on security and privacy of TAPs [22, 23, 25, 45, 50, 53], 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 [23, 25], 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 [22, 23, 25]. 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.

4.2 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 [32]; (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.

4.3 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 (Section 5) and how we design minTAP to use minimizers without modifying IFTTT (Section 6).

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* [17]. 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 [17]. Prior work on building minimizers either resorts to verification [17] relying on manual intervention or testing value coverage [43] 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 7, filter code consists of small code snippets not written with adversarial intent [7]. 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.

Minimization by (in)dependency analysis. 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* [27]. 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.

We can find independence via static and dynamic program data-flow analysis techniques [42]. These techniques track whether a given input is used in computing the output of the rule. For the example rule from Fig. 1, by statically analyzing the rule, we infer that the body of the email is never used in the output. Therefore, the rule function is *independent* of the email body. A minimizer can thus withhold the email body (e.g., by replacing it with the empty string) without changing the rule's functionality. Similarly, when invoked outside the working hours, by dynamically analyzing a run of the rule in Fig. 1 we establish that the rule becomes *run-independent* of all inputs, in which case no data needs to be sent to the TAP.

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 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. 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

```
GenMinimizerInfo $(r = (T, A, f))$:
for $a_i \in A$ do
    $T_{a_i} \leftarrow \{t \mid t \in T \wedge t$ appears in $a_i\}$
$T_f \leftarrow \{\}$
for $stmt \in$ AST$(f)$ do
    $T_f \leftarrow T_f \cup \{t \mid t \in T \wedge t$ is accessed by $stmt\}$
$T' \leftarrow (T_{a_1}, \ldots, T_{a_n}, T_f)$
$f' \leftarrow$ transform$(f)$
Return $m = (T', f')$
```

```
SMinimizer $(\mathsf{D}_T,\ m = (T', f'))$:
/* $f'$ is not used */
$(T_{a_1}, \ldots, T_{a_n}, T_f) \leftarrow T'$
$T_a \leftarrow \bigcup_{a_i \in A} T_{a_i}$
for $(t, v) \in \mathsf{D}_T$ do
    if $t \notin (T_a \cup T_f)$ do
        $\mathsf{D}_T[t] \leftarrow \perp$
Return $\mathsf{D}_T$
```

```
DMinimizer $(\mathsf{D}_T,\ m = (T', f'))$:
$(T_{a_1}, \ldots, T_{a_n}, T_f) \leftarrow T'$
/* $T_f$ is not used */
$(T_{f'}, A') \leftarrow f'(\mathsf{D}_T)$
$T_a \leftarrow \bigcup_{a_i \in (A \setminus A')} T_{a_i}$
for $(t, v) \in \mathsf{D}_T$ do
    if $t \notin (T_a \cup T_{f'})$ do
        $\mathsf{D}_T[t] \leftarrow \perp$
Return $\mathsf{D}_T$
```

Figure 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.

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 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.

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 overprivilege privacy issues. We also discuss how the design achieves the functionality and security goals from Section 4.2. We integrate minTAP with IFTTT due to its wide user base [32]. 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.

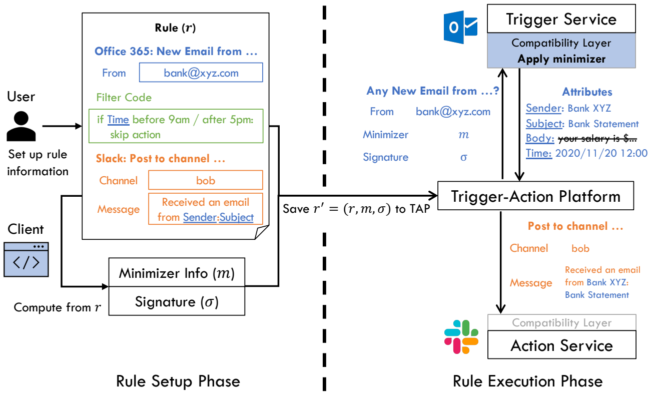


Figure 5: minTAP Framework. The blue-shaded background represents the components of minTAP: a client application and a modification to the existing IFTTT-compatibility layer of trigger services. The user creates a rule r, which is then transformed by the client into r' that contains minimizer information (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.

Design Overview. minTAP framework consists of two components (Fig. 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 correctly 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 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 (Section 6.1); (2) Setting up rules (Section 6.2); and (3) Rule execution (Section 6.3).

6.1 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

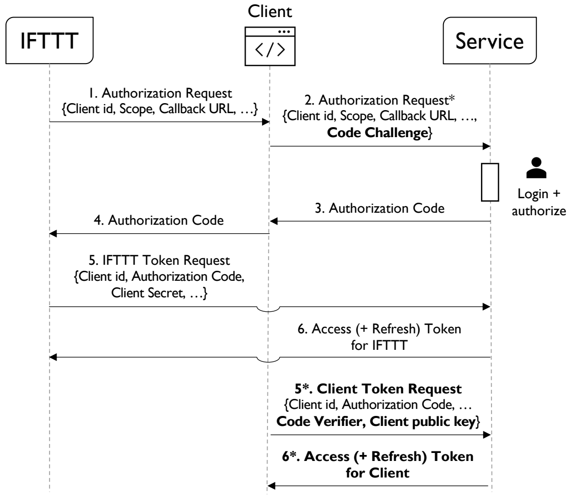

IFTTT
Client
Service
1. Authorization Request {Client id, Scope, Callback URL, …}
2. Authorization Request* {Client id, Scope, Callback URL, …, Code Challenge}
Login + authorize
3. Authorization Code
4. Authorization Code
5. IFTTT Token Request {Client id, Authorization Code, Client Secret, …}
6. Access (+ Refresh) Token for IFTTT
5*. Client Token Request {Client id, Authorization Code, … Code Verifier, Client public key}
6*. Access (+ Refresh) Token for Client

Figure 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.

usability. Therefore, we integrate the OAuth Proof Key for Code Exchange (PKCE) protocol [5] 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 (sk, pk). Fig. 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.

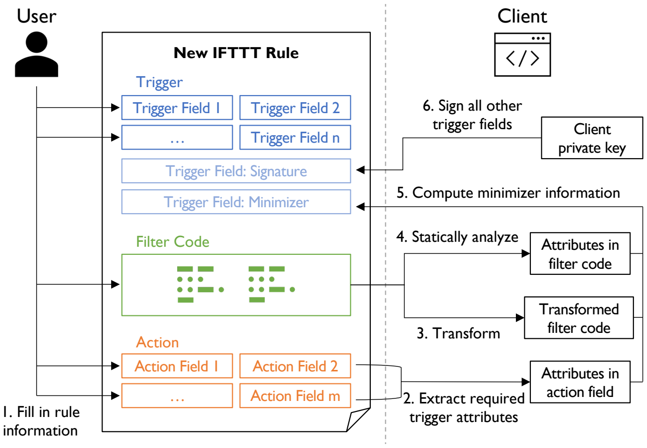


Figure 7: Rule setup phase. The left part represents a high-level abstraction of IFTTT's rule setup interface. The right part details the steps performed by the client (as a browser extension) in the background.

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 Appendix A 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.

6.2 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. 7 shows the workflow of the setup phase. The user creates a 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.

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 4.2), 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. 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.

6.3 Rule Execution Phase

When a rule executes, IFTTT contacts the trigger service to obtain data attributes [33]. This HTTP request from IFTTT bundles all the trigger fields as query parameters, including the auxiliary information of minimizer *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 (*m*) to apply the minimizer on the trigger attributes to sanitize unused values. As mentioned in Section 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) on the trigger data D_T. While running SMinimizer is straightforward, running DMinimizer could require executing untrusted client/user-provided code f'. We show the dynamic execution flow in Fig. 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.

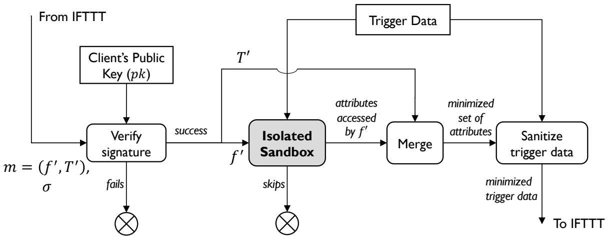


Figure 8: Figure shows the rule execution steps with *dynamic minimization* at the trigger service. IFTTT queries the service with the minimizer auxiliary information $m = (f', T')$ and the signature (σ). The trigger service applies DynamicMinimizer on the trigger data *m*, and responds with the sanitized trigger data to IFTTT.

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 7.

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. 1 and 2. Section 8 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 minimizer 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 Appendix A, 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.

7 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 7.1). Then, we analyze the privacy benefits of minTAP on this dataset in Section 7.2. Finally, we discuss our implementation and evaluate its performance overhead in Section 7.3.

7.1 Dataset

Existing IFTTT datasets [39, 46] 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. [19], 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.

7.2 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 6.2) 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.

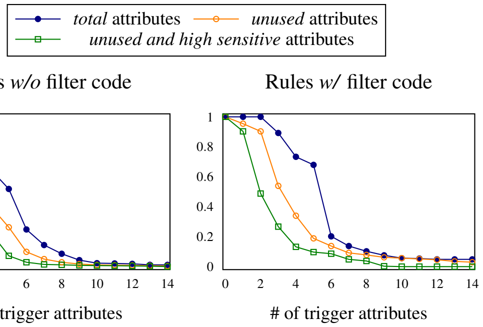


Figure 9: CDFs showing the percentage of rules that have at least *x* total / unused / unused-and-highly-sensitive attributes.

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 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 (see Appendix C for detailed keyword lists) and the fourth column denotes the percentage of rules that have at least one unused attribute with such keywords.

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. 9 shows a cumulative distribution of rules based on the number of 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 con-

[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.

ducted 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. 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. 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 Appendix C) 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. 10).

Rules *with* filter code. We show the CDFs of unused attributes for the 376 rules with filter codes in Fig. 9. Most of these rules contain very simple snippets with a few lines of code (left part of Fig. 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 one, sanitizing 1.7 attributes more on average. We show one of these rules in Appendix B. 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. 11).

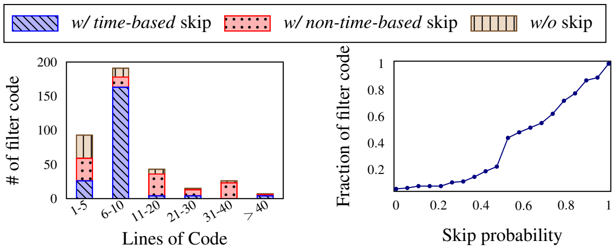


Figure 11: Filter code characterizations. (**left**) Histogram of filter codes based on lines of code. (**right**) CDF of the simulated skip probability for time-based filter code.

7.3 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 [35] 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 [31], 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.

7.3.1 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

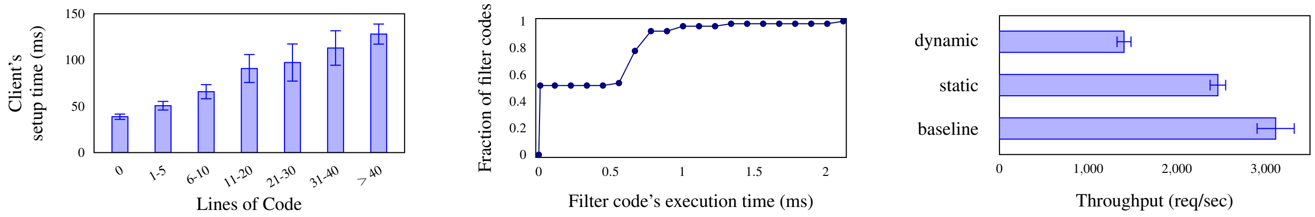


Figure 12: Evaluation results of minTAP. **(left)** The average execution time for the client during the rule setup. The rules are separated into different groups, based on the lines of code. **(middle)** CDF of the filter code's execution times in the trigger service's isolated environment. **(right)** The throughput of the trigger service for using static or dynamic minimzer, or baseline (i.e., w/o modification to the compatibility layer).

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. 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.

7.3.2 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 [39], 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. 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. 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 [39], 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 [29] for every rule, the additional overhead of minTAP is negligible.

8 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 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 7.3), 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 6.1, 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 minimization. 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* [17]. Distributed minimizers focus on minimizing data that is provided by multiple sources. This is relevant to IFTTT's emerging feature of *queries* [34] 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.

Multiple actions. TAPs sometimes allow rules to have *multiple actions*, a feature enabled, for example, for IFTTT's premium users. Our approach straightforwardly generalizes to multiple actions. Static minimization remains largely unchanged, tracking access of trigger attributes in the action fields of *all actions combined*. Similarly, dynamic minimization will track the access of trigger attributes in the sanitized filter code with the only modification that the trigger service will skip as a whole only if it detects calls to skip for *each* of the actions in the rule.

Encrypting trigger fields and attributes. The OTAP system encrypts trigger attributes and fields when *no* filter code is present [23]. 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 7.2 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. 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 [11] and OpenWhisk [10]) 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 [37].

9 Related work

We refer the reader to the recent work [13, 18, 20] outlining the state-of-the-art on securing TAPs. Our work is inspired by the principles of *least privilege* and *need-to-know* [44].

Privileges on TAPs. Prior work has shown that TAPs obtain overprivileged access to trigger/action APIs [25] allowing them to harvest private information without the user knowing [50] and opening for malicious rule makers to exploit TAP's privileges [12, 19]. This motivates our work.

The DTAP system protects the integrity of rules under a malicious TAP [25]. 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 [23]. 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 uses garbled circuits for rule execution [22]. 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 [50]. 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 [45, 53]. Besides requiring hardware changes to the TAP backends, current TEEs suffer from fundamental security design issues [21, 41, 47].

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" [6]. Antignac et al. [17] 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 [26]. Pinisetty et al. [43] 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. [14–16] 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.

10 Conclusion

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.

Acknowledgements We thank Adwait Nadkarni, Sandro Stucki, Musard Balliu, Benjamin Eriksson, and the anonymous reviewers for useful feedback. This work was partially supported by the University of Wisconsin-Madison Office of the Vice Chancellor for Research and Graduate Education with funding from the Wisconsin Alumni Research Foundation. It was also partially supported by the Swedish Foundation for Strategic Research (SSF), the Swedish Research Council (VR), and Facebook.

References

[1] Add All Day Event For Office. https://ifttt.com/applets/CfSd6B9w.

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

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

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

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

[6] General Data Protection Regulation (GDPR). Art. 5 Principles relating to processing of personal data. https://gdpr-info.eu/art-5-gdpr/, May 2018.

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

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

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

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

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

[12] M. M. Ahmadpanah, D. Hedin, M. Balliu, L. E. Olsson, and A. Sabelfeld. SandTrap: Securing JavaScript-driven Trigger-Action Platforms. In *USENIX Security Symposium*, 2021.

[13] M. Alhanahnah, C. Stevens, and H. 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.

[14] N. Anciaux, W. Bezza, B. Nguyen, and M. Vazirgiannis. Minexp-card: limiting data collection using a smart card. In *EDBT*, pages 753–756. ACM, 2013.

[15] N. Anciaux, D. Boutara, B. Nguyen, and M. Vazirgiannis. Limiting data exposure in multi-label classification processes. *Fundam. Informaticae*, 137(2):219–236, 2015.

[16] N. Anciaux, B. Nguyen, and M. 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.

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

[18] M. Balliu, I. Bastys, and A. Sabelfeld. Securing IoT Apps. *IEEE Security & Privacy Magazine*, 2019.

[19] I. Bastys, M. Balliu, and A. Sabelfeld. If This Then What? Controlling Flows in IoT Apps. In *ACM Conference on Computer and Communications Security*, 2018.

[20] Z. B. Celik, E. Fernandes, E. Pauley, G. Tan, and P. D. McDaniel. Program Analysis of Commodity IoT Applications for Security and Privacy: Challenges and Opportunities. *ACM Computing Surveys*, 2019.

[21] 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.

[22] Y. Chen, A. R. Chowdhury, R. Wang, A. Sabelfeld, R. Chatterjee, and E. Fernandes. Data Privacy in Trigger-Action Systems. In *IEEE Symposium on Security and Privacy*, 2021.

[23] Y.-H. Chiang, H.-C. Hsiao, C.-M. Yu, and T. H.-J. Kim. On the privacy risks of compromised trigger-action platforms. In L. Chen, N. Li, K. Liang, and S. Schneider, editors, *Computer Security – ESORICS 2020*, 2020.

[24] djblend777. Private links and photos from https://locker.ifttt.com - how to clear history? https://www.reddit.com/r/ifttt/comments/ao3sfr/private_links_and_photos_from_httpslockerifttttcom/, 2019.

[25] E. Fernandes, A. Rahmati, J. Jung, and A. Prakash. Decentralized action integrity for trigger-action iot platforms. In *Proceedings 2018 Network and Distributed System Security Symposium*, 2018.

[26] C. Flanagan and S. Qadeer. Predicate abstraction for software verification. In *POPL*, pages 191–202. ACM, 2002.

[27] J. Goguen and J. Meseguer. Security policies and security models. In *IEEE S&P*, 1982.

[28] 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.

[29] IFTTT. Applets Cookbook. https://platform.ifttt.com/docs/applets#applets-cookbook, 2018.

[30] IFTTT. Important update about the Gmail service. https://help.ifttt.com/hc/en-us/articles/360020249393-Important-update-about-the-Gmail-service, 2019.

[31] IFTTT. IFTTT: Creating Applets. https://platform.ifttt.com/docs/applets, 2020.

[32] IFTTT. IFTTT: Number of Users and Online Services. https://platform.ifttt.com/plans, 2020.

[33] IFTTT. IFTTT: Service API requirements. https://platform.ifttt.com/docs/api_reference, 2020.

[34] IFTTT. IFTTT's Glossary: Query. https://platform.ifttt.com/docs/glossary#query, 2020.

[35] M. Laverdet. Secure & Isolated JS Environments for Node.js. https://github.com/laverdet/isolated-vm, 2020.

[36] D. Lee. Uber concealed huge data breach, 2017. http://www.bbc.com/news/technology-42075306.

[37] B. Loring, D. Mitchell, and J. 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.

[38] J. A. Martin and M. Finnegan. What is IFTTT? How to use If This, Then That services. Computerworld. https://www.computerworld.com/article/3239304/what-is-ifttt-how-to-use-if-this-then-that-services.html, 2019.

[39] X. Mi, F. Qian, Y. Zhang, and X. Wang. An empirical characterization of ifttt: ecosystem, usage, and performance. In *Proceedings of the 2017 Internet Measurement Conference*, pages 398–404, 2017.

[40] Microsoft Power Automate. https://flow.microsoft.com/, 2020.

[41] K. Murdock, D. Oswald, F. D. Garcia, J. Van Bulck, D. Gruss, and F. Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[42] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of program analysis*. Springer, 1999.

[43] S. Pinisetty, T. Antignac, D. Sands, and G. Schneider. Monitoring data minimisation. *CoRR*, abs/1801.02484, 2018.

[44] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[45] S. Schoettler, A. Thompson, R. Gopalakrishna, and T. Gupta. Walnut: A low-trust trigger-action platform, 2020. https://arxiv.org/pdf/2009.12447.pdf.

[46] B. Ur, M. Pak Yong Ho, S. Brawner, J. Lee, S. Mennicken, N. Picard, D. Schulze, and M. 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.

[47] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T. F. Wenisch, Y. Yarom, and R. 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.

[48] Q. Wang, P. Datta, W. Yang, S. Liu, A. Bates, and C. 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, New York, NY, USA, 2019. Association for Computing Machinery.

[49] Q. Wang, W. U. Hassan, A. Bates, and C. A. Gunter. Fear and logging in the internet of things. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*. The Internet Society, 2018.

[50] R. Xu, Q. Zeng, L. Zhu, H. Chi, X. Du, and M. Guizani. Privacy leakage in smart homes and its mitigation: IFTTT as a case study. *IEEE Access*, 7:63457–63471, 2019.

[51] Zapier. https://zapier.com, 2020.

[52] Zapier. Zapier Platform CLI Docs. https://platform.zapier.com/cli_docs/docs, 2020.

[53] I. Zavalyshyn, N. Santos, R. Sadre, and A. Legay. My House, My Rules: A Private-by-Design Smart Home Platform. In *EAI MobiQuitous*, 2020.

A 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.

A.1 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 6.1, 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.

A.2 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.

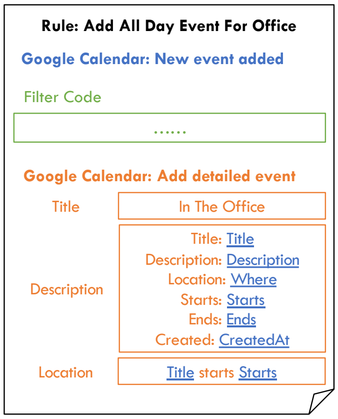
Modify user's inputs. This is not possible because the user only interacts with the client that is isolated from the IFTTT frontend code by the browser security model. The client eventually communicates the programmed rule and its signature directly to the IFTTT backend. At that point, the attacker can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

A.3 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 8).



Rule: Add All Day Event For Office

Google Calendar: New event added

Filter Code

......

Google Calendar: Add detailed event

Title: In The Office

Description: Title: Title
Description: Description
Location: Where
Starts: Starts
Ends: Ends
Created: CreatedAt

Location: Title starts Starts

```
if ( GoogleCalendar.newEventAdded.Where.indexOf("[some street address]") < 0 )
{
  GoogleCalendar.addDetailedEvent.skip();
} else {
  GoogleCalendar.addDetailedEvent.setDescription("In the office from "
    + GoogleCalendar.newEventAdded.Starts + " to "
    + GoogleCalendar.newEventAdded.Ends);
  GoogleCalendar.addDetailedEvent.setAllDay("true");
  GoogleCalendar.addDetailedEvent.setStartTime(GoogleCalendar.newEventAdded.Starts);
  GoogleCalendar.addDetailedEvent.setEndTime(GoogleCalendar.newEventAdded.Ends);
}
```

Figure 13: Example IFTTT rule with filter code.

All attributes	**With *static* minimizer**	**With *dynamic* minimizer**	
		when *not skipping*	**when *skipping***
Title, Description, Where, Starts, Ends, EventUrl, VideoCallUrl, CreatedAt	Title, Description, Where, Starts, Ends, ~~EventUrl~~, ~~VideoCallUrl~~, CreatedAt	Title, ~~Description~~, Where, Starts, Ends, ~~EventUrl~~, ~~VideoCallUrl~~, ~~CreatedAt~~	~~Title~~, ~~Description~~, ~~Where~~, ~~Starts~~, ~~Ends~~, ~~EventUrl~~, ~~VideoCallUrl~~, ~~CreatedAt~~

Figure 14: The outcomes of applying different minimizers to the example rule in Fig. 13.

B Example IFTTT Rule

In Fig. 13, we present an IFTTT rule [1] with filter code in our dataset. This rule converts a Google Calendar event that occurs in the user's office location to a detailed all-day event. We compare the outcomes of applying different minimizers in Fig. 14.

C Attribute Category Criteria

We list below the detailed criteria for how we determine which category each attribute belongs to in our evaluation of privacy benefits (Section 7.2). 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 `xxxxAt`, 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`).